Diss. ETH No. 17326

Modeling and Animation with Relaxed Connectivity Requirements

A dissertation submitted to
ETH Zurich

for the Degree of
Doctor of Sciences (ETH)

presented by
Dipl. Inform. Martin Wicke
born 3. May 1979
citizen of Germany

accepted on the recommendation of
Markus Gross, **ETH Zurich**, examiner
Hans-Peter Seidel, **MPI für Informatik Saarbrücken**, co-examiner

2007

Abstract

The choice of data representation is one of the most important design decisions when implementing algorithms or applications. This thesis will examine alternative representations in two core domains of computer graphics: Surface modeling and physically-based computer animation. It will be shown that algorithms based on less structured data representations can outperform those requiring more regularity. The first part of this thesis treats point-sampled surfaces, a minimally structured surface representation that has recently gained considerable momentum in the field of computer graphics. In the second part, algorithms for physically-based simulation in the context of computer animation will be considered.

As one of the foundations of computer graphics, surface modeling has been thoroughly researched, making use of various surface representations. These include spline patches, implicit surfaces, and polygonal meshes. Due to the availability of specialized hardware, triangle meshes have been dominant in the last years. Recently, point-sampled surfaces, representing the surface as an unorganized set of surface samples, have emerged as an alternative.

One of the long-standing problems with point-sampled surfaces is the representation of discontinuities. A rendering algorithm is proposed that handles geometric discontinuities. No preprocessing is necessary, and no per-sample information has to be provided, making the approach ideally suited for dynamic data.

Exploiting the flexibility of point-sampled surfaces for fast dynamic resampling, a painting system based on point-sampled surfaces is presented. The system provides an intuitive user interface and demonstrates the strengths of a less organized and more flexible surface representation.

To ensure interoperability, an algorithm converting point clouds to textured meshes is described.

Since point-sampled surfaces to not provide consistent connectivity, traditional animation methods like finite element methods are not directly applicable. To make animations of point-based surfaces possible, a method for thin shell simulation of point-sampled surfaces is presented. The geometric properties necessary for the evaluation of the shell energy functional are approximated using locally defined splines embedded in the surface.

Particle-based methods were at the heart of the earliest animation techniques, and are still popular in fluid simulation. Avoiding persistent connectivity, a method for computing elastic forces within a fluid simulation is proposed. This method allows for simulation of elastic objects within the fluid simulation. The unified

simulation method and material representation greatly facilitates phase transitions between liquids and solids.

Arguably the most important simulation methods in animation of deformable solids are finite element methods. These methods usually require a tetrahedral, or in some cases hexahedral mesh. A new finite element method presented in this thesis can handle meshes consisting of arbitrary convex polyhedra, and greatly increases the flexibility with regard to the discretization. Relaxing the requirements for the discretization yields advantages when considering topological changes in the simulation domain, e. g. introduced by cutting or fracture.

Kurzfassung

Die Wahl der Datenrepräsentation ist eine der wichtigsten Entscheidungen bei der Implementation von Algorithmen oder Anwendungen. In dieser Arbeit sollen alternative Repräsentationen in zwei Kerngebieten der graphischen Datenverarbeitung untersucht werden: Flächenmodellierung und physikalisch-basierte Computeranimation. Es wird gezeigt, dass Algorithmen, die auf weniger strukturierten Repräsentationen arbeiten, Vorteile gegenüber solchen bieten, die höhere Regularität nutzen. Der erste Teil dieser Arbeit behandelt punkt-basierte Oberflächen, eine Flächenreprs̈entation mit minimaler interner Struktur, die sich in den letzten Jahren zu einer wesentlichen Komponente in der graphischen Datenverarbeitung entwickelt hat. Im zweiten Teil werden Algorithmen zur physikalisch-basierten Simulation im Kontext der Computeranimation vorgestellt.

Oberflächenmodellierung ist eines der grundlegenden Probleme der graphischen Datenverarbeitung und Ansätze mit verschiedensten Flächenrepräsentationen wurden dementsprechend gründlich untersucht. Aufgrund der Verfügbarkeit von spezialisierter Hardware haben sich Dreiecksnetze zur dominanten Repräsentation entwickelt. Punkt-basierte Oberflächen sind alternative Diskretisierungen, welche die Fläche als unstrukturierte Punktwolke repräsentieren.

Ein hartnäckiges Problem von punkt-basierten Oberflächen ist die Darstellung von Unstetigkeiten. Ein Verfahren, das punkt-basierte Oberflächen mit Unstetigkeiten darstellen kann, wird vorgestellt. Da keine Vorberechnungen notwendig sind, und keine Information für jeden Punkt explizit spezifiziert werden müssen, ist der Ansatz für dynamische Daten geeignet.

Die Flexibilität von punkt-basierten Oberflächen kann ausgenutzt werden, um das Sampling auf Teilen der Fäche schnell zu ändern. An einer Applikation zum interaktiven Malen auf drei-dimensionalen Objekten werden Vorteile der punkt-basierten Oberfläche gegenüber weniger flexiblen Flächenrepräsentationen deutlich.

Um Interoperabilität mit herkömmlichen Verfahren sicherzustellen, wird ein Algorithmus zum Konvertieren von Punktwolken zu texturierten Dreiecksnetzen beschrieben.

Da punkt-basierte Oberflächen keine konsistente Konnektivität zur Verfügung stellen, können herkömmliche Simulationsverfahren wie finite Elemente Methoden nicht direkt verwendet werden, um solche Flächen zu animieren. Um Animationen von punkt-basierten Oberflächen zu ermöglichen, wird eine Methode zur

Simulation der Fläche als Schalenmodell vorgestellt. Die geometrischen Eigenschaften der Fläche, die für die Auswertung der Schalenenergie benötigt werden, werden mittels lokal definierten Splines angenähert.

Partikelmethoden waren ein Kernstück der frühen Arbeiten zur Computeranimation, und sind noch immer beliebt zur Simulation von Flüssigkeiten. Eine Methode, die elastische Kräfte in eine Flüssigkeitssimulation integriert, ohne auf gespeicherte Konnektivität zurückzugreifen, wird vorgestellt. Die Methode erlaubt es, elastische Materialien als Flüssigkeiten zu simulieren. Die einheitliche Simulationsmethode vereinfacht die Simulation von Phasenübergängen zwischen Flüssigkeiten und Festkörpern.

Die wahrscheinlich wichtigsten Simulationsmethoden in den Ingenieurswissenschaften, und auch in der graphischen Datenverarbeitung sowie der Computeranimation sind finite Elemente Methoden. Diese benötigen im Allgemeinen eine Diskretisierung des zu simulierenden Materials in Tetraeder oder in manchen Fällen kubische Elemente. In dieser Arbeit wird eine finite Element Methode vorgestellt, die volumetrische Netze bestehend aus beliebigen konvexen Polyedern als Diskretisierung akzeptiert. Die reduzierten Regularitätsanforderungen erhöhen die Flexibilität der Methode drastisch. Daraus ergeben sich Vorteile, die insbesondere in Simulationen, in denen sich — beispielsweise durch Brüche oder Schnitte — die topologische Struktur des Simulationsbereichs ändert, sichtbar werden.

Acknowledgments

First and foremost, I want to thank my adviser Markus Gross. His experience and expertise are invaluable, and his incessant encouragement and support were crucial to the successful completion of this thesis. I am glad that he (contrary to myself) always had the necessary critical distance to avoid getting lost in details.

One of the most pleasing characteristics of scientific research is the intense intellectual exchange with collaborating researchers. The work presented in this thesis is no exception, and many of the insights I have gained throughout these past four years are the result of discussions and scientific debate. I am grateful to Matthias Teschner, with whom I worked during my first years. I also want to thank Matthias Müller, whose interesting ideas have been a great inspiration. Many thanks also to Mark Pauly and Leo Guibas for inviting me to spend a productive and stimulating month in Stanford. Special thanks to Bart Adams. It was a pleasure to work with you, and I hope to continue our fruitful collaboration in the future. Finally, thanks to Mario Botsch for the great teamwork during this last year, and for playing the guitar.

Before I started working on my PhD at ETH Zurich, the adviser of my diploma thesis, Alexander Keller, introduced me to Computer Graphics and sparked my interest in pursuing an academic career in the field. Without him, this thesis would not have been written.

Last, but not least, I am indebted to my office mates. I learned a lot from Christian Sigg during our frequent discussions and our tournaments. Thanks also to Richard Keiser and Bernd Bickel for making our office a great place to spend my days in — and before deadlines, also my nights.

This work has been made possible by funding supplied by NCCR Co-Me, and KTI/CTI.

Contents

Chapter 1

Introduction

When modeling the real world in a computer, volumes and surfaces are discretized. Different types of discretizations vary widely in which operations they support. Hence, the choice of discretization and data representation has great impact on the design of algorithms. Choosing the right representation for a given problem is one of the most important design decisions early in the development process. There is a trade-off involved: Simpler data structures are easier to maintain and some basic operations are typically faster. However, the access and processing operators available are often restricted by the lack of structural information within the data representation. On the other hand, as representations get more and more sophisticated, they become harder to maintain under changes. The additional work might pay off in the end: Since the information is represented in a more organized fashion, and as more guarantees on the content are available, more powerful operations are often possible.

The field of computer graphics is mainly concerned with geometry. Consequently, geometry representations, either representing surfaces, or volumes, are of highest interest for computer graphics research. Traditionally, surfaces are represented with either spline patches or polygonal meshes. With the advent of specialized hardware, triangle meshes have gained additional importance. Triangle meshes are highly structured, with strong guarantees on topology and connectivity.

More recently, *point-sampled surfaces* have been developed as a surface representation. A point-sampled surface representation is essentially a set of sample points taken from the surface. Connectivity is not stored explicitly, the only connectivity information available is what can be extracted from spatial neighborhood relationships. Guarantees on topology are difficult to formulate, and require additional information on the input. The set of available surface operators is more limited than for triangle meshes. Texture discontinuities and sharp geometric features are difficult to represent using point-based surfaces.

For some applications, these drawbacks are worth accepting. Point-sampled surfaces are easier to maintain than triangle meshes. Since the points are unstructured

by definition, no additional structural information such as connectivity has to be carried along during manipulations. This is particularly noticeable for operations like surface resampling. Especially in modeling applications, the surface needs to be resampled frequently to adequately represent the changed surface geometry or appearance. However, in order to use point-sampled surfaces as a representation in a modeling tool, problems like the representation of discontinuities have to be overcome.

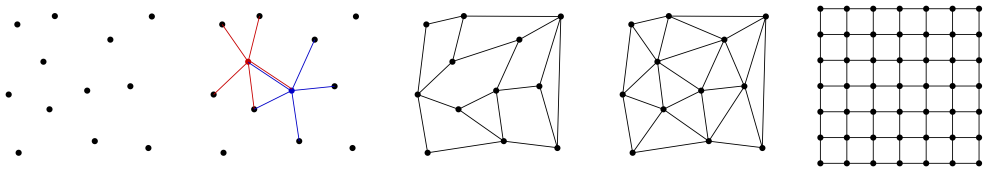
Computer animation increasingly relies on physical simulations. With the exception of thin shell simulation, working entirely on surfaces, such physical simulations require a volume discretization. For physically-based animation, the choice of data representation determines the simulation algorithms that can be applied, and vice versa.

In the area of fluid simulations, *Eulerian* approaches are very popular. These methods discretize space, and regular grids or tetrahedral meshes are used as underlying representations. For simulation of soft bodies, *Lagrangian* methods are dominant. Lagrangian methods discretize the material, not its embedding space. Hence, it is crucial that the discretization conforms well to the material shape. Tetrahedral meshes are used in almost all approaches. In these highly regular discretizations, important simulation methods such as the finite element method (FEM) have a particularly simple form.

Meshless approaches are an alternative for both fluid and soft body simulation. Uncoupled and weakly coupled particle systems sample the material with particles that are animated using scripted trajectories or interaction forces defined for pairs of particles. For fluid animation, the smoothed particle hydrodynamics (SPH) method has been used. Meshless fluid simulation techniques do not require connectivity, the necessary operators from vector algebra are replaced with approximate discrete operators that are derived assuming uniformity of the particle distribution.

On the other hand, meshless approaches for the simulation of deformable soft bodies typically do require some connectivity, although they do not require a consistent mesh. In other words, meshless approaches for elasticity computations relax the requirements for admissible connectivity compared to their mesh-based counterparts. The connectivity present in a consistent mesh partitions the discretized domain into a disjoint set of primitives. Often, for example in the case of tetrahedral or triangle meshes, these are further restricted to simplicial primitives. In contrast, meshless methods do not require a consistent connectivity that induces a disjoint partitioning of the domain. Commonly, the connectivity is simply defined as the neighborhood graph over the samples in the initial state of the material.

Figure 1.1 illustrates different levels of organization for space discretization. Meshless fluid simulation methods like SPH are on the far left of the scale (a), while stored connectivity is used in almost all solid animation techniques, including meshless approaches. Meshless elasticity simulations, as well as some types of mass-spring systems do no require a consistent mesh (b). Most mass spring



(a) (b) (c) (d) (e)

Figure 1.1: Different requirements for space discretizations. The degree of regularity increases from left to right, while the computational complexity of operators decreases. (a) Unstructured points without stored connectivity. (b) Point samples with stored connectivity (shown are 5 nearest neighbors for two points). (c) Irregular, but consistent mesh. (d) Mesh with only one primitive type (in this case, a simplicial mesh). (e) Regular mesh.

systems can work on a irregular mesh (c), while traditional FEM and mass-spring systems with better convergence guarantees require a mesh with a specific element type, such as a tetrahedral mesh (d). Eulerian Fluid simulations discretize space as a simplicial (d) or regular mesh (e).

1.1 Point-Based Modeling

Representing surfaces as a collection of sample points trades connectivity for flexibility and ease of maintenance under some operations. Without connectivity information available, most operators devised for meshes are not applicable, and ways to work around these restrictions have to be found. Research on point-samples surfaces first developed techniques for rendering. Methods for processing and modeling of point-sampled geometry followed. The increasing availability of data from laser range scanners accelerated this development, as meshing the huge amounts of data produced by these scanners is not always practical. Instead of triangulating these point clouds, they can be processed directly. Often, additional information about the local structure of the point cloud is available or can be inferred, such as per-sample normals, or optimal sample sizes, encompassing information about local sample spacing and anisotropy. This information can be used for subsequent modeling and high-quality rendering.

Despite efforts in recent years, the set of algorithms that are available for processing and modeling of point-sampled geometry is far from complete. A long-standing issue with point-sampled surfaces is the representation of discontinuities. Surface reconstruction methods for point-sampled surfaces have to fill the space between sample positions. It is an important assumption in these algorithms that the surface is smooth, hence, without explicit handling, discontinuities cannot be represented in point-sampled geometry.

There are clear advantages of point-sampled surfaces over more organized triangle meshes in applications heavy on dynamic resampling. This is of particular interest during the data acquisition process, when samples are generated and added to the model, as well as during shape modeling. Appearance modeling also requires dynamic resampling to accurately capture texture detail.

1.1.1 Contributions

In this thesis, the point-based surface representation is extended to handle arbitrary discontinuities in both geometry and texture. This is a crucial step to enable the use of point-based surfaces in general-purpose modeling tools. Additionally, an appearance modeling system leveraging the strengths of point-sampled surfaces is proposed. As an indispensable addition to the pool of tools for working with point-sampled surfaces, a method for converting point clouds to textured triangle meshes is presented. Specifically, the contributions regarding modeling with point-sampled surfaces are:

- A framework for discontinuity modeling for point-sampled surfaces. Discontinuities are handled explicitly, but only readily available high-level information has to be supplied during modeling, while the per-sample classification is computed during rendering. This makes the approach ideally suited for dynamic data and interactive editing.

- To date, appearance modeling for point-sampled surfaces relies on 2D texture painting, or relatively crude interaction metaphors on three dimensions. In this thesis, an intuitive appearance modeling system is presented. Using haptic input and a realistic brush input metaphor, this system makes use of the superior resampling capabilities of point-sampled surfaces. Both texture and fine-scale geometric surface detail can be edited in a natural way.

- As most of the tools available for geometric modeling work on triangle meshes, the content that is acquired and edited as a point cloud needs to be converted to a triangle mesh in order to use such tools. In this thesis, an algorithm is presented that converts a point cloud to a textured mesh, controlling the error in both geometry and texture.

1.2 Physically-Based Animation

As computing power grows, the complexity of models and scenes used to automatically or semi-automatically create content has exploded. At the same time, the art of computer animation has seen a transition from entirely hand-modeled scenes to scripted content, and more and more towards animations that are the result of physical simulations. Often, simulations used for animation place less

weight in physical accuracy, trading it for flexibility, or speed. The primary goal of animations is to look realistic, thus the criterion of *plausibility* replaces numerically quantifiable accuracy in most cases. Although accuracy implies plausibility to some extent, it is not always wanted. To the animator, *controllability* is more important. A purely physical simulation, while being correct in a physical sense, might curtail their creativity and produce unwanted effects, destroying the illusion it was supposed to create.

As outlined above, the choice of algorithm is inseparably joined to the choice of data representation and discretization. Each discretization has its own drawbacks and advantages. For instance, some phenomena like fluid spray or cloud formation are more amenable for simulation using particles-based approaches, while incompressible fluid simulation requires a Eulerian discretization of space.

Generally, as the flexibility in the discretization increases, the algorithms based on it become more versatile. While the optimal choice of discretization and simulation algorithm naturally depends on the problem at hand, a wide choice of feasible simulation methods makes a better adaptation to each individual problem possible.

1.2.1 Contributions

This thesis proposes several algorithms for physical simulation that require a less structured data representation than the current state of the art, and studies the effects for the resulting algorithms. To this end, methods for thin shell simulation, continuum elasticity and visco-elastic fluid simulation are proposed. For thin shells, the simulation does not need a consistent triangulation, and relies on point-based surfaces with stored connectivity as the underlying representation. It will be shown that adding elastic forces to a fluid simulation to create visco-elastic effects can be achieved without requiring stored connectivity. Finally, a finite element method requiring less regularity in the discretization is proposed. The contributions to the field of animation are:

- Thin shells can be simulated with higher-order FEM or using a geometric formulation on triangle meshes. Using local spline fits, the shell energy functionals can be discretized on point clouds with stored initial connectivity. A method using such *fibers* to approximate surface properties is presented. This is applied to thin shell simulation, including physical phenomena like plasticity and fracture.

- SPH is the most important particle method for fluid simulation. A method for enhancing SPH simulations with elastic forces without relying on stored connectivity is proposed.

- Finite element methods for continuum elasticity are usually implemented using tetrahedral or hexahedral elements. In this thesis, a finite element

method on arbitrary convex elements is proposed. This is particularly useful for simulations involving changes to simulation domain, such as incurred by cutting and fracture during the simulation. Adapting the discretization is considerably easier if elements are not required to have one specific shape.

1.3 Thesis Outline

This text is organized into two parts. While the first will deal with point-sampled surfaces in the context of modeling, the second part moves to simulation, first treating animation of point-sampled surfaces as thin shells, then moving on to more general topics in animation.

The next chapter will give an overview over past research and related work. In Chapter 3, fundamentals on point-sampled surfaces are discussed. The chapter covers surface definitions and some surface operators based on these definitions such as a ray-surface intersection, or inside/outside tests. Rendering techniques in both software and hardware are described. Finally, some acceleration data structures improving performance of nearest neighbor searches for point clouds are discussed. These are relevant not only in the context of point-sampled surfaces, but also significantly impact performance of particle-based simulation techniques such as SPH.

Chapter 4 presents a novel technique for representing discontinuities in point-sampled surfaces. Using CSG operations of smooth point-sampled surface patches, arbitrary discontinuities can be represented. This technique handles discontinuities explicitly, however, the actual per-sample evaluation of the discontinuity is deferred until rendering (or evaluation of an inside/outside test), such that the modeler does not need to supply low-level, per-sample information. The approach is ideally suited for dynamic data, as no preprocessing is necessary.

Chapter 5 describes a surface editing system for point-sampled geometry. It supports input with a haptic device and uses a physically-based brush model as its main interaction metaphor. Using point-sampled surfaces in this context allows for fast dynamic resampling of modified surface areas. This makes modifications of the small-scale geometry of the surface possible, creating effects such as brush imprints in viscous paint.

In Chapter 6, an algorithm for computing texture atlases containing the color and normal information stored in a point cloud is presented. Special attention is paid to ensuring that the information present in the point cloud is adequately represented in the textures created. An iterative refinement process using image-based texture creation makes sure that the texture error is within a user-supplied tolerance. Finally, a customized texture packing algorithm produces texture atlases usable with generic mesh processing software.

The second part of this thesis moves toward topics in animation. Chapter 7 presents a method for animating point-sampled objects as thin shells. The nec-

essary surface properties are computed using *fibers*, one-dimensional splines embedded in the surface. Thus, the method relies on stored connectivity but does not require a consistent triangle mesh. After discussing the physics of thin shells, a discretization of the thin shell energy using *fibers* is proposed. A wide range of material properties can be modeled using this discretization, including fracture and plasticity.

In Chapter 8, the SPH fluid simulation framework is discussed in detail, including methods for enforcing incompressibility of the simulated fluid. Pure SPH does not store any connectivity between particles. A method for integrating elastic forces into the SPH framework without introducing additional connectivity requirements is presented. The connectivity-free formulation of elasticity makes it particularly easy to model phase transitions. The consequences of abandoning stored connectivity are discussed: The range of materials that can be simulated using this method is restricted.

Chapter 9 describes a novel finite element method working on arbitrary convex elements. While most established finite element methods require their elements to be of a particular shape, like tetrahedral, or hexahedral cells, the new method allows for a much larger class of shapes. Recently developed barycentric coordinates for convex polytopes in three dimensions make this generalization possible. The modifications necessary to a standard FEM approach are discussed in detail. The more relaxed requirements for the discretization enable simpler topological changes of the discretized domain. This is demonstrated using cutting simulations. Additionally, sliver elements that have plagued FEM simulations can be removed from a given discretization as the output is allowed to be a complex of arbitrary convex elements.

Finally, Chapter 10 concludes this thesis by offering an overview of my findings, and giving an outlook on possible future research directions.

1.4 Publications

This thesis is based on technical contributions that have been published in peer-reviewed conference proceedings and journals.

- Parts of the discontinuity representation for point-sampled surfaces presented in Chapter 4 have been published as [220]:

  Wicke, M., Teschner, M., and Gross, M. CSG Tree Rendering of Point-Sampled Objects. In *Proceedings of Pacific Graphics'04*, pages 160–168, 2004.

- Parts of the painting system described in Chapter 5 have also been described in [4]:

Adams, B., Wicke, M., Dutré, P., Gross, M., and Teschner, M. Interactive 3D Painting on Point-Sampled Surfaces. In *Proceedings of the Symposium on Point-Based Graphics'04*, pages 57–66, 2004.

- The conversion algorithm to convert from point-sampled surfaces to triangle meshes, as described in Chapter 6, was previously published as [218]:

  Wicke, M., Olibet, S., and Gross, M. Conversion of Point-Sampled Models to Textured Meshes. In *Proceedings of the Symposium on Point-Based Graphics'05*, pages 119–124, 2005.

- The method for thin shell simulation presented in Chapter 7 has been published as [219]:

  Wicke, M., Steinemann, D., and Gross, M. Efficient Animation of Point-Based Thin Shells. Computer Graphics Forum 24:3, pages 667–676, 2005.

- The enhanced SPH algorithm described in Chapter 8 has been previously described in [217]:

  Wicke, M., Hatt, P., Pauly, M., Müller, M., and Gross, M. Versatile Virtual Materials Using Implicit Connectivity. In *Proceedings of the Symposium on Point-Based Graphics'06*, pages 137–144, 2006.

- Chapter 9 presents a finite element method that is accepted for publication in [216]:

  Wicke, M., Botsch, M., and Gross, M. A Finite Element Method on Convex Polyhedra. Computer Graphics Forum 26:3, 2007. to appear.

Chapter 2

Related Work

In this chapter, previous research is discussed that is relevant to the topics treated in this thesis. While it should give an overview of the state of the art and provide some context for the reader, more specialized references are given in the description of the technical contributions in the following chapters.

2.1 Point-Sampled Surfaces

Representing surfaces as sets of points was first proposed by Levoy and Whitted as an intermediate stage for rendering [129]. With the advent of optical scanners that can generate huge point clouds (for example [98, 110, 163, 167, 179]), the question of how to render and edit such data became more pressing. Creating meshes from scanned point clouds is a challenging task, especially in the presence of noise. For large models, this option quickly becomes impractical.

Instead, the geometry can be represented by the sample points alone, as an *unstructured point cloud*. For a comprehensive treatment of the topic, see [93]. A comparison of points-sampled surfaces to other surface representations can be found in [92].

No explicit connectivity is stored in a point cloud — the connectivity induced by spatial neighborhood is used where a neighborhood relationship is needed. This connectivity is not *consistent* in the sense that it does not define a disjoint partitioning of the surface in a way a polygonal mesh does. Useful concepts easily defined for meshes, such as the one-ring, cannot be used as they have no equivalent if no consistent connectivity is available. As a consequence, algorithms working on polygonal or triangle meshes cannot be transferred to unstructured point clouds.

When only sample points of the surface are given, defining the actual surface is an important step towards making point-sampled surfaces a useful tool in graphics. The most common surface definition is an implicit *moving least squares* (MLS) surface [5, 10, 11, 18, 127, 128].

This implicit surface definition can then be used to render the surface using raytracing techniques [3, 6, 8, 11, 210]. While producing images of high quality, raytracing or ray-casting is often not appropriate when interactive or even real-time rendering is necessary.

Rendering techniques based on point splatting fill this gap. Samples are smeared out in object space and projected onto the viewing plane [165, 231, 232]. This technique also allows for hardware acceleration [37, 38, 233]. Although it is not natively supported by current graphics hardware, interactive frame rates are possible for typical model sizes up to a million samples. For large models, adaptive rendering techniques have been developed [59, 156, 175], making it possible to display huge models at interactive rates. These techniques exploit the lack of structure in the point cloud to quickly add and remove samples as needed.

Of course, existing models in form of meshes can be converted to point clouds for rendering or editing using tools for point-sampled geometry. Some of the early rendering algorithms include methods for converting textured meshes to point clouds [94, 165]. *Layered depth cube* (LDC) sampling, or a variety thereof, has emerged as the standard technique for generating a point cloud from a triangle mesh. Here, parallel rays are cast onto the mesh from three orthogonal directions, and all intersections with the surface become sample points. If the rays' distance is d, the maximum sample distance in the sampling created with this approach is $\sqrt{3}d$ [94, 165].

The inverse operation is more involved. One way to generate a triangle mesh from a point-sampled surface is to use global Delaunay triangulation on the sample points [14, 15, 65]. This method can be extended to guarantee watertight meshes [66], or adapted to tolerate noise [67]. Others have applied local measures to find a triangulation [31]. The meshes produced by triangulation are unnecessarily large: They contain every sample point as a vertex and need to be simplified in order to be useful for further processing. Other approaches to surface reconstruction use implicit surfaces as an intermediate representation [101, 156], however, these approaches discard all additional information present in the point cloud, such as colors and normals. The problem of converting point-sampled objects to textured meshes is treated in detail in Chapter 6.

With the capability of acquiring and rendering point clouds, they become interesting as a general surface representation. Consequently, algorithms to directly process point clouds without first converting them to triangle meshes were developed. Automatic filtering techniques for noisy scanned data greatly increase the efficiency of model generation using optical scanners [180]. A set of tools for cleaning of noisy point clouds acquired by an optical scanner was presented by Weyrich et al. [215]. Pauly et al. used frequency analysis for smoothing and feature enhancement [157] and implemented feature detection [160] as well as simplification [158] algorithms for point clouds.

The lack of explicit connectivity also simplifies some operations, such as resampling. This has been used in a variety of applications, such as the adaptive

rendering methods mentioned earlier. The ability to adaptively resample the surface has proven particularly useful for interactive editing and modeling.

2.1.1 Modeling

Szeliski and Tonnesen were the first to use particles for surface modeling [197]. Zwicker et al. used point-sampled surfaces in a surface editing application, their Pointshop3D software [229]. Pointshop3D offers a set of tools to paint, filter and sculpt point-sampled objects. Painting is performed by projecting a brush footprint bitmap. Reuter et al. [171] developed a Pointshop3D plugin to interactively texture an object using a point-sampled multi-resolution surface representation. From a interaction point of view, these tools are rather crude. An interactive painting system with an intuitive user interface based on point sampled surfaces is described in Chapter 5.

Point-sampled surfaces have also been used for shape modeling [161], and CSG operations [2,161]. CSG operations create sharp geometric features such as edges, corners, and creases. Surface meshes can easily represent such discontinuities, and research into CSG rendering for meshes has mainly focused on acceleration techniques (e. g. [51,83,169,173,194,221]). The question of how to represent such sharp features in point cloud data is much more challenging. Surface splatting, the prevalent rendering and surface reconstruction algorithm used in interactive editing systems, blurs sharp edges.

Adams and Dutré [2] resample the area around the edges using very small surfels in order to push the blurred area below the pixel area. This method is fast, but can never completely conceal the point-sampled nature of the edge. Magnifying the edge will result in blurring.

In contrast, Pauly et al. [161] introduce a special surfel class that explicitly represents an edge. This surfel carries two normals, and is rendered as two clipping surfels with identical centers. This surfel is moved onto the edge using MLS projection, and the affected area is resampled to fill holes. This method always provides sharp edges, but does not support more complex intersection types like corners. Zwicker et al. [233] present a method for hardware accelerated rendering that can clip surfels with one or more clipping planes. However, when more than two clipping planes affect one surfel, the results are ambiguous. Complex intersections can not be represented with this approach. A method for representing sharp features of arbitrary complexity in point-sampled geometry is proposed in Chapter 4.

Point-sampled surfaces have been successfully applied to problems in computer animation [96, 112, 113, 149, 159, 162, 219]. In this context, the question of how to generalize the connectivity-free sampling paradigm to 3D naturally arises from research on animation of point-sampled surfaces. Previous research on volumetric point sampling in computer graphics has focused on visualization and representation of volumetric data [52, 88, 230]. In physically-based animation, particle sys-

tems and molecular dynamics approaches are closest to unstructured point clouds. The next section will give some context in the topic of computer animation.

2.2 Physically-Based Animation

As scenes generated with computer graphics methods become more and more complex, automatic and semi-automatic techniques help animators to handle such large modeling tasks. Early work on automatic generation of animations used scripted animations [86] or weakly coupled particle systems [170]. These approaches are easy to control and extremely versatile, and have been used to simulate effects such as fire [124, 164, 170], smoke and clouds [139, 186], waves and flowing water [77, 100], rain [79], and crowd behavior [172].

Using pairwise, distance-dependent interaction forces between particles yields spatially coupled particle systems [202, 203]. This method is inspired by molecular dynamics, and often uses the Lennard-Jones potential as interaction force. Molecular dynamics approaches can be used to model a wide range of materials, from solids to viscous liquids [204], as well as phase transitions between them [198, 204]. They have also been applied to physically-inspired surface modeling [197]. However, spatially coupled particle systems rely on parameters that are known to be hard to fine-tune. For stiff materials, these methods impose severe time step restrictions on the simulation. In the context of this thesis, spatially coupled particle systems are particularly interesting because they can be used to simulate solids without storing connectivity. A more detailed discussion and a comparison with a novel connectivity-free method can be found in Chapter 8.

Another connectivity-free particle method is SPH, which can also be used to model soft elastic, highly plastic objects as viscous fluids [64]. It is now mostly used in the context of fluid simulation.

2.2.1 Fluid Simulation

The SPH simulation method was originally developed for simulations in the context of astrophysics [81, 137]. A good overview is given in [132, 145]. Introduced by Desbrun and Gascuel to model soft plastic objects [64], SPH has become increasingly popular in computer graphics. At low resolutions, the method can be used interactively [146]. Boundary conditions, including complicated boundary shapes, moving boundaries [152], and interaction with other fluids [150] are relatively easy to implement. Visco-elasticity can be implemented in particle-based fluid simulations using dynamically generated springs [56, 198] or additionally running an elasticity model [112]. These methods require stored connectivity.

Smoothed Particle Hydrodynamics suffers from two major drawbacks at the discretization resolutions usually used in computer graphics applications: Compressibility and excess viscosity. The function approximation used in the SPH

framework uses kernel functions to obtain a continuous function from scattered samples. Hence, the velocity field is smoothed in each time step, resulting in numerical viscosity. These problems are discussed in more depth in Chapter 8.

Because of the problems mentioned above, Eulerian methods are dominant for high-quality fluid simulations in compute animation. First introduced by Foster and Metaxas [74–76], these methods discretize the fluid velocity field on a grid. The incompressible Navier-Stokes equation is used as a model for both liquids as well as smoke. A stable solution was presented by Stam [190]. Simulation using regular grids is computationally expensive, and adaptive methods based on octrees have been proposed for large problems [134, 184]. Still, these methods have problems representing complex boundaries. More flexible discretizations into tetrahedral cells solve this problem [70, 118].

Simulation of phase transitions require transfer of material between representations. Losasso et al. [136] show how to melt solids represented by Lagrangian tetrahedral meshes into a fluid simulation running on a Eulerian grid. The process of phase transitions, in particular freezing, is a complex physical phenomenon. For interactive animation, some approximate or stochastic model is usually preferred over a full physical simulation [116, 117].

Eulerian approaches can be used to model solids as high viscosity fluids [44], and rigid bodies by applying projection operators on the velocity field [43]. Simulating phase transitions is quite involved due to different discretization types involved [135]. Visco-elastic fluids can be simulated by integrating strain change over time [82].

2.2.2 Simulation of Elastic Deformable Bodies

While fluid simulation algorithms can be used to animate solid material [43, 44, 64], these methods do not offer a general solution to the simulation of deformable bodies. Elastic materials are usually discretized in a Lagrangian fashion. Starting with geometrically inspired deformation measures [199], several methods for animating deformable objects have been proposed. For a good overview over existing techniques in the area of physically-based animation, see the recent state of the art report by Nealen et al. [153].

The methods that are easiest to implement are mass-spring systems (some examples include [48, 166, 205]). These have been generalized to include area- or volume-preserving energy terms [40, 200]. The spring metaphor is stretched when considering these more general models, which are therefore often called particle systems. To avoid confusion, in this text, the term *particle system* is only used for models with no stored connectivity, such as the methods described in the previous sections. Convergence of mass-spring approximations is generally problematic, and the discretization significantly influences the solution.

For some applications like games, where physical accuracy is not nearly as important as fast evaluation and stability, approximate models have been devel-

oped [148]. These methods have virtually no requirements on the input data, in particular, no consistent connectivity is necessary. Another particle-based approach also uses particles to sample the volume, but computes standard strain energy using an MLS approximation [149].

By far the most common simulation methods in computer graphics are finite element methods.

Finite Element Methods

The finite element method is a very powerful tool to discretize and solve partial differential equations, offering strong guarantees for convergence and accuracy. For an introduction to the topic, see for example [23].

Due to its higher computational cost, FEM was introduced to computer graphics later than simpler methods like mass spring models. FEM methods were first used in medical simulation and facial animation, where accuracy is important especially if quantitative analyses are performed (e. g. [57, 84, 119, 177]). For large simulations, adaptive methods have been developed [61, 62, 90].

For simulation using FEM, the simulation domain needs to be discretized into elements. Typically, the elements are chosen to be tetrahedral or, in some cases, hexahedral [151]. In the FEM framework, quantities are interpolated between the nodes using locally supported basis functions. For tetrahedral and hexahedral elements, these basis functions can be chosen to be piecewise linear or trilinear, yielding particularly simple equations. Even if nonlinear basis functions are considered to improve accuracy, the discretization is usually tetrahedral [174].

In order to create a more flexible FE method that does not require tetrahedral or hexahedral elements, basis functions for the new element shapes have to be found. FE methods based on more flexible basis functions have been available for two-dimensional problems since Wachspress introduced basis functions for arbitrary convex polygons [208, 209]. Chapter 9 describes how a finite element method based on new three-dimensional interpolation functions [73, 108, 109] can be constructed.

Cutting and Fracture

The easiest way to incorporate topological changes into a simulation is to split the material along element boundaries, thus avoiding restructuring of the discretization [147]. Using this simple approach, newly created surfaces must conform to the initial discretization of the simulation domain. While this is acceptable in applications with hard real-time constraints, it is a severe limitation for animation. Accurate cutting along arbitrary trajectories is not possible.

O'Brien et al. [154, 155] apply continuous remeshing to make the discretization conform to the crack surface after fracture events. Bielser et al. [32, 91] subdivide

tetrahedra locally in order to accommodate the crack surface. This approach frequently leads to ill-conditioned elements, which cannot be used robustly in a FEM simulation [181]. Steinemann et al. [192] try to avoid these problems by snapping the simulation nodes to the cut trajectory. The remeshing process is highly non-trivial, because ill-conditioned sliver tetrahedra have to be avoided in order to guarantee a stable simulation.

An alternative to remeshing is the virtual node algorithm introduced by Molino et al. [141]. Instead of actually splitting simulation elements, the element that is to be split is duplicated. The newly created surface is embedded in both elements and used for rendering and collision detection. The pitfalls of remeshing can be entirely avoided using this scheme, however, each element can be split at most three times. Hence, the original mesh resolution limits the resolution of fracture patterns or cuts, requiring a high input resolution for realistic results.

Meshless approaches for modeling elastic solids [149] do not require an underlying tetrahedral mesh. Interactions between particles are evaluated using the material distance between the particles. In the case of fracture, the material distance within the object might not be equal to the Euclidean distance any more, prompting recomputation of the shape functions of nearby particles [159]. This process is local, however, coverage issues complicate the fracturing process, making it necessary to adjust the particle distribution near cracks. Steinemann et al. [193] have therefore reintroduced explicit connectivity into a particle-based simulation. A distance graph is used to approximate the material distance. After cutting, this graph is modified to reflect the new situation. Still, resampling is necessary to maintain an adequate discretization near cracks.

Cloth and Thin Shells

Objects that are almost two-dimensional cannot be simulated using the methods developed for volumetric objects. Even if a finite thickness is assumed, methods discretizing the volume become numerically unstable as the thickness of the object decreases. Both cloth and thin shells are in this class of materials. In the literature, cloth simulation typically refers to objects with a planar rest state. Such object can be simulated using thin plate energies, which are a special case of the thin shell energy functional.

Thin shells have been introduced to Computer Graphics by Terzopoulos et al. in 1987 [199], who described their behavior in terms of the first and second order metric tensors. However, the efficient numerical solution of these functionals is a major challenge. Mass-spring approximations to thin shell behavior are available [41,200], and FEM approaches using linearized energy terms have been developed [47, 119]. For the simpler case of cloth simulation, mass-spring systems are still the dominant simulation method (e. g. [22, 41, 42, 200]). This has sometimes been justified with the presence of strands and fibers in cloth, making a mass-spring system a more natural model than a continuum approximation [40].

For thin shell simulation, Cirak et al. [55] investigated the utility of subdivision surfaces. If subdivision surfaces are used to construct the required 2^{nd} order basis functions, their special properties guarantee that the resulting approximation is conforming.

A simple and efficient simulation method for thin shells has been proposed by Grinspun et al. [89]. Like the earlier work of Terzopoulos et al. [199] it is based on discretization of the metric tensors. These methods employ surface operators discretized on triangle meshes to compute the elastic energy. In particular, bending energy is represented by a discrete curvature operator [140], and stretching energy is approximated by edge lengths and triangle areas.

Recently, Guo et al. [96] have proposed a simulation method for thin shells that uses unstructured point clouds as the underlying representation. The point cloud is parameterized globally, and the surface metrics induced by the parameterization are used to define elastic energies. While this yields good results for most surfaces, the parameterization is not isometric, which can lead to artifacts. In Chapter 7, a different method for the simulation of point-sampled surfaces as thin shells is presented that avoids parameterization. Surface operators are defined using splines defined by neighboring sample points.

Part I

Modeling

Chapter 3

Point-Sampled Surfaces

Instead of using a triangle mesh or higher order polynomial spline patches, using only sample points on these surfaces has proven to be a fruitful approach to surface representation.

Abandoning the connectivity information present in more structured representations will make some operations more difficult, while simplifying others. It has turned out that especially in situations where dynamic resampling is necessary, and continuous remeshing is complicated or intractable, surfaces represented by unstructured point samples have advantages over meshes and other connected representations.

This chapter summarizes the basic concepts of surface representation using unstructured point samples. Section 3.1 introduces the notion of a moving least squares (MLS) surface, the most common implicit surface definition. This surface definition is useful for high-quality rendering using raytracing techniques. In Section 3.2, surface representation using a set of elliptical discs, so called *surfels*, will be discussed. These can be rendered by splatting them, thus making real-time rendering and interactive manipulation of point-sampled objects possible. Finally, Section 3.4 gives some details on the data structures typically used to accelerate algorithms working with unstructured point clouds.

3.1 Moving Least Squares Surfaces

Let $\mathcal{S}$ be a manifold surface and $P = \{\mathbf{x}_1 \dots \mathbf{x}_n\}$ a set of points on that surface. The goal is to find an implicit surface $\tilde{\mathcal{S}}$ that approximates $\mathcal{S}$. In practice, one defines a projection operator $\Psi(\mathbf{x})$ that maps any point to the closest point on $\tilde{\mathcal{S}}$.

A weight function or kernel $w_i(\cdot)$ determines how each point i influences its surroundings. The kernel is usually chosen as a smooth, non-negative function with a single maximum at $\mathbf{x}_i$. Section 3.1.5 gives more details on weight functions.

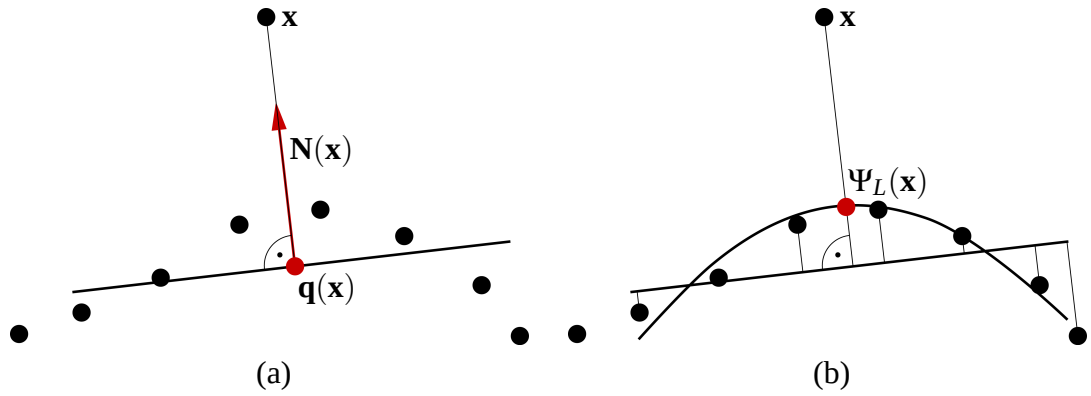


Figure 3.1: (a) Step 1: Fitting the plane. (b) Step 2: Fitting the polynomial.

Levin's MLS surface

Levin [127, 128] defines a projection operator Ψ_L as a two-step procedure. In a first step, a plane given by its normal $\mathbf{N}(\mathbf{x})$ and a point $\mathbf{q}(\mathbf{x})$ is fitted to the samples close to the query point $\mathbf{x}$. We find $\mathbf{q}(\mathbf{x})$ and a normalized $\mathbf{N}(\mathbf{x})$ such that

$$\sum_i w_i(\mathbf{q}(\mathbf{x}))\,(\mathbf{N}(\mathbf{x}) * (\mathbf{x}_i - \mathbf{q}(\mathbf{x})))^2 \tag{3.1}$$

is minimized, while fulfilling the additional constraint that

$$(\mathbf{x} - \mathbf{q}(\mathbf{x})) \parallel \mathbf{N}(\mathbf{x}). \tag{3.2}$$

This constraint ensures that the resulting operator is an orthogonal projection. To exclude trivial solutions to the above minimization, the solution should also fulfill $\sum_i w_i(\mathbf{q}(\mathbf{x})) \neq 0$. There might be several minima of (3.1), in such cases, the one closest to $\mathbf{x}$ is chosen. Figure 3.1 (a) illustrates the plane fit.

In a second step, a bivariate polynomial is fitted to the points, using a local coordinate system centered at $\mathbf{q}(\mathbf{x})$. Let $\mathbf{t}_1(\mathbf{x})$ and $\mathbf{t}_2(\mathbf{x})$ be two vectors forming an orthonormal basis with $\mathbf{N}(\mathbf{x})$, and let $\mathbf{x}_i' = [(\mathbf{x}_i - \mathbf{q}(\mathbf{x})) * \mathbf{t}_1(\mathbf{x}), (\mathbf{x}_i - \mathbf{q}(\mathbf{x})) * \mathbf{t}_2(\mathbf{x})]$ be an orthogonal projection of $\mathbf{x}_i$ onto the least-squares plane, expressed in the local coordinate system. The polynomial f is chosen to minimize the distance to samples parallel to $\mathbf{N}(\mathbf{x})$:

$$\sum_i w_i(\mathbf{q}(\mathbf{x}))\,\big(f(\mathbf{x}_i') - \mathbf{N}(\mathbf{x}) * (\mathbf{x}_i - \mathbf{q}(\mathbf{x}))\big)^2. \tag{3.3}$$

See Figure 3.1 (b) for an illustration. Finally, the MLS projection is defined as

$$\Psi_L(\mathbf{x}) = \mathbf{q}(\mathbf{x}) + f(\mathbf{x}')\mathbf{N}(\mathbf{x}) = \mathbf{q}(\mathbf{x}) + f(\mathbf{0})\mathbf{N}(\mathbf{x}). \tag{3.4}$$

Finding $\mathbf{N}(\mathbf{x})$ and $\mathbf{q}(\mathbf{x})$ requires a nonlinear optimization. In practice, simplifications of the above technique are of higher relevance.

3.1.1 Simplified MLS Surface

The problematic part of the surface definition above is the nonlinear optimization in the first step of the algorithm. Adamson and Alexa [5,6] introduce a simplification of the procedure. As noted by Amenta and Kil [18], the simplified procedure defines a different surface. The differences, however, are not noticeable in most applications.

Instead of minimizing (3.1) in the first step of the projection, Adamson and Alexa minimize

$$\sum_i w_i(\mathbf{x}) \left[\mathbf{N}(\mathbf{x}) * (\mathbf{x}_i - \mathbf{q}(\mathbf{x}))\right]^2, \tag{3.5}$$

always evaluating the weight functions at the query point. Since the weights are now independent of $\mathbf{N}(\mathbf{x})$ and $\mathbf{q}(\mathbf{x})$, the minimization is simplified significantly. $\mathbf{q}(\mathbf{x})$ can be computed as a weighted average of sample positions:

$$\mathbf{q}(\mathbf{x}) = \frac{\sum_i w_i(\mathbf{x})\mathbf{x}_i}{\sum_i w_i(\mathbf{x})}. \tag{3.6}$$

The normal of the least squares fit (3.5) can be computed as the eigenvector to the smallest eigenvalue of the weighted covariance matrix of the sample points [5].

The polynomial fit in the second step of the projection procedure also uses the weights $w_i(\mathbf{x})$ instead of $w_i(\mathbf{q}(\mathbf{x}))$. In practice, f is often chosen to be a polynomial of degree one. In this case, the result of the polynomial fit is $f = 0$, and the second step can be omitted altogether. The simplified projection operator becomes

$$\Psi_S(\mathbf{x}) = \mathbf{q}(\mathbf{x}). \tag{3.7}$$

Recently, Guennebaud and Gross [95] proposed a MLS surface reconstruction technique based on sphere fits instead of plane fits. This techniques drastically improves the quality of the reconstruction especially when the surface is only sparsely sampled.

3.1.2 Surface Normals

The normal of the polynomial f at $\mathbf{q}$ is usually taken to be the surface normal of $\tilde{\mathcal{S}}$. For the simplified procedure described in Section 3.1.1, this means that the normal of the projected point $\Psi_S(\mathbf{x})$ is $\mathbf{N}(\mathbf{x})$.

If Hermite data is available for the sample points, a least squares normal can be computed analogously to (3.6):

$$\mathbf{N}(\mathbf{x}) = \frac{\sum_i w_i(\mathbf{x})\mathbf{N}_i}{\sum_i w_i(\mathbf{x})}, \tag{3.8}$$

where $\mathbf{N}_i$ is the normal given at point $\mathbf{x}_i$.

Alexa and Adamson [10] show that these normals are not necessarily perpendicular to $\tilde{\mathcal{S}}$, and propose a way of computing the actual surface normal. Again, the differences are negligible for most applications.

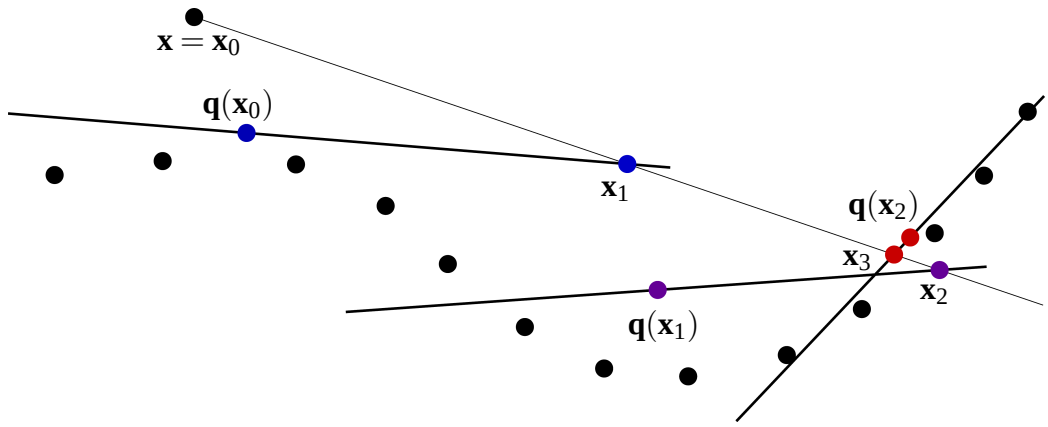


Figure 3.2: Raytracing using iterative MLS plane fits. The results of the first three iterations are shown. $\mathbf{q}(\mathbf{x}_0)$, $\mathbf{x}_1$ are colored blue, $\mathbf{q}(\mathbf{x}_1)$ and $\mathbf{x}_2$ of the second iteration are shown in purple, and $\mathbf{q}(\mathbf{x}_2)$ and $\mathbf{x}_3$ in red.

3.1.3 Raytracing

The main use for the MLS projection operators in the context of computer graphics is modeling and raytracing. For use in raytracing, a ray-surface intersection has to be defined. Adamson and Alexa [10] have proposed the following iterative procedure to intersect a ray $\mathbf{x}+t\mathbf{d}$ with the MLS surface $\tilde{\mathcal{S}}$. We assume that $\mathbf{x}$ is close to $\mathcal{S}$. This is easily achieved by using a bounding volume hierarchy to cull rays [6].

1 **set** $k = 0$, $\mathbf{x}_0 = \mathbf{x}$
2 **repeat**
3 **set** $k = k+1$
4 **set** $\mathbf{x}_k = \mathbf{x} + \frac{\mathbf{N}(\mathbf{x}_{k-1}) * (\mathbf{x} - \mathbf{q}(\mathbf{x}_{k-1}))}{\mathbf{N}(\mathbf{x}_{k-1}) * \mathbf{d}}\,\mathbf{d}$ // intersect plane and ray
5 **until** $\|\mathbf{x}_k - \mathbf{x}_{k-1}\| < \varepsilon$ // until convergence
6 **if** $\|\mathbf{q}(\mathbf{x}_k) - \mathbf{x}_k\| < \varepsilon_2$ **then**
7 **return** $\mathbf{x}_k$
8 **else**
9 **return** no intersection

Algorithm 3.1: Ray-Surface intersection

Algorithm 3.1 iteratively computes the least squares plane, and intersects the ray with that plane. If the iteration converges, it converges to the point on the ray that is closest to $\tilde{\mathcal{S}}$. If this point lies on the surface, then an intersection has been found. Figure 3.2 illustrates the procedure. The animations shown in Chapter 7 were rendered using this technique.

3.1.4 Inside/Outside Test

Using the projection operator $\Psi(\mathbf{x})$ as well as normal information, we can design a simple inside/outside test for MLS surfaces. A point is outside an object bounded by $\mathcal{S}$ if the normal at its projected position points in its direction.

$$[\mathbf{x} \textit{ outside } \mathcal{S}] \Leftrightarrow [(\Psi(\mathbf{x}) - \mathbf{x}) * \mathbf{N}(\Psi(\mathbf{x})) > 0] \tag{3.9}$$

Due to the definition of the MLS surface, this test only works reliably close to the surface, and will fail if the surface is not adequately sampled [17].

3.1.5 Weight Functions

The choice of weight function is a critical parameter in the surface definition. It determines the smoothness of the surface as well as the performance of the projection. The kernel w_i for a point $\mathbf{x}_i$ should have the following properties:

Smoothness The weight function should be smooth. The continuity of the weight function determines the continuity of the resulting surface [128].

Monotony There should be a single maximum at $w_i(\mathbf{x}_i)$, i. e. the influence of the sample is strongest at the sample point itself. More (local) maxima lead to additional problems when solving the minimization described in Section 3.1.

Locality The function should go smoothly to zero as the evaluation point is moved further away from $\mathbf{x}_i$. Together with monotony, this implies non-negativity. Many weight functions have only local support. In this case, a sample only contributes to its neighborhood. Thus, acceleration data structures can be used to quickly compute the relevant set of neighbors (see Section 3.4).

Most weight functions are radially symmetric, i. e. of the form $w_i(\mathbf{x}) = w(\|\mathbf{x}_i - \mathbf{x}\|) = w(d)$. Gaussian kernels are a popular choice due to their C^∞ continuity:

$$w_G(d) = e^{-\frac{d^2}{\sigma^2}}. \tag{3.10}$$

Here, σ is a parameter determining how much the samples are to be smeared out in space. High values of σ lead to a smoother surface. A major drawback of Gaussian kernels is their global support. In practice, this means that all sample points have to be considered for each projection. Often, clipped Gaussians are used instead:

$$w_{CG}(d) = \begin{cases} w_G(d) & d < c\sigma^2, \\ 0 & \text{otherwise.} \end{cases} \tag{3.11}$$

If clipped Gaussians are used, the surface loses its continuity. This effect is not noticeable if c is chosen large enough.

Polynomial approximations to Gaussians have several advantages over clipped Gaussians. They have local support, and they can be chosen such that all derivatives vanish at the domain boundary. If the polynomial is chosen carefully, polynomial weights can be evaluated much more efficiently than Gaussian weights. Several such weight functions have been introduced by Wendland [212]. An example with C^2 continuity is

$$w_W(d) = \begin{cases} (1-\frac{d}{r})^4(1+\frac{d}{r}) & d < r, \\ 0 & \text{otherwise.} \end{cases} \tag{3.12}$$

The parameter r determines the support radius of the kernel.

Where performance is critical, even simpler weights may be necessary. Wald and Seidel [210] use linear weights in their interactive raytracing system for point clouds.

Not all commonly used kernels are radially symmetric. If the sampling density is not isotropic, elliptic kernels can greatly improve the quality of surface reconstruction. Each sample point i is equipped with tangent axes $\mathbf{t}_1^i$ and $\mathbf{t}_2^i$. Their lengths determine the size of the kernel in that direction. With $\mathbf{d} = \mathbf{x}_i - \mathbf{x}$, the weight is computed as

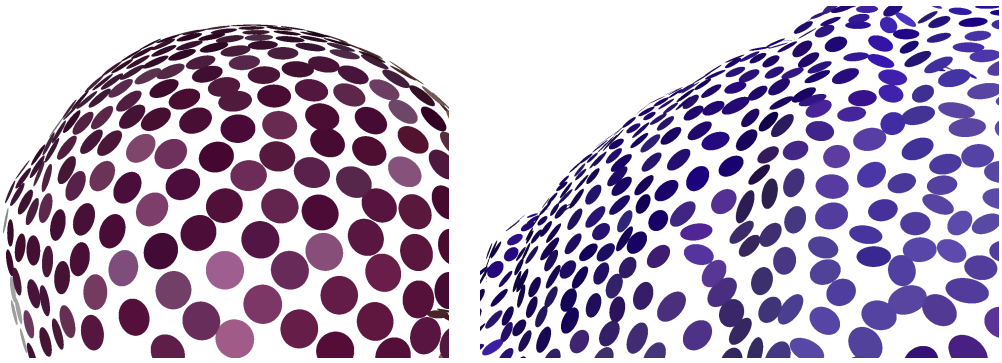
$$w_E(\mathbf{d}) = w\left(\sqrt{\frac{(\mathbf{t}_1^i * \mathbf{d})^2}{\|\mathbf{t}_1^i\|^4} + \frac{(\mathbf{t}_2^i * \mathbf{d})^2}{\|\mathbf{t}_2^i\|^4}}\right), \tag{3.13}$$

where $w(\cdot)$ is a radially symmetric weight function. The weight functions used for MLS approximations share some characteristics with SPH kernels, discussed in Section 8.1.1.

3.2 Surface Splatting

While MLS surfaces are of great importance in modeling, rendering them is computationally expensive. Therefore, *surface splatting* [165, 231, 232], analogous to rasterization for triangles, is used to render point-sampled surfaces in interactive applications.

In the following, we will consider *surfels*, small circular or elliptical disks. In the context of rendering, *splat* is used synonymously with surfel. In order to render a surfel, we need its position and tangent axes. If the splat is circular, normal and radius are usually substituted for tangent axes for memory efficiency. In order to guarantee a hole-free surface, the radii or tangent axes need to be chosen such that neighboring surfels overlap [223]. A weight function is associated with each surfel, which is used to blend colors and normals of overlapping splats. This weight function needs to have local support to make interactive splatting computationally feasible. Clipped Gaussians are the most popular choice [231]. The discontinuity at the boundary of the splat is not a problem here as its effects are small compared



(a) (b)

Figure 3.3: A surface represented with circular (c) and elliptical (d) surfels. Elliptical surfels can handle irregular and anisotropic sampling better.

to the discretization error of the 8-bit RGB color space. Figure 3.3 shows surfaces sampled with circular and elliptical surfels. Elliptical surfels require more memory, but are superior especially when anisotropic sampling is considered.

Each surfel in rendered by projecting it onto the screen. This results in an ellipse in screen-space, which can then be rasterized. For each pixel, an extended depth test first determines whether this splat is visible at this point. If this is the case, a weight is computed using the surfel's projected weight function, and the weighted color contributions are added to the framebuffer. Once all surfels have been processed, each pixel's color is re-normalized with the sum of blending weights accumulated for that pixel.

Lighting can either be computed per surfel or per pixel. In the latter case, normals are averaged using the splat weight functions, and shading is performed in a second pass [231].

The splatting algorithm described above may lead to aliasing artifacts if the projected kernels become smaller than the pixel size. A solution to this problem is EWA splatting [231]. Here, the kernels are depth-pass filtered before sampling them at the screen resolution. The projected weight function of each splat is convolved with a Gaussian. Using elliptic Gaussians as weight functions, the result of this convolution is again an elliptic Gaussian. This is then used as blending weight as before.

3.2.1 Extended Depth Test

Fragments generated during ellipse rasterization should only be rendered if they belong to a visible surface. Since the actual surface $\mathcal{S}$ is not known, a fuzzy depth test is used to determine the blending behavior [165]. This depth test has three possible outcomes: A surfel can be clearly hidden by the surface in the framebuffer, and is hence discarded. It can be clearly visible, and thus overwrites the

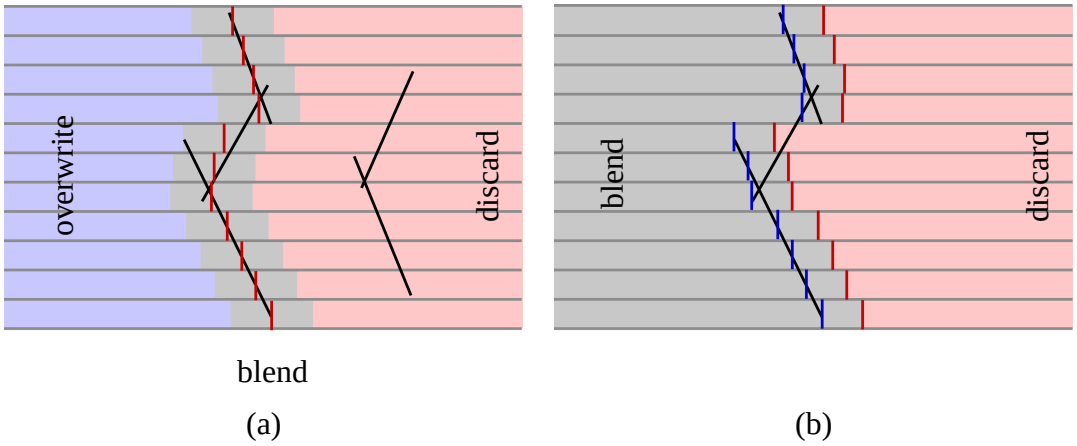


Figure 3.4: Extended depth test for splatting. The red lines show the content of the depth buffer. (a) 3-way depth test. If the depth tolerance ε_z is chosen too small or too large, blending results will be wrong. (b) Hardware implementation. In a first pass, the frontmost surfel is computed (blue depth values), its depth value is translated away from the viewer and stored (red depth values). A simple depth test using these values determines visibility.

framebuffer content, or it is very close to the surface in the framebuffer, and its contribution is added to the current surface. Consider an incoming fragment with a depth value of z_f. The framebuffer currently stores a depth value of z at this pixel. The action taken during rasterization is then

$$\begin{array}{ll} z_f < z - \varepsilon_z & \text{overwrite} \\ |z_f - z| < \varepsilon_z & \text{blend} \\ z_f > z + \varepsilon_z & \text{discard} \end{array}$$

Figure 3.4 (a) shows an example. Note that the choice of the depth tolerance ε_z is critical for the rendering result. If ε_z is too large, close-by surfaces are blended together, if it is too small, blending will be disabled even within a contiguous surface, leading to shading discontinuities. To remedy these problems, ε_z is usually chosen dependent on the average surfel size. The best quality can be achieved by choosing ε_z depending on the average surfel size and orientation at the current pixel, as well as the surfel size and orientation of the currently rasterized surfel.

3.2.2 Hardware Acceleration

Interactive rendering times for point-sampled models of high complexity can only be achieved with hardware support. Recently, specialized hardware has been developed that implements splatting as described above [214]. Since the advent of vertex and fragment programs, commodity graphics hardware can also be used

to render point-sampled surfaces. Several modifications and simplifications of the original algorithms are necessary, however.

Current hardware rendering algorithms [37, 233] need three passes instead of the two used in software rendering. In a first pass, a depth image is computed that is used to determine visibility. The second pass performs per-surfel shading and accumulates weights and weighted color contributions. A final pass is needed to normalize the colors by dividing each pixel's color by the sum of accumulated weights. Only the first two passes have object order complexity, while the complexity of the third pass only depends on the framebuffer size. Even for medium-sized models, its contribution to the overall complexity is negligible.

On newer hardware, floating point framebuffers are available, which makes deferred shading possible [38] (see also Section 5.7). The deferred shading pass can be integrated into the the third rendering pass used for normalization and does not add much to the overall complexity.

Visibility Computations

Since current graphics hardware does not support reading and writing to the depth buffer at the same time, the visibility computations have to be distributed to two passes. In the first pass, only depths are written, colors and weight functions are not evaluated. In a vertex shader, each surfel is moved away from the viewing plane by ε_z. If $z_{(x,y)}$ is the depth of the frontmost surfel at each pixel (x,y), the depth buffer now contains $z_{(x,y)} + \varepsilon_z$ in each pixel. The contents of the framebuffer after the first pass is the upper bound for the depth if fragments that will be blended in the second pass. In the second pass, a regular depth test culls fragments with a depth greater them $z_{(x,y)} + \varepsilon_z$.

Compared to the original fuzzy z-test, the first pass decides which surface is visible, while the second pass culls fragments that belong to hidden surfaces. Figure 3.4 (b) illustrates the hardware implementation of the fuzzy depth test.

Although not identical to the original fuzzy z-test, the hardware implementation has proven to be a good approximation. Apart from the fact that it needs two passes over the geometry, its major drawback is that the depth tolerance ε_z can only be chosen dependent on the frontmost surfel in each pixel. In fact, a constant ε_z is most common. This can lead to artifacts when models with varying splat sizes are rendered.

3.3 Sampling

Contrary to mesh-based representations, where the topology is explicitly given by the connectivity of the mesh, the topology of a point cloud is implicitly defined by neighborhood relationships of the sample points. Therefore, it is important that

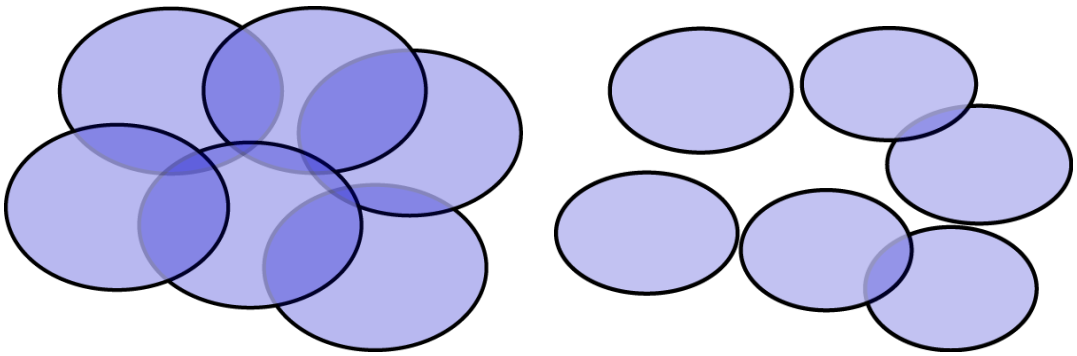


Figure 3.5: (a) When the surfel radii are chosen appropriately, splatting or MLS reconstruction yields a contiguous surface. (b) If the support radii of the weight functions do not overlap, holes appear in the surface.

the surface is *adequately sampled*. What exactly is meant by adequate sampling depends on the reconstruction algorithm used.

For surface splatting, *coverage* and *separation* are the most important issues: The surfels have to be close enough that their reconstruction kernels overlap. If this is not the case, holes appear in the surface [165] (see Figure 3.5). Also, if two surface sheets are close to each other, the surfel radius should be small enough to ensure that the fuzzy z-test can separate the two surface sheets. Note that the latter criterion only applies if the depth tolerance ε_z can be chosen on a per-fragment basis. For simple hardware implementations, where ε_z is constant over the framebuffer, surfel sizes have no impact on the separation of surface sheets.

Very similar criteria apply for MLS surfaces. If the support radius of the weight functions associated with the sample points is too small, the surface will break up into separate components. Also, if surface sheets come too close to each other, the plane fits used in the MLS projection become unreliable, and the sheets cannot be separated robustly [17].

Under certain conditions, the surface $\tilde{\mathcal{S}}$ reconstructed from the sample points can be proven to be geometrically close and homeomorphic to the input surface $\mathcal{S}$. We call such a reconstruction a *good* reconstruction of the surface. The necessary conditions for a provably good MLS reconstruction [68, 120] are very similar to the criteria used for surface reconstruction techniques relying on Delaunay triangulation [13, 14, 16, 34]. The predicate that is used in most proofs concerning surface reconstruction is the concept of ε-sampling. Let $\mathcal{S}$ be a surface and P a set of samples taken from the surface. The *medial axis* M of the surface is the set of points that have more than one closest point on $\mathcal{S}$. For each point $\mathbf{x} \in \mathcal{S}$, the local feature size $\Lambda(\mathbf{x})$ is defined as the distance of $\mathbf{x}$ to the medial axis:

$$\Lambda(\mathbf{x}) = \min_{\mathbf{q} \in M} \|\mathbf{x} - \mathbf{q}\|. \tag{3.14}$$

Using the local feature size, we can define a criterion for adequate sampling: A set of samples $P \subset \mathcal{S}$ is called an ε-sample of $\mathcal{S}$ if for each $\mathbf{p} \in P$, there is a point $\mathbf{q} \in P$, $\mathbf{p} \neq \mathbf{q}$, which is close to $\mathbf{p}$ compared to the local feature size in $\mathbf{p}$:

$$\|\mathbf{p} - \mathbf{q}\| < \varepsilon \Lambda(\mathbf{p}), \tag{3.15}$$

for some small ε. Several variations of this definition have been proposed [68,120].

Knowing that a point set is an ε-sampling of a surface, we can prove that a good reconstruction of this surface can be computed using a Delaunay-based triangulation [12, 13]. Similar guarantees can be derived for MLS surfaces [68, 120]

ε-sampling is a very strong requirement. In particular, ε-sampling leads to an infinite sampling density around sharp features. Checking whether a given point cloud is an ε-sample involves computing the medial axis, which is a challenging task by itself. Therefore, simpler measures are used to determine adequate sampling in practice. In many applications, making sure that adjacent surfels overlap such that no holes appear in the surface is sufficient.

3.4 Acceleration Data Structures

When working with point-sampled surfaces, and more generally meshless representations, many operations rely on local neighborhood relationships. Given a set of points $P = \{\mathbf{x}_1 \dots \mathbf{x}_n\}$ and a query point $\mathbf{x}$ (which may not be in P), two types of queries are of special interest:

1. Range query: return all points inside a sphere with radius r centered at $\mathbf{x}$.

2. Nearest neighbor query: return the m closest points to $\mathbf{x}$.

Either query requires $O(n)$ time if a brute force algorithm is used. Acceleration data structures can significantly lower the complexity of these queries. This section discusses kd-trees and spatial hashing, which are particularly well suited for the query types of interest. Other data structures, such as uniform grids, octrees, or bounding volume hierarchies, can be used as well. For a detailed comparison, see [29, 178].

3.4.1 kd-Trees

A kd-tree, or k-dimensional binary search tree, is a special case of a binary space partition tree with axis-aligned separating planes [28]. Its root cell contains all points in $\mathbb{R}^k$. In each level of the hierarchy, each cell is divided once by an axis aligned plane. Since cell boundaries are axis-aligned, inside/outside tests for points are extremely fast. See Figure 3.6 (a) for an illustration.

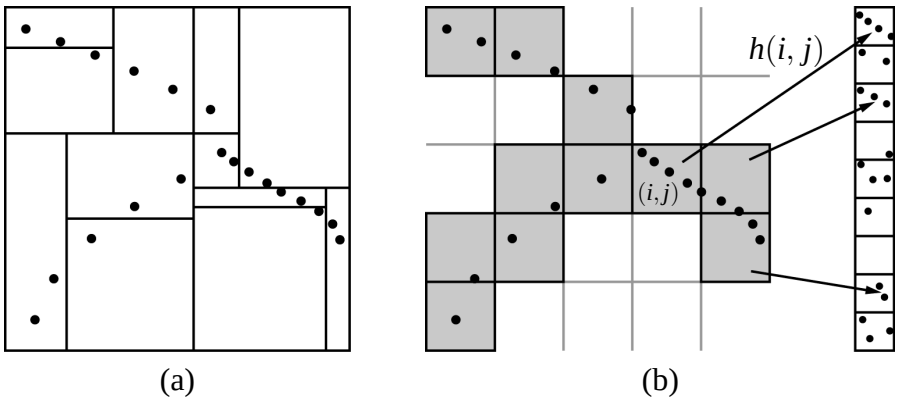


Figure 3.6: (a) A balanced kd-tree. The tree adapts to the spatial distribution of the input samples. (b) A hash grid. The grid structure is virtual, only the hash table (right) is actually stored. The hash function h maps cell indices to hash table indices.

Construction

Given the point set P, a kd-tree can be constructed by recursively splitting cells until a minimum number of points remain in the cell. For each cell, a good separating plane has to be found. Most commonly, the orientation of the planes is simply cycled through, and the median of point positions within the cell is used to determine the position of the separating plane. More sophisticated methods based on statistical analysis of the input points have been proposed for high-dimensional data [168, 213]. In the context of computer graphics, the trade-off between construction time and query time rarely justifies these more complex construction methods.

The construction time for a kd-tree is $O(n \log n)$. Adding, deleting, or moving samples will create an unbalanced tree and degrade search performance [71]. In practice, a kd-tree is rebuilt from scratch whenever the set of points changes.

Queries

Querying a kd-tree requires finding the cell containing the query point $\mathbf{x}$, and then computing all cells that are intersected by the sphere of radius r centered around $\mathbf{x}$ (see Figure 3.7 (a)). The relevant cells are found by backtracking up the hierarchy until a cell fully contains the sphere. All children of this cell are recursively tested for intersection with the query range. Points contained in intersecting cells are candidates to be returned by the query and have to be tested against the sphere [78]. See [19] for a good description of the algorithm.

Nearest neighbor queries in kd-trees are executed as a series of incremental range queries that are terminated once the sphere containing exactly the desired number of points does not intersect unvisited cells [19]. The average time complexity for both range and nearest neighbor queries is linear in the number of

returned points, and logarithmic in the total number of points [49]. However, this result is derived based on assumptions on the point distribution and cannot be straightforwardly generalized for more orderly point sets representing 2-manifolds embedded in $\mathbb{R}^3$.

3.4.2 Spatial Hashing

Hash grids are regular grids whose contents is stored in a hash table. The embedding space is discretized into cells, and each cell is assigned an index. The cell indices are passed through a hash function, resulting in an index in a hash table. All data associated with the cell is stored in this hash table entry. Hash grids provide fast access to stored data elements and are very easy to maintain. Since cells are not explicitly stored in memory, the searchable domain is not bounded. The memory requirements of a hash grid grow with the number of non-empty cells, but are independent of the number of empty cells. See Figure 3.6 (b) for an illustration.

Construction

To construct a hash grid, the search space has to be divided into cells. Most commonly, an infinite regular grid is used as the discretization. For some applications, it is advantageous to let cells overlap. It is critical that the cell(s) containing a point, as well as neighboring cells of a given cell can be computed efficiently. Each cell is assigned an index, or a tuple of indices. Even though it is not strictly required that these indices are unique, artificial collisions are introduced if they are not. For a regular, disjoint discretization of $\mathbb{R}^3$, the following function maps any point $\mathbf{x} = [x,y,z]$ to a 3-tuple $\mathbf{I} = (i,j,k)$ of integer indices:

$$\mathbf{I}(\mathbf{x}) = (\lfloor x/d \rfloor, \lfloor z/d \rfloor, \lfloor z/d \rfloor), \tag{3.16}$$

where d is the grid spacing. The cells in this indexing scheme are non-overlapping cubes with edge length d. A simple hash function for such index tuples has been proposed by Teschner et al. [201]:

$$h(i,j,k) = (ip_1 \ \mathbf{xor} \ jp_2 \ \mathbf{xor} \ kp_3) \ \mathbf{mod} \ s, \tag{3.17}$$

Where $p_{1,2,3}$ are prime numbers and s is the hash table size. In order to minimize collisions, s should also be prime [201]. More advanced hash functions might result in fewer collisions for small hash table sizes s, however, they are typically more complicated to compute. Which hash function is best depends on the application. For the algorithms discussed in this thesis, the additional cost incurred by an occasional collision is low, and consequently there is no performance gain from using more complicated hash functions. Note that since queries test each candidate for its distance to the query point, false positives due to hash collisions are eliminated and do not cause fundamental problems.

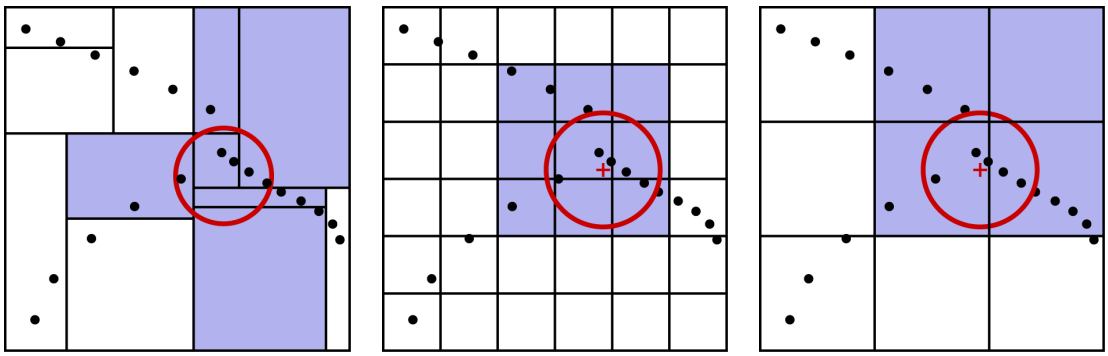


Figure 3.7: (a) Range query in a kd-tree. The highlighted cells have to be searched for sample points within the query radius. (b) Range query in a hash grid, where $d = r_{\max}$. The 9 cells closest to the query point have to be searched (27 in 3D). (c) $d = 2r_{\max}$. The 4 nearest cells (8 in 3D) need to be searched.

For any given point set, a perfect hash function can be constructed [126], avoiding collisions altogether while minimizing the hash table size. Perfect hash functions can greatly increase performance, although their construction is too expensive to be useful in dynamic settings.

To construct a hash grid from a given point set P, points from P are sequentially added to the hash grid, and assigned to their respective hash table entries. The construction time of a hash grid for a point set of size n is $O(n)$. Adding, deleting, or moving points in an existing hash grid is easy and requires only $O(1)$ time on average.

Queries

Range and nearest neighbor queries are straightforward to implement for hash grids. First, the hash cell (i, j, k) that contains the query point $\mathbf{x}$ is found. For range queries, all cells intersected by a sphere of radius r around $\mathbf{x}$ have to tested for points within the query range. If the maximum query range $r_{\max}$ is known, The cell spacing d should be chosen to be either $r_{\max}$ or $2r_{\max}$. In the former case, the cell (i, j, k) and all 26 neighboring cells have to be checked for points within the query radius. With $d = 2r_{\max}$, only the 8 closest cells to $\mathbf{x}$ have to be considered (see Figure 3.7). Although setting $d = r_{\max}$ allows for a more fine-grained discretization, experiments have shown that $d = 2r_{\max}$ is faster in almost all cases. Memory access time dominates the query times, hence less queried cells containing more candidate points are preferable over having to load more cells from memory. The average complexity of range query of fixed radius is linear in the number m of returned points.

The m-nearest neighbor query maintains a list of the closest points. While not enough points have been found, the points in neighboring cells are added to the list. Then, a sphere with radius d_m, the distance of the m'th nearest point in the list is searched for closer points. Once it is assured that there are no closer points, the query is complete. In situations where the spatial density of points does not vary much, the cell spacing d can be chosen such that with high probability, the n closest points lie in the 8 closest cells to a point $\mathbf{x}$.

3.4.3 Comparison

Both hash grids and kd-trees are used in applications presented in this thesis. Which data structure to use depends on several factors, and a final decision can often only be made after performance figures can be compares for several test cases. There are, however, some general rules that help determine which method is most appropriate.

Most importantly, kd-trees are fully adaptive data structures, while hash grids discretize space uniformly. In situations with strongly varying sampling density, kd-trees offer advantages over hash grids since the spatial resolution adapts to the local spatial density of the point set. However, the adaptivity comes at a cost. Constructing a kd-tree takes $O(n \log n)$ time as opposed to $O(n)$ for hash grids. While construction cost is not an issue for static data, it might become a limiting factor for dynamic point sets. However, even for many dynamic applications, constructing the data structure from scratch in every step is not the limiting factor as other parts of the algorithms have higher complexity.

Another distinguishing criterion is the query type needed by the application. Hash grids are most efficient with range queries of known constant radius. The grid spacing can then be adapted to the query radius, thus making sure that all queries can be completed by considering only 8 cells. If the sampling density does not vary significantly throughout the domain, this leads to a retrieval in $O(1)$.

Weaker guarantees are available for m-nearest neighbor queries with constant m: If the sampling density is constant throughout the domain, the grid spacing can be chosen such that it is likely that m-nearest neighbor queries terminate within the first 8 visited cells. However, constant sampling densities are rare. In general, kd-trees are better suited for m-nearest neighbor queries. Only for huge point sets, the logarithmic dependence of query times on the total number of points is a significant drawback of kd-trees.

Consequently, kd-trees are used for static point data of varying resolution in Chapter 5. The shell animation approach presented in Chapter 7 also uses kd-trees for m-nearest neighbor searches. The construction time for the kd-tree is not a limiting factor in this case as query times and simulation times dominate the overall complexity. Chapter 8 describes a fluid simulation technique that uses range queries of known constant radius. Hash grids have been used to speed up searches in this example.

Chapter 4

Modeling Discontinuities

While point-sampled surfaces are well suited to represent smooth manifolds, representing sharp features or texture discontinuities is difficult. Features of higher frequency than the finite sampling rate can represent are smoothed out by the reconstruction method.

In this chapter, a method is presented that explicitly represents sharp features in point-sampled objects. The geometry is stored as several *patches*, each of which represents a smooth surface. The intersections of these patches are evaluated during rendering according to given CSG operations, resulting in discontinuities at patch boundaries.

The surface patches are represented as sets of surfels, i. e. elliptical disks. Surface reconstruction during rendering is performed using EWA splatting. Sharp features are created by clipping surfels during rendering. Thus, for the sake of defining sharp features, the surfaces are considered to consist of intersecting and overlapping disks.

The next section describes how CSG operations can be implemented for point-sampled surfaces, before presenting a rendering algorithm that renders any CSG combination of surface patches. Finally, representing texture discontinuities explicitly using special discontinuity surfels is discussed.

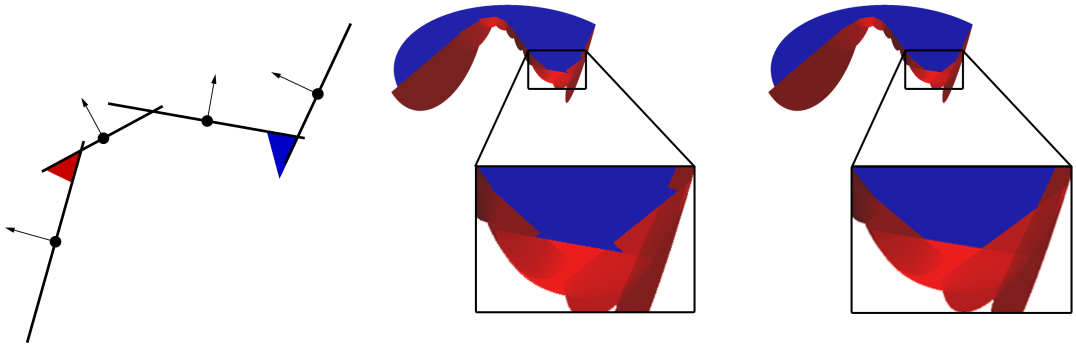
4.1 CSG Operations

CSG operations can be reduced to an inside/outside test. Consider two objects $\mathcal{A}$ and $\mathcal{B}$ with surfaces $\mathcal{S}_\mathcal{A}$ and $\mathcal{S}_\mathcal{B}$, respectively. The surface resulting from the CSG operation $\mathcal{A} \cap \mathcal{B}$ is constituted of the parts of surface $\mathcal{S}_\mathcal{A}$ that are *inside* $\mathcal{B}$ combined with the parts of surface $\mathcal{S}_\mathcal{B}$ *inside* $\mathcal{A}$:

$$\mathcal{S}_{\mathcal{A}\cap\mathcal{B}} = (\mathcal{S}_\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{S}_\mathcal{B} \cap \mathcal{A}). \tag{4.1}$$

The union operation is analogous:

$$\mathcal{S}_{\mathcal{A}\cup\mathcal{B}} = (\mathcal{S}_\mathcal{A} \cap \bar{\mathcal{B}}) \cup (\mathcal{S}_\mathcal{B} \cap \bar{\mathcal{A}}). \tag{4.2}$$



(a) (b) (c)

Figure 4.1: Using only the closest surfel for inside/outside classification leads to classification errors. (a) Blue: Incorrect outside classification. Red: Incorrect inside classification. (b) An intersection of two differently sampled surfaces, rendered using only the closest surfel for classification. (c) The same edge rendered using the two closest surfels for classification.

Throughout the algorithmic description, only union and intersection operations will be considered. Section 4.3 gives details on how to handle other operations.

In order to model complex objects, a CSG tree is stored. The leaves of the tree represent *patches*, i. e. point sampled surfaces without sharp edges. Each inner node stores a CSG operator, i. e. union or intersection. Hence, each inner node represents the result of applying its operator to its child nodes.

Each CSG operation may create sharp edges. If edges intersect, they can form corners of increasing complexity. It is essential that such corners can be represented correctly, independent of the number of patch primitives involved. Prior approaches based on surfel clipping are limited to sharp edges between two primitives [161], or purely convex corners [233].

4.1.1 Inside/Outside Classification

It is common practice to reduce the inside/outside classification for a point-sampled object $\mathcal{A}$ represented by a set of surfels $P_{\mathcal{A}}$ to a front-face/back-face test with respect to the closest surfel in $P_{\mathcal{A}}$ [2]. A point $\mathbf{x}$ is considered *inside* $\mathcal{A}$ if it is on the back-facing side of its nearest neighbor surfel s:

$$[\mathbf{x} \textit{ inside } \mathcal{A}] \Leftrightarrow [(\mathbf{x} - \mathbf{c}_s) * \mathbf{N}_s < 0]. \tag{4.3}$$

Here, $\mathbf{c}_s$ denotes the center, and $\mathbf{N}_s$ the normal of the closest surfel to $\mathbf{x}$. If a point $\mathbf{x}$ is classified as *outside* by a surfel s, we also say s *clips* $\mathbf{x}$.

As illustrated in Figure 4.1, this leads to classification errors. For many applications these errors are acceptable, however, when used in the context of rendering, the simple classification leads to disturbing jagged edges.

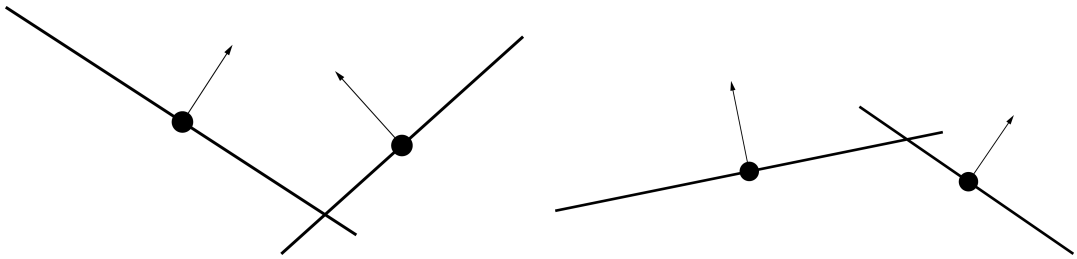


Figure 4.2: (a) Two *concave* surfels. Each surfel center $\mathbf{c}_{1,2}$ is on the front facing side of the other surfel. (b) Two *convex* surfels. At least one of $\mathbf{c}_{1,2}$ is on the back facing side of the other surfel.

The artifacts can be alleviated by incorporating information of the two closest surfels s_1 and s_2 for inside/outside classification. The inside/outside test according to (4.3) is performed for both s_1 and s_2, yielding results that are combined depending on the configuration of the closest surfels. We call two surfels s_1 and s_2 with centers $\mathbf{c}_1$ and $\mathbf{c}_2$ *concave* if s_1 *clips* $\mathbf{c}_2$ and s_2 *clips* $\mathbf{c}_1$. Otherwise, they are called *convex*. Figure 4.2 illustrates the configurations.

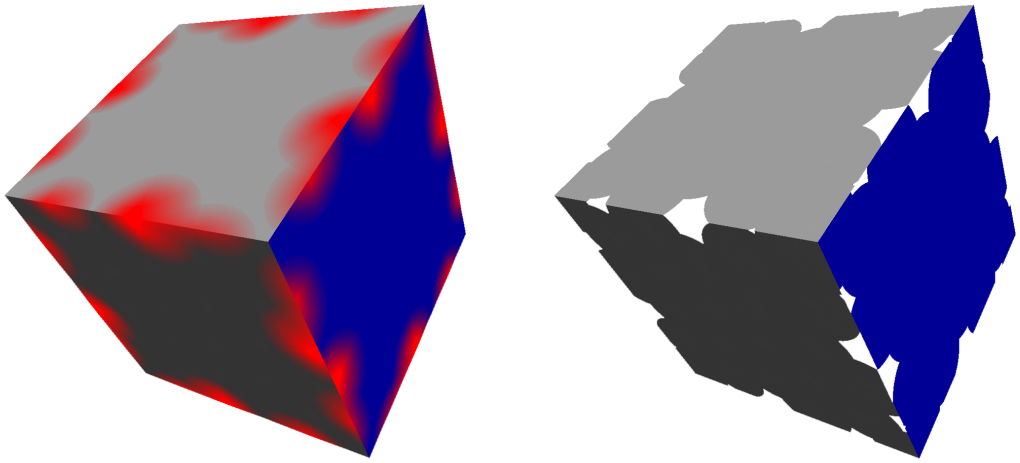
Then, we can define the inside/outside classification with respect to an object $\mathcal{A}$ using the two closest surfels $s_{1,2}$ of a point $\mathbf{x}$:

$$[\mathbf{x}\ \textit{inside}\ \mathcal{A}] \Leftrightarrow \begin{cases} \neg[s_1\ \textit{clips}\ \mathbf{x}] \vee \neg[s_2\ \textit{clips}\ \mathbf{x}] & \text{if } s_{1,2}\ \textit{concave}, \\ \neg[s_1\ \textit{clips}\ \mathbf{x}] \wedge \neg[s_2\ \textit{clips}\ \mathbf{x}] & \text{if } s_{1,2}\ \textit{convex}. \end{cases} \tag{4.4}$$

Figure 4.1 (c) shows the effect of the new classification scheme. Note that in three dimensions, one can construct situations where even the use of the two closest surfels for classification leads to non-intuitive clipping results. However, in practice, these situations are very rare, and the resulting artifacts are only noticeable in extreme close-ups. In order to avoid such artifacts, a consistent surface needs to be defined, for example using MLS projection. Pauly et al. [161] use the MLS surface to obtain an exact classification (see also Section 3.1.4) on a per-surfel basis. The classifications are computed in a preprocess, hence the high cost of the MLS projection is not a problem. The rendering algorithm described here requires evaluation of inside/outside classification on a per-fragment basis. Since the goal of this method is interactive rendering, using the MLS surface for inside/outside classification is not viable. This might change if recently developed hardware accelerated MLS techniques are used [95].

4.1.2 Surface Representation

When combining two surfaces $\mathcal{S}_\mathcal{A}$ and $\mathcal{S}_\mathcal{B}$ using a CSG operation $\circ$, the surfaces are classified into regions that are part of the combined surface $\mathcal{S}_{\mathcal{A}\circ\mathcal{B}}$, and those



(a) (b)

Figure 4.3: (a) A cube, with surfels discarded by a conventional CSG operation marked red. (b) The cube rendered without these surfels. Holes appear along the edge.

that are not. In the case of point-sampled surfaces, $\mathcal{S}_\mathcal{A}$ and $\mathcal{S}_\mathcal{B}$ are represented as discrete sets of sample points $P_\mathcal{A}$ and $P_\mathcal{B}$.

The surfaces are rendered using splatting. Thus, sample points contribute to the surface beyond their actual position: They influence the surface appearance wherever their weight function is non-zero. If all surfels whose center does not belong to $\mathcal{S}_{\mathcal{A}\circ\mathcal{B}}$ are discarded, holes appear along the patch edges (see Figure 4.3).

To make sure that the surface can be properly reconstructed, only surfels that do not contribute to the surface are discarded. Instead of computing an inside/outside classification for the surfel center, surfels are classified into one of three categories, depending on whether the surfels intersect another surface. For a surfel s which is part of $P_\mathcal{A}$, the classification is

- **surface surfels:** $\mathbf{c}_s \in \mathcal{S}_{\mathcal{A}\circ\mathcal{B}}$, and s does not intersect $\mathcal{S}_\mathcal{B}$. Surface surfels are splatted normally.
- **edge surfels:** s intersects $\mathcal{S}_\mathcal{B}$. These surfels might be clipped.
- **outside surfels:** $\mathbf{c}_s \notin \mathcal{S}_{\mathcal{A}\circ\mathcal{B}}$ and s does not intersect $\mathcal{S}_\mathcal{B}$. Outside surfels do not contribute to the resulting surface and are discarded.

Figure 4.4 shows the different surfel types. Edge surfels from $P_\mathcal{A}$ need to be clipped against $\mathcal{B}$, and vice versa. Each of the objects $\mathcal{A}$ and $\mathcal{B}$ might be a complex object consisting of several patches combined by CSG operations. Edge surfels are rendered using a specialized renderer which is described in Section 4.2.

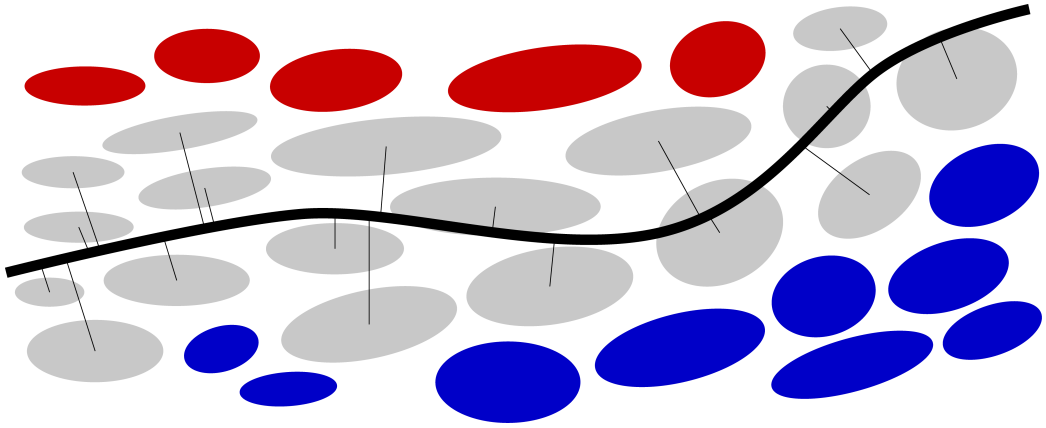


Figure 4.4: The different surfel types: Outside surfels (red), Edge surfels (gray), surface surfels (blue). If a surfel's distance to the intersection is smaller than its radius (or maximum extent, for ellipses), it is classified as edge surfel. The classification is conservative to avoid numerical problems: More surfels are classified as edge surfels than are strictly necessary.

4.2 Rendering Algorithm

Rendering geometry represented as described above requires that edge surfels are clipped against intersecting surfaces. Since all surfaces are represented by point samples, the algorithm identifies a set of intersecting surfels I_s for each edge surfel s. In a second step, all edge surfels are clipped against their respective set of intersecting surfels, using the CSG tree to correctly interpret the intersecting surface. Surface surfels are splatted as usual. The two images are merged in a final rendering pass.

4.2.1 Finding clipping partners

The surface is considered to be the union of surfel disks. Thus, a preprocessing step can identify the set of surfels I_s intersecting a surfel s. We call this set of surfels the *clipping partners* of s, as it is later used for clipped splatting. This section details an approach for finding clipping partners based on splatting. The edge surfels are splatted into a specialized *clip buffer* storing surfel lists, and intersecting surfels are identified by analyzing the clip buffer content. The complexity of this approach is $O(n)$ where n is the number of edge surfels, and it is fast enough to be executed in each frame. This makes it ideally suited for dynamic data, where interacting surfels need to be recomputed in every frame.

The idea of finding intersecting surfels using a rendered image is based on the observation that if two surfels intersect in object space, they also intersect in any projection. All edge surfels are splatted into a the clip buffer, which stores a set

of surfels $P_{(x,y)}$ for each cell (pixel). Whenever a fragment of a surfel s_1 from patch $P_\mathcal{A}$ is added to a cell (x,y), it is a potential clipping partner of all surfels $s_2 \in P_{(x,y)}$ belonging to patches $P_\mathcal{B}$ with $P_\mathcal{A} \neq P_\mathcal{B}$. If s_1 and s_2 intersect, s_2 is added to I_{s_1} and s_1 is added to I_{s_2}. A conservative estimate is used to determine whether s_1 and s_2 potentially intersect. To speed up computations, they are considered intersecting whenever the distance of their centers is smaller than the sum of their radii: $\|\mathbf{c}_{s_1} - \mathbf{c}_{s_2}\| < r_{s_1} + r_{s_2}$. For elliptical splats, the largest of the ellipse radii is used in the estimate. In pseudo-code, the algorithm can be written as:

Input: cell position (x,y), Surfel s_1
1 **foreach** $s_2 \in P_{(x,y)}$ **do**
2 **if** $s_1.patchid \neq s_2.patchid$ **and** $\|\mathbf{c}_{s_1} - \mathbf{c}_{s_2}\| < r_{s_1} + r_{s_2}$ **then**
3 **set** $I_{s_1} = I_{s_1} \cup \{s_2\}$
4 **set** $I_{s_2} = I_{s_2} \cup \{s_1\}$

Algorithm 4.1: Adding a fragment of s_1 to the cell (x,y)

Note that this method uses a 2D regular grid to speed up neighborhood computations in three dimensions. Other acceleration data structures, such as those discussed in Section 3.4, can be used instead. When a search data structure for all samples is constructed, range queries can be used to identify clipping splats. In the algorithm described above, the problem of identifying clipping partners is reduced from three to two dimension using splatting. Complex adaptive data structures such as kd-trees or quad-trees cannot easily be adjusted for this technique. It would of course be possible to replace the regular grid that constitutes the clip buffer by a hash grid. However, for typical scenes, the clip buffer is densely occupied by splats. A hash grid will therefore not offer tangible advantages, while requiring significant computational overhead due to random memory access.

4.2.2 Clipped splatting

After clipping partners have been determined for all surfels, the surface can be rendered. Clipping is performed during ellipse rasterization. Object space coordinates are computed for each fragment. These are then used to determine whether or not the fragment lies on the surface resulting from the CSG operations as stored in the CSG tree.

We consider a fragment at world space position $\mathbf{x}$ created by a surfel from patch $P_\mathcal{C}$. This fragment is certainly part of the surface $\mathcal{S}_\mathcal{C}$. The algorithm determines if $\mathbf{x}$ is part of the surface of the object represented by the root node of the CSG tree. The tree is traversed bottom up, starting at the leaf node representing $\mathcal{S}_\mathcal{C}$, until the root node is reached.

In each level of the hierarchy, $\mathbf{x}$ is part of a surface $\mathcal{S}_\mathcal{A}$, represented by some node $T_\mathcal{A}$ in the tree. The sibling of $T_\mathcal{A}$ in the CSG tree, $T_\mathcal{B}$, representing a surface

$\mathcal{S}_\mathcal{B}$, is combined with $T_\mathcal{A}$ with the CSG operator stored in their father node T'. The CSG operations (4.1) and (4.2) are applied to determine whether $\mathbf{x}$ lies on the surface represented by T'. Since $\mathbf{x}$ is known to be part of $\mathcal{S}_\mathcal{A}$, these equations simplify to

$$[\mathbf{x} \in \mathcal{S}_{\mathcal{A}\cap\mathcal{B}}] \quad \Leftrightarrow \quad [\mathbf{x} \textit{ inside } \mathcal{B}], \tag{4.5}$$

$$[\mathbf{x} \in \mathcal{S}_{\mathcal{A}\cup\mathcal{B}}] \quad \Leftrightarrow \quad \neg[\mathbf{x} \textit{ inside } \mathcal{B}]. \tag{4.6}$$

If $\mathbf{x}$ is on the surface represented by T', we continue up the tree. Otherwise, the fragment at $\mathbf{x}$ can be discarded.

Algorithm 4.2 shows a pseudo-code version of the tree traversal for fragment clipping. Appendix A sketches a proof that the algorithm yields the correct result.

```
Input: Point x, Patch C
1  set T_A = T_C, on = true
2  while on ∧ ¬isRoot(T_A) do
3      set T_B = getSibling(T_A)
4      set in = insideTree(x, T_B)
5      if in ≠ unknown then
6          switch T_A.operator do
7              case ∩ : set on = in
8              case ∪ : set on = ¬in
9      set T_A = getFather(T_A)
10 return on
```

Algorithm 4.2: Determining whether a fragment at $\mathbf{x}$ of patch $\mathcal{C}$ should be rendered

The `insideTree` predicate is defined in the next section. Note that the inside/outside classification can yield the values *true*, *false* and *unknown*. The *unknown* classification is used if an inside/outside test is attempted for a surfel s with respect to a patch $P_\mathcal{A}$ which is not represented by any surfels in I_s. This case is discussed below.

Inside/Outside Test for CSG Trees

We now need to specify the `insideTree` predicate used to determine if a point $\mathbf{x}$ is inside an object represented by a CSG tree T. Using the operators stored in T, we can recursively descend into T and thus reduce the problem to finding inside/outside classifications with respect to the leaf nodes of T, each representing a patch. For the patches at leaf nodes, the inside/outside test (4.4) is used. At each node whose children represent objects $\mathcal{A}$ and $\mathcal{B}$, the following operations are applied:

$$[\mathbf{x} \text{ inside } \mathcal{A}\cup\mathcal{B}] \quad \Leftrightarrow \quad [\mathbf{x} \text{ inside } \mathcal{A}] \vee [\mathbf{x} \text{ inside } \mathcal{B}], \tag{4.7}$$

$$[\mathbf{x} \text{ inside } \mathcal{A}\cap\mathcal{B}] \quad \Leftrightarrow \quad [\mathbf{x} \text{ inside } \mathcal{A}] \wedge [\mathbf{x} \text{ inside } \mathcal{B}]. \tag{4.8}$$

A pseudo-code version of the tree traversal is given in Algorithm 4.3. The `insidePatch` predicate implements the inside/outside classification as defined in (4.4) and is described below.

Input: Point **x**, CSG Tree Node T
1 **if** `isLeaf`(T) **then**
2 **return** `insidePatch`($\mathbf{x}, T.patchid$)
3 **else**
4 **switch** $T.operator$ **do**
5 **case** $\cap$: **return** `insideTree`($T.child1$) $\wedge$ `insideTree`($T.child2$)
6 **case** $\cup$: **return** `insideTree`($T.child1$) $\vee$ `insideTree`($T.child2$)

Algorithm 4.3: `insideTree`: Inside/outside test for CSG trees

Inside/Outside Test for Patches

When clipping a surfel s, we can only use clipping partners stored with s. However, during tree traversal, we might need inside/outside classifications for other patches. If not all patches are represented by clipping partners for s, some inside/outside classifications remain unknown.

The extended classification including the *unknown* value is shown in Algorithm 4.4. The `insidePatch` predicate returns a classification of *unknown* if no clipping partners for the patch in question can be found. If there is only one clipping partner from the relevant patch, (4.3) is used to determine whether the point is inside or outside. Only if two or more clipping partners are available, (4.4) is evaluated.

Input: Point **x**, Surfel s, Patch $P_{\mathcal{A}}$
1 **set** $C = I_s \cap P_{\mathcal{A}}$ // find all clipping partners of s that are part of patch $P_{\mathcal{A}}$
2 **switch** $|C|$ **do**
3 **case** 0: **return** *unknown*
4 **case** 1: **return** $\neg$`clips`($C_0, \mathbf{x}$) // Eq. 4.3
5 **otherwise**
6 `sortForDistance`($C, \mathbf{x}$)
7 **if** `concave`(C_0, C_1) // Eq. 4.4, see Section 4.1.1
8 **then return** $\neg$`clips`($C_0, \mathbf{x}$) $\vee$ $\neg$`clips`($C_1, \mathbf{x}$)
9 **else return** $\neg$`clips`($C_0, \mathbf{x}$) $\wedge$ $\neg$`clips`($C_1, \mathbf{x}$)

Algorithm 4.4: `insidePatch`: Inside/outside test for a patch

An *unknown* classification does never lead to discarding a fragment (see Algorithm 4.2). Consider a fragment at position **x** of a surfel s. If the inside/outside classification for **x** with respect to a patch $P_{\mathcal{A}}$ cannot be determined, no surfels of

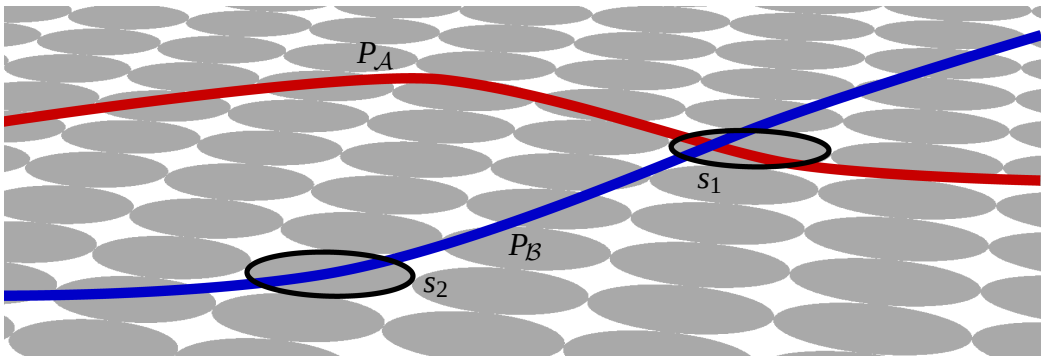


Figure 4.5: Three intersecting patches. The patch $P_\mathcal{C}$ (gray) intersects $P_\mathcal{A}$ (red) and $P_\mathcal{B}$ (blue). For $s_1 \in P_\mathcal{C}$, the inside/outside classification with respect to both $P_\mathcal{A}$ and $P_\mathcal{B}$ can be computed. For $s_2 \in P_\mathcal{C}$, the classification with respect to $P_\mathcal{A}$ yields *unknown*. Thus, the decision which fragments to discard only depends on $P_\mathcal{B}$.

P are clipping partners of s, i. e. $P_\mathcal{A}$ does not intersect s. Thus, s is either completely inside or completely outside $P_\mathcal{A}$. If any of the fragments of s needed to be discarded due to $P_\mathcal{A}$, s would have been discarded entirely by the CSG operation.

If the inside/outside classification for a complex object is computed and the classification for one of the involved patches is unknown, we need to deal with *unknown* as a value in Algorithm 4.3. In order to evaluate (4.7) and (4.8), the $\vee$ and $\wedge$ operators are extended to handle the *unknown* value:

$$b \vee \mathit{unknown} = b, \tag{4.9}$$
$$b \wedge \mathit{unknown} = b. \tag{4.10}$$

for all $b \in \{true, false, unknown\}$.

The reason for using these propagation rules is as follows: If the inside/outside classification for patch $P_\mathcal{A}$ yields *unknown*, the clipped surfel s is not affected by $P_\mathcal{A}$. Thus, if $P_\mathcal{A}$ and $P_\mathcal{B}$ form a complex object, we can use the classification returned by $P_\mathcal{B}$ for the combination of $P_\mathcal{A}$ and $P_\mathcal{B}$. Only if the inside/outside status for both $P_\mathcal{A}$ and $P_\mathcal{B}$ is *unknown*, the classification for their combination is *unknown* as well.

Figure 4.5 shows an example. Let an object be defined by the CSG tree $(\mathcal{A} \cap \mathcal{B}) \cap \mathcal{C}$. We consider two surfels $s_1, s_2 \in P_\mathcal{C}$. In order to determine which fragments $\mathbf{x}$ to discard, we need inside/outside classifications with respect to the object $\mathcal{A} \cap \mathcal{B}$. According to the CSG tree, $[\mathbf{x}\ \mathit{inside}\ \mathcal{A} \cap \mathcal{B}] \Leftrightarrow [\mathbf{x}\ \mathit{inside}\ \mathcal{A}] \wedge [\mathbf{x}\ \mathit{inside}\ \mathcal{B}]$. Because s_1 intersects the surfaces of both $\mathcal{A}$ and $\mathcal{B}$, this expression can be evaluated for fragments of s_1. s_2 does not intersect the surface of $\mathcal{B}$, and the classification cannot be computed. The value of $[\mathbf{x}\ \mathit{inside}\ \mathcal{B}]$ is *unknown*. In this case, the classification becomes independent of $\mathcal{B}$: $[\mathbf{x}\ \mathit{inside}\ \mathcal{A} \cap \mathcal{B}] \Leftrightarrow [\mathbf{x}\ \mathit{inside}\ \mathcal{A}] \wedge \mathit{unknown} \Leftrightarrow [\mathbf{x}\ \mathit{inside}\ \mathcal{A}]$.

4.2.3 Combined Hardware/Software Renderer

One big advantage of surface splatting is that hardware acceleration is possible [37, 233]. This partly holds for the algorithm presented here. All surface surfels can be rendered using an arbitrary renderer, possibly hardware accelerated. The edge surfel rendering is done in software.

In a final step, the images obtained by rendering edge surfels and rendering surface surfels are combined using the blending weights accumulated during splatting. Hence, the blending step has to be executed before the normalization. The easiest way to achieve this is to pass the software rendered image to the hardware renderer, and modify the regular normalization pass such that it adds the weighted contribution from the software renderer before normalization.

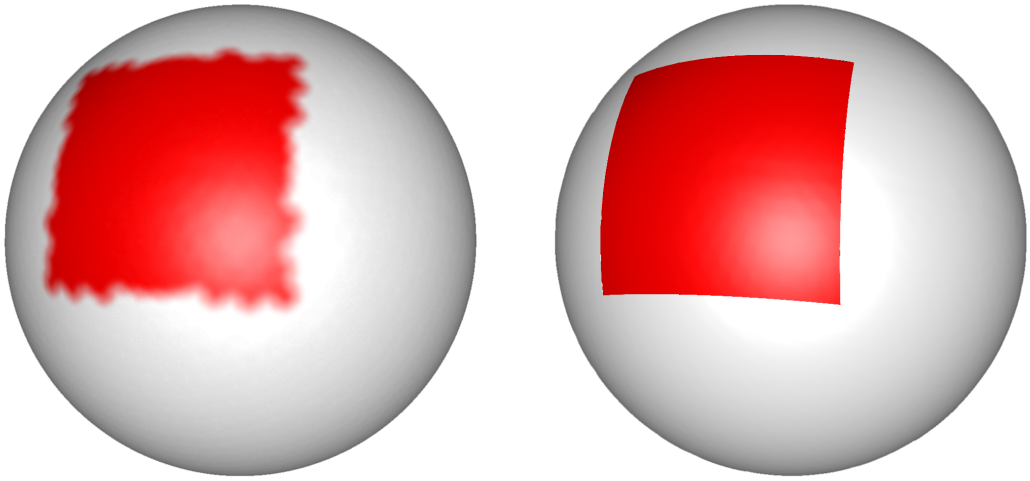
For splatting, the algorithm described in [233] is used. The algorithm involves computing the matrix inverse of the splat space to screen space mapping $\mathbf{M}$. When this mapping is close to singular, i. e. when the splat normal is almost perpendicular to the viewing direction, numerical instabilities make the matrix inverse unreliable. In [233], such splats are discarded. Since they are almost perpendicular to the viewing plane, the resulting artifacts are hardly noticeable.

However, we use splatting to determine clipping partners. Missing a clipping plane can result in serious artifacts, independent of the direction of the clipping plane. It is hence imperative that all potential clipping partners are considered. Therefore, we can not simply discard surfels with almost singular $\mathbf{M}$. Instead, for surfels whose normal is almost perpendicular to the viewing direction, a thick line is rasterized into the clip buffer, covering the ellipse. This might result in clipping partners being added to surfels that do not overlap in the clip buffer. As only the closest surfels are taken into account when computing inside/outside classifications, adding too many clipping partners does not cause any problems. Surfels that are clipping partners although they do not overlap in the clip buffer are never used for an inside/outside decision.

4.3 Modeling

There are several ways in which models with sharp features can be created. The most obvious is CSG modeling. It is sufficient to classify the surfels as described in Section 4.1.1 and save the CSG tree built during modeling. The actual clipping is then performed during rendering.

CSG modeling typically uses more operators than only union and intersection that are described above. However, all other operations can be easily assembled from union and intersection. When a CSG tree originally contains other operators than union and intersection, it is rebuilt as follows: First, other operations are rewritten as combinations of union, intersection and inversion. Then, all inversion operations are propagated down the CSG tree until the tree only contains inversion operations at its leaf nodes. At leaf nodes, inversion is performed by inverting the



(a) (b)

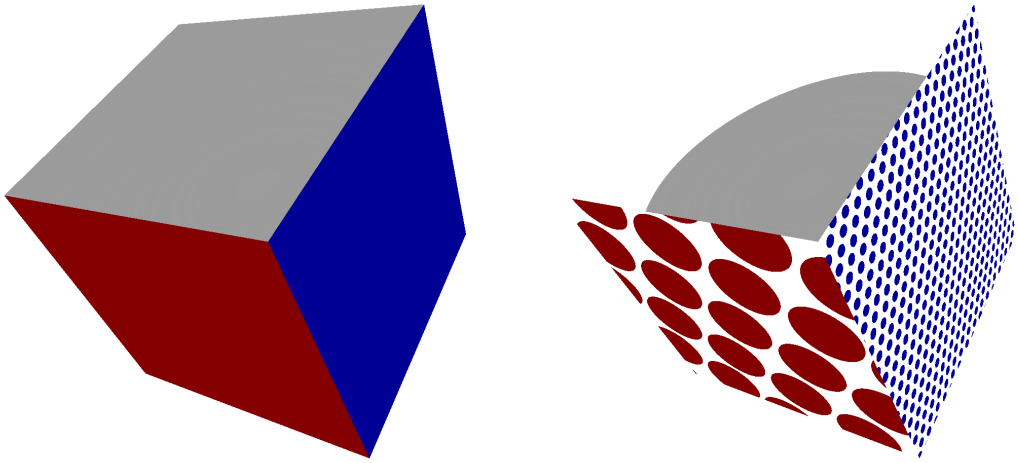
Figure 4.6: Introducing texture discontinuities. (a) A square is painted onto a sphere by coloring the surfels inside the square. Splatting blurs the edges of the square. (b) Texture discontinuity surfels enforce sharp features in the texture, while allowing smooth normal interpolation.

result of the inside/outside test (i. e. the result of `insidePatch`). Consider for example the CSG tree $\mathcal{A} \setminus (\mathcal{B} \cup \mathcal{C})$. Rewriting the $\setminus$ operator yields $\mathcal{A} \cap \overline{(\mathcal{B} \cup \mathcal{C})}$. The inversion is then propagated to the leaf nodes: $\mathcal{A} \cap (\overline{\mathcal{B}} \cap \overline{\mathcal{C}})$, which is an expression that can be evaluated using the clipping algorithm.

Sharp features can also be identified by feature detection algorithms, e. g. [160]. Region growing algorithms can then be used to identify patches.

Animations of point-sampled objects involving fracturing can also create sharp edges. An example of such animations is presented in Chapter 7. In these cases, the sharp edge is at a boundary of the surface. Contrary to the situation after CSG operations, there is no visible surface that the geometry is clipped against. The discontinuity is therefore enforced by special *discontinuity surfels*. These surfels are invisible, and are only used to clip surfels to create a clean edge.

The technique presented above can not only be used to represent geometric discontinuities, but also discontinuities in texture. To achieve this, texture discontinuity surfels are added at texture discontinuities. During splatting, the normal and depth information is splatted as usual, and only the color values are clipped. Thus, the surface is smooth across the intersection with the texture discontinuity, while a discontinuity is introduced into the texture. Figure 4.6 shows an example. A square is texture-mapped onto a sphere by simply coloring the surfels. Splatting the surfels blurs out the edges. By introducing texture discontinuity surfels, the shading is smooth across the edges of the square, while the color information is



(a) (b)

Figure 4.7: (a) A cube with differently sampled faces. (b) Sampling of the cube: The faces are represented with 1, 25, and 900 surfels, respectively.

discontinuous. Texture discontinuities can be identified using texture analysis, or they are explicitly specified during modeling.

4.4 Results

To date, the algorithm described in this chapter is the only rendering algorithm for point-sampled objects supporting arbitrary CSG combinations of smooth primitives. It enhances point-sampled surfaces with the possibility of representing sharp features, making no assumptions on sampling, or the shape and size of the features.

Figure 4.7 illustrates the robustness of the algorithm against varying sampling densities. Since the algorithm allows for an arbitrary number of clipping partners, the edges and corners of the cube can be rendered without problems, even though the sampling density varies abruptly across the edges. The faces are sampled with 1, 25, and 900 samples, respectively. An example of curved edges formed by patches with different sampling densities and patterns is shown in Figure 4.8.

In Figure 4.9, an extremely thin geometric structure illustrates the importance of the possibility to clip surfels from several sides. Figure 4.10 shows another example: An icosahedron is created as the intersection of 20 half-spaces, each face is represented with a single surfel.

The two-pass renderer performing edge clipping is significantly slower than a hardware splatting algorithm, such as [37] or [233]. However, the former can

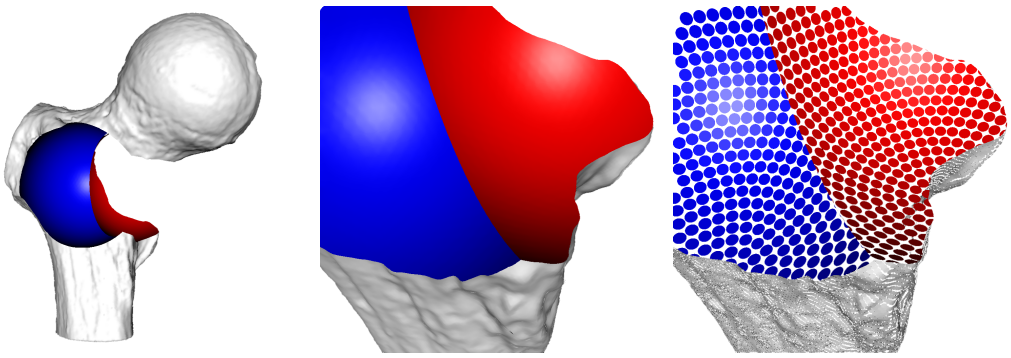
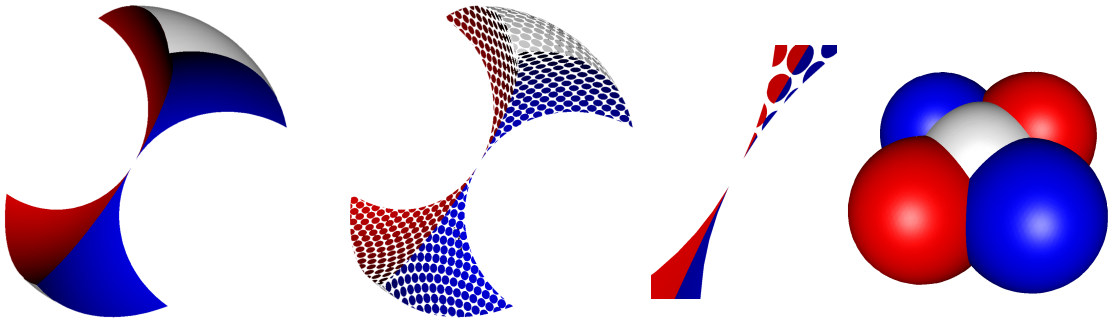


Figure 4.8: (a) A femoral neck minus two spheres. (b) Close-up of a 3-surface intersection. The edges can be magnified indefinitely without blurring. (c) Sampling. Note the different sampling densities.



(a) (b) (c) (d)

Figure 4.9: (a) An extremely thin structure created by CSG difference operations. (b) Sampling of the spheres. (c) Close-up of the spikes. Top: normal view, bottom: Sampling. (d) Spatial setup creating the structure.

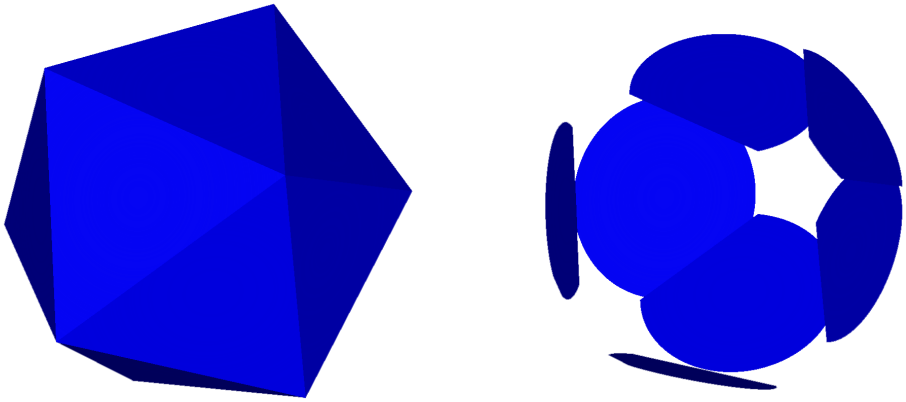


Figure 4.10: (a) An icosahedron created by the intersection of 20 half-spaces. (b) Sampling. Each face is represented with one surfel.

not perform any edge clipping while the latter provides only limited support for corners.

Using the combined hardware/software renderer, the method presented here is faster than the software renderer provided with Pointshop3D [161], as long as edges only fill a small fraction of the screen. Rendering times are similar when approximately one fourth of all pixels show edges. In practice, only a small part of the screen is covered by edge surfels.

Because the costly edge rendering algorithm is only used for surfels close to the edge, the rendering time mainly depends on the number of viewport pixels covered by edge surfels. Usually, edge surfels only amount to a small fraction of all surfels, and only cover a small portion of the screen. Since time complexity for regular splatting algorithms depends on the total number of surfels as well as the total number of pixels covered by the model, a timing comparison is difficult.

Table 4.1 shows rendering performance for the scenes shown in Figures 4.9, 4.10 and 4.8, at different zoom levels. The timings were taken on a Pentium 4, 3 GHz with a GeForce FX5900 graphics board.

The data supports the assumption that rendering time is linearly dependent on the number of visible edge pixels, while the total number of surfels has almost no impact on computation times. The depth of the CSG tree has only minor influence on complexity, as the CSG tree is rarely fully traversed. Spatial coherence can be exploited to further speed up the CSG tree traversal. Past results of the traversal are cached and reused for neighboring fragments.

Since clipping partners can easily be recomputed in every frame, the method is particularly well-suited for scenes including dynamic CSG operations. The pro-

Figure	# Surfels	# Edge surfels	# Edge pixels	FPS
4.9	1871	972	7189	9.09
			33404	2.63
4.10	20	20	8763	8.3
			47469	2.27
4.8	116650	2272	12234	5.55
			69450	2.0

Table 4.1: Rendering performance depending on the number of surfels and visible edge pixels.

posed algorithm can be directly applied after CSG classification, without the need for further processing such as edge resampling.

4.5 Discussion

There are two alternative algorithms for edge rendering after CSG operations on point-sampled objects [2, 161]. In contrast to the approach presented here, both rely on resampling.

Adams and Dutré [2] focus on interactive CSG operations. Sharp edges are created by resampling the surfaces close to the edges. Surfels close to the edge are replaced with smaller surfels in order to minimize blending between the two surface parts. Only the closest surfel is used for inside/outside classification, and there can only be one clipping plane per surfel during the resampling step. After the CSG operation is complete, the resolution of the edge is arbitrarily high, but fixed. An advantage of [2] is that the resampling process does not result in surfels requiring special treatment. After a CSG operation is applied, the resulting object can be rendered using hardware splatting. The resampling ensures sharp edges up to a magnification at which the surface reconstruction visibly blurs edges created by small surfels.

Pauly et al. [161] also resample the edges after a CSG operation. They use the MLS projection operator for inside/outside classification. After the CSG operation, the edges are resampled as follows: Along the edge, pairs of surfels are identified. Each of these pairs is moved to a point on the edge, and the two surfels are fused into a special surfel storing two normals. This surfel is rendered as two half-surfels. The holes in the surface that are created by moving the surfels are closed by resampling the affected areas and inserting surfels where necessary. The edge can be refined arbitrarily, and converges toward the edge that would have been created by the CSG operation applied to the MLS reconstruction of the surfel sets, as defined in [5]. As the edges are created by the special half-surfels, they remain sharp even when zooming in to the edge. When applying more than one CSG operation, the resulting corners cannot be represented this way. Zwicker et

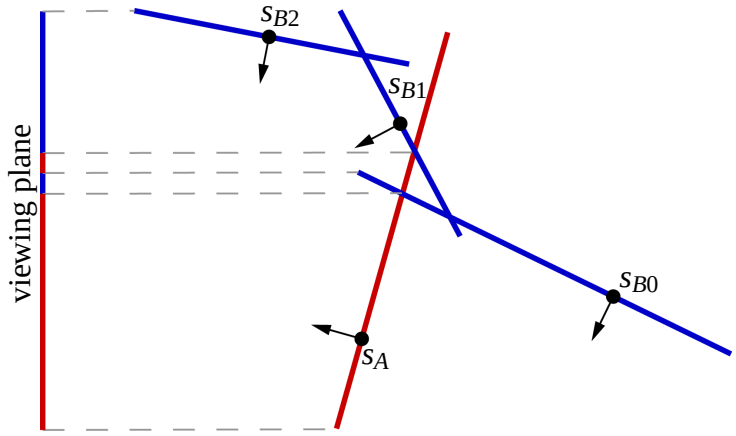


Figure 4.11: Artifact created by the classification method. Shown is a union operation on surfaces $\mathcal{S}_\mathcal{A}$ (red) and $\mathcal{S}_\mathcal{B}$ (blue). Surfel $s_\mathcal{A}$ should be clipped at its intersection with $s_{\mathcal{B}0}$. However, as the center of $s_{\mathcal{B}0}$ is far away, $s_{\mathcal{B}1}$ and $s_{\mathcal{B}2}$ are used for classification of the pixels in question. $s_\mathcal{A}$ "bleeds" into the surface of $\mathcal{B}$.

al. [233] extend the idea of clipped surfels by storing several clipping planes per surfel. Note that arbitrary corners cannot be rendered with neither [161] nor [233]. In [233], corners are generally assumed to be convex: a fragment is discarded if it is clipped by any one of the clipping planes. This is not correct in the general case, see Figure 4.1 for an example of an edge that is neither convex nor fully concave.

CSG operations between objects with highly different sampling densities pose problems for both algorithms presented in [2, 161]. As noted in [161], these problems can be resolved by upsampling the areas close to edges before the CSG operation.

The algorithm presented here does not resample the edges created by CSG operations. Instead, surfels along the edges are interpreted as circular disks, which are then clipped. This way, it is possible to render arbitrary CSG trees applied to point-sampled objects. The sampling density of the participating surfaces can vary arbitrarily, as shown in Figure 4.7. Complex corners or saddle points can be rendered. As a single surfel can be clipped by many others, also very sharp edges and degenerate cases as seen in Figure 4.9 can be rendered without artifacts.

In order to define CSG operations for surfaces represented by surfels, we define the surfaces as a union of disks, which is not necessarily smooth. Figure 4.11 shows a schematic drawing of a situation resulting in an artifact. The artifact is caused by depth discontinuities in the surface and appears only when the sampling density within a single surface patch changes abruptly.

The only way to avoid this problem is to use a smooth surface definition in the whole pipeline. As graphics hardware becomes more flexible, it is possible to accelerate more advanced inside/outside tests based on MLS projection, which eliminates such artifacts. Guennebaud and Gross [95] propose a modified MLS

projection procedure that can be accelerated in hardware and could be used in a CSG framework as presented here.

Like other approaches [2, 161, 233], the algorithm for inside/outside classification faces numerical instabilities when two clipping surfels are almost coplanar. In those cases, the frontface/backface test (4.3) becomes unreliable, which might lead to jagged edges at surface intersections.

Since CSG operations define the final model, the patch primitives have to be orientable manifolds. For non-orientable, or non-manifold surfaces, the notion of inside and outside is not well defined, and CSG operations can not be used in modeling. However, it is sufficient that the surface patches are manifold near the intersection, as only neighboring surface parts are used in classification. Although geometric sharp features pose some difficulties, it is possible to enforce texture discontinuities on non-manifold or non-orientable surfaces without modifications to the rendering algorithm. Since discontinuity surfels are not drawn, inside/outside tests with respect to the surface containing the texture discontinuity are not necessary.

This chapter presented a solution to one of the inherent problems of point-sampled surfaces. The surface representation was enhanced by the possibility to represent arbitrary discontinuities. This is an important feature for shape modeling, especially when modeling artificial shapes like machine parts that have many sharp edges and complex corners. The method can also be applied to texture discontinuities.

The next chapter will present a system harnessing one of the core strengths of point-sampled surfaces: their capability to locally adapt the sampling density on the fly.

Chapter 5

Appearance Modeling using Haptic Interaction

This chapter introduces an interactive system for appearance modeling. Mimicking a painter's workspace, the system allows the user to use brushes and a color palette in order to modify the appearance of a three-dimensional object. The user paints on the surface of the object using a haptic input device, which also gives tactile feedback to the user. In order to represent the changing appearance of the object, the surface is resampled wherever a part of it is modified. Point-sampled surfaces offer advantages when the surface needs to be dynamically resampled, and consequently, all objects in the system are represented using surfels.

The interface is designed to be intuitive to use and hides the underlying representation as completely as possible. By rigidly following a painting metaphor, the goal is that knowing how to paint in reality should be enough to use the virtual painting system. Therefore, a paint model simulates the behavior of real paint when it comes in contact with the object surface, capturing effects such as paint viscosity, diffusion, and drying. Figure 5.1 shows a photograph of the system in use.

5.1 Virtual Painting

Several 2D painting systems have been described in the literature. These mostly focus on realistic paint modeling [24–26, 226]. Elaborate brush models have been developed [53, 222, 225, 227], since they are crucial to Chinese calligraphy. Such systems can can achieve an expressive power similar to painting on real canvases, owing to advanced brush models and bidirectional paint transfer.

Painting on 3D surfaces has proven significantly more challenging. Hanrahan and Haeberli [99] first suggested a 3D painting system, using a mouse to position the brush. Agrawala et al. [9] color the vertices of a scanned object using a spheri-

Figure 5.1: A photograph of the virtual painting system. The haptic device (left) is used to control the brush, another 6D input device (right) moves and rotates the object. Similar to the real painting process, a palette is used to mix paint.

cal brush volume. There is no remeshing and therefore the painted detail is limited by the original resolution. More recent painting systems [87, 106, 115] provide a haptic interface, but still use a sphere-shaped brush to apply paint to the object. In all these systems, color and material information is stored in fixed-sized textures, limiting the level of detail that the artist can apply.

To overcome these limitations, Berman et al. [30] propose the use of multi-resolution images to represent 2D paintings with regions of varying levels of detail. In 3D, the *Chameleon* painting system [104] overcomes the limitations of fixed-sized textures and predefined *uv*-mappings by automatically building a texture map and corresponding parameterization during interactive painting. This adaptive mapping is further improved by Carr and Hart [46]. However, even these elaborate techniques cannot fully solve the parameterization problems inherent in texture mapping. DeBry et al. [60] as well as Benson and Davis [27] solve the parameterization problems by storing paint properties in adaptive octrees, only creating texture detail when necessary.

In order to avoid parameterization problems inherent in texture mapping techniques, the system presented here uses point-sampled surfaces. The object surface as well as the surface of the brush and the palette are represented using sample points. Using resampling operators defined for point-sampled surfaces, the sampling density of these surfaces can be dynamically adapted to the needs of the user. Point-sampled surfaces provide tremendous advantages over textures meshes in this context, as local resampling operations are cheap. Texture atlases are hard to maintain in an environment requiring dynamic resampling, and suffer from discontinuity artifacts across patch boundaries.

5.2 System Overview

From a user's perspective, the user interface presents itself as a 6D input device with haptic feedback that is used to control the brush, and a 6D input device which allows the user to move the object being painted. A regular computer monitor displays an image of the object, as well as a palette that is used to mix colors using the brush.

On the technical side, the system consists of the following components, each of which shall be discussed below:

Object representation: The object is represented as a point cloud. This allows for easy and efficient local resampling. When the user adds detailed textures to the surface, it will be locally upsampled. Similarly, if detail is removed (e. g. by painting over detailed parts), a downsampling algorithm reduces the number of surfels in that region.

Brush model: The brush surface is represented as a point-sampled surface, while the brush dynamics are computed using a simple mass-spring skeleton. The brush surface is deformed using skinning. Collisions between the brush and the object are resolved using only the brush skeleton. Interpenetration of object and brush surface is used to determine affected regions in the paint transfer model.

Paint transfer model: Paint transfer between two point-sampled surfaces is computed based on parameters obtained from the brush simulation, such as penetration depth and pressure, as well as attributes of the paint model. To transfer paint onto the object surface, the object surface is resampled to match the required resolution and ensure that no detail is lost. Since the surface is resampled, it is possible to change not only texture (color) information, but also change the geometry by carving.

Paint Model: Paint is a complex and diverse material. The system can capture the phenomenology of several types of paint. Surface diffusion is computed in the background, creating an effect of watercolor-like mixing. Thick layers of highly viscous oil- or acrylic paint can be applied to the surface leading to changes in the surface geometry.

Renderer: In order to guarantee interactive display rates, a specialized surfel render supports partial updates. The contributions of surfels can be removed from an image, hence surfel colors can be altered without redrawing the complete image. The renderer implements shadow mapping to enhance realism and give the user critical depth cues. Environment mapping is used to render reflective surfaces such as metallic paint.

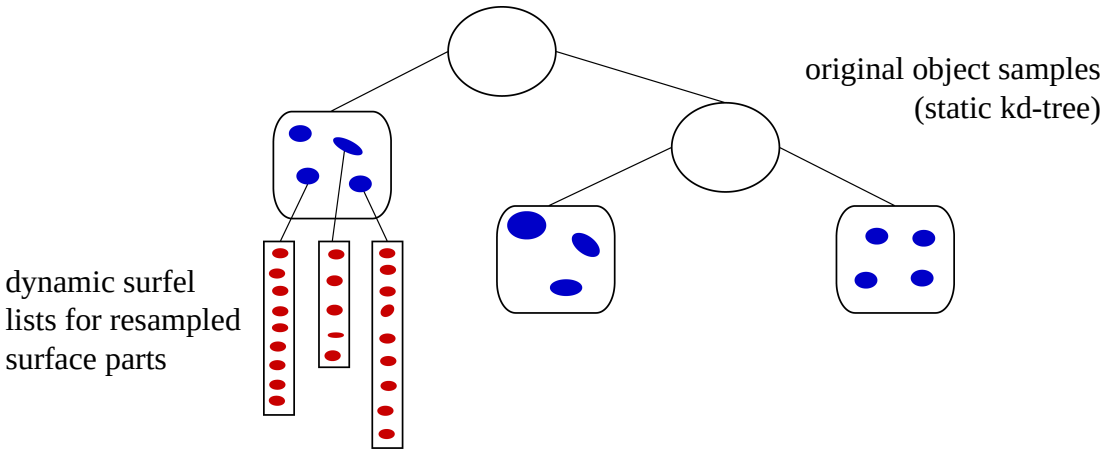


Figure 5.2: The data-structure used to store the object geometry. The original surfel geometry is stored in a static kd-tree. As parts of the surface are resampled, the new surfels are added to lists stored with each surfel.

5.3 Object Representation

Painting systems described in the literature use fixed sized textures [9, 25, 87, 99, 106, 115], or dynamic texture atlases [30, 46, 104] to store the color information. Methods relying on a static texture are severely limited in the resolution they can represent. If dynamic texture atlases are used, artifacts at patch boundaries may become visible.

In the system presented here, the object is represented as a point cloud. Since geometry and texture information is not separated, issues of parameterization do not arise. Pieces of the surface can be up- and downsampled without affecting other surface parts.

Because the painting system needs to be interactive, hard real-time constraints apply for resampling operations, as well as geometric queries such as collision detection for haptic interaction and brush simulation. Therefore, the surfels that represent the object surface are stored in a specialized search data structure, shown in Figure 5.2: The original geometry of the object is stored in a static kd-tree. This geometry is used for collision detection with the brush, and as a lower bound for downsampling operations. When surfels are replaced during resampling operations, the new sampled are added to lists at each leaf node, replacing the original object surfels for rendering and paint transfer. That way, all neighborhood and range queries are still accelerated by the kd-tree up to the original resolution. Static object samples are never deleted, as they are used in collision detection.

A kd-tree is used for the static object samples because the situation calls for an static, space-adaptive search data structure (cf. Section 3.4). Since the object samples are stored statically, the kd-tree will not become unbalanced by insertion or deletion operations and only has to be constructed once. For collision detection

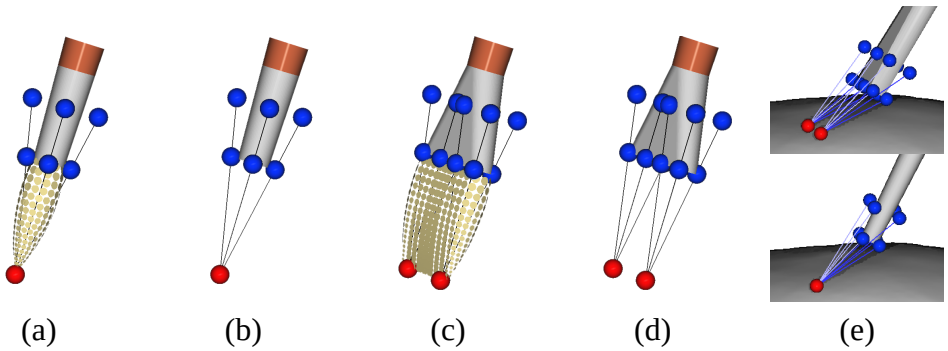


Figure 5.3: (a), (b) A round brush and its skeleton. (c), (d) A flat brush. The active masses shown in red are simulated, while the blue handle points are directly controlled by the haptic device. (e) The brush skeleton deforms in contact with the surface.

as described in Section 5.4.1, only nearest neighbor queries with a fixed number of surfels are used, yielding an average time complexity of $O(\log n)$ per query, where n is the number of surfels in the tree.

Additionally to the regular surfel attributes such as color, normal, and radius, the surfels representing the object surface carry paint-specific attributes. This set of attributes is described in more detail in Section 5.6.

5.4 Brush Model

The brush is modeled as a mass-spring skeleton with a point-sampled surface animated using the deformation computed by the skeleton simulation. Using this simple description, different brush shapes and types can be modeled. Figure 5.3 shows some brushes along with their skeleton. The endpoints of the springs are attached to a rigid brush handle that is controlled by the haptic device. These points will be called *handle points*. The dynamically simulated masses at the tip of the brush are called *active masses*. The springs used in the brush model are simple linear springs exerting a force of

$$\mathbf{F}_s = K_s(l - l_0)(\mathbf{x}_0 - \mathbf{x}_1), \tag{5.1}$$

where K_s is a stiffness constant depending on the brush type, l is the current length of the spring and l_0 is its rest length. $\mathbf{x}_0$ and $\mathbf{x}_1$ are the current positions of the spring's end points. The dynamic behavior of the active masses is computed using Verlet integration [207].

Once the skeleton deformation has been computed, the brush surface is deformed using a combination of linear blend skinning [130] and free-form deformation for point-sampled surfaces [161]. The procedure is described in detail in [1].

5.4.1 Collision Detection

Collision detection is computed for all spring end points. A simple inside/outside test based on the closest surfel is used to determine whether collision has occurred (cf. Section 3.1.4). If there is a collision, a variant of force shading [176] adapted for point-sampled surfaces is used do determine penetration direction and penetration depth. Given the position $\mathbf{x}$ of the mass point that is colliding with the surface, the algorithm finds the n (typically $n = 10$) closest object samples with positions $\mathbf{x}_i$ and normals $\mathbf{N}_i$ and computes a weighted average penetration depth d and local surface normal $\mathbf{N}$:

$$d = \sum_{i=1}^{n} w_i \cdot \mathbf{N}_i * (\mathbf{x}_i - \mathbf{x}), \tag{5.2}$$

$$\mathbf{N} = \sum_{i=1}^{n} w_i \cdot \mathbf{N}_i, \tag{5.3}$$

where the weights w_i are defined using normalized distances of the surface points $d_i = \|\mathbf{x}_i - \mathbf{x}\|$:

$$w_i = \frac{d_{max} - d_i}{\sum_{j=1}^{n} d_{max} - d_j}. \tag{5.4}$$

Here, $d_{\max} = \max(\{d_i\})$. This weighting scheme provides a smooth interpolation of normals over the surface. When a collision for an active mass point is detected, the mass is projected onto the surface of the object:

$$\mathbf{x}' = \mathbf{x} + d\frac{\mathbf{N}}{\|\mathbf{N}\|}. \tag{5.5}$$

If handle points are inside the object, they are not projected outside the object. Instead, a linear force model with a penalty stiffness K_p is used to provide haptic feedback to the user:

$$\mathbf{F}_p = K_p d\mathbf{N}. \tag{5.6}$$

Depending on the velocity $\mathbf{v}_t$ that is tangential to the surface, friction forces are applied:

$$\mathbf{F}_t = -\eta d\mathbf{v}_t, \tag{5.7}$$

where the friction coefficient η depends on the brush used and the object surface.

5.4.2 Haptic Interaction

All forces acting on handle points are passed to the haptic device as feedback to the user. Haptic interfaces require a very high update rate in order to generate a believable impression of a smooth surface. For the haptic device in use, the necessary update frequency is 1 kHz. Therefore, the haptic update is decoupled from all other computations, such as paint transfer or collision detection. In order to provide realistic feedback, the main loop updates the collision detection parameters

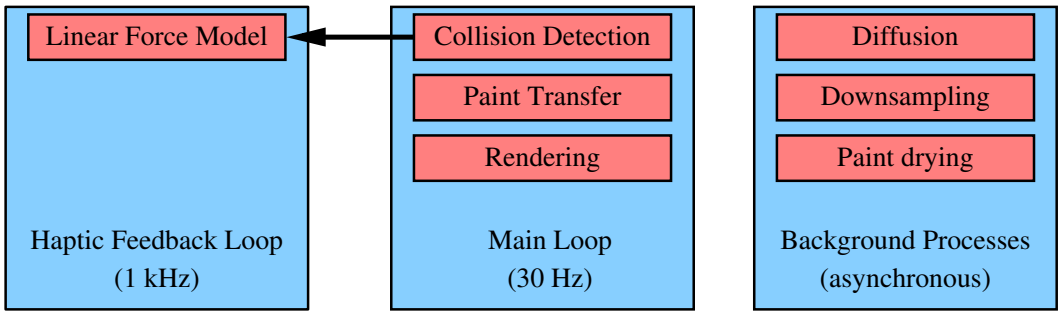


Figure 5.4: Processes used in the haptic painting application. The haptic feedback is decoupled from the rest of the application to allow high update rates. The actual painting as well as rendering is performed in a significantly slower loop. Background processes are used for tasks that are not time-critical, such as diffusion, paint drying and downsampling.

d and $\mathbf{N}$ whenever possible, while the haptic update evaluates the linear penalty force and friction model at 1 kHz. Figure 5.4 shows a schematic drawing of the processes.

In order to avoid discontinuities in the forces displayed to the user, the collision penalty and friction forces are passed through a simple low-pass filter. This filter averages the forces over the last 1/60 second. In theory, this results in slightly viscous behavior. In practice, the inertia of the actual haptic device and the non-rigidity of the object due to the penalty-based collision response hide these effects entirely.

5.4.3 Brush Splitting

If a brush with several active masses moves across regions of high curvature, it might split. Figure 5.5 (a) shows such a case. Such situations can be detected by checking whether the normal directions for each of the colliding active masses differ significantly. A threshold of 60° is used to determine whether the brush is split. If it is, internal surfaces in the brush are activated, and paint transfer is computed using these surfaces. Note that brush splitting is a binary process, the brush is either split or it is not. Due to the visual similarity of the two states, this effect is not noticeable for the user. However, it is important during painting, see Section 5.5.1 for details.

5.5 Paint Transfer

When there is a collision between the brush and the object, paint is transferred. Figure 5.6 shows the steps of the transfer process. First, a local planar approxi-

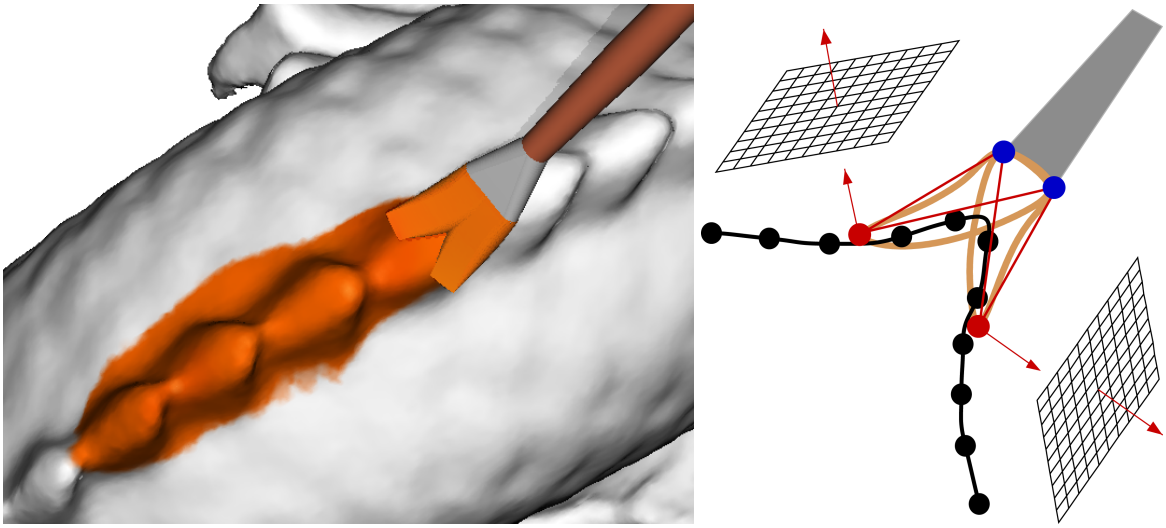


Figure 5.5: (a) When a brush with two active masses moves over a region of high curvature, the brush is split. (b) If the brush splits, paint transfer is computed in several paint buffers to better approximate the surface.

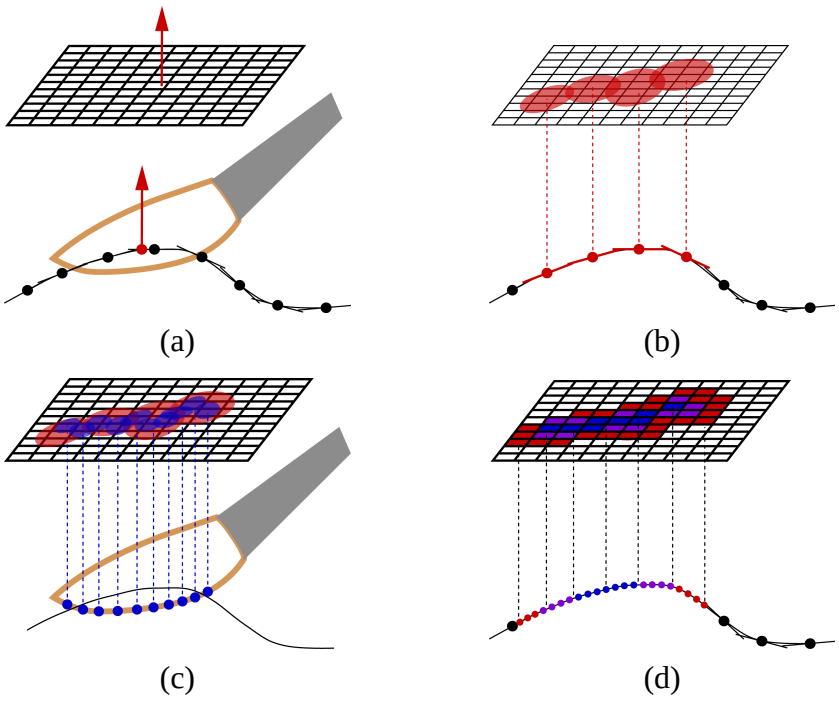


Figure 5.6: Paint transfer computations. (a) The *paint buffer* is constructed as a least squares plane around the brush tip. (b) Surface surfels affected by the paint transfer are splatted into the paint buffer. (c) Penetrating parts of the brush surface are splatted into the paint buffer. (d) After paint transfer is computed in the paint buffer, the object surface is resampled and the resulting colors from the paint buffer are re-projected onto the object and brush surfaces.

mation of the surface around the brush tip is constructed. In this plane, a *paint buffer* is created. The paint buffer is a pixel grid in which the actual paint transfer computations take place. Its resolution is chosen such that each pixel is about the size of a brush surfel. This ensures that no detail is lost during paint transfer. Once the paint buffer has been constructed, object surfels and brush surfels are splatted into the paint buffer. A simple depth test determines the penetrating parts of the brush surface. Only these penetrating surface parts are affected by the paint transfer. Attribute sets for both brush surfels and object surfels are stored in the paint buffer to have all information at hand for paint transfer.

Paint transfer is computed on a per-pixel basis in the paint buffer. A full set of paint attributes as well as penetration depth is available in each pixel, so an arbitrary paint model can be used. The actual paint model used in this system is described in detail in Section 5.6.

Once paint transfer is complete, the part of the object surface penetrated by the brush is resampled to accommodate the information in the paint buffer. One surfel is created for each paint buffer pixel in the resampled region. Again, this process guarantees that information present in the paint buffer is not lost. Although the paint transfer model is inherently bidirectional, the brush surface is not resampled. Instead, the information in the paint buffer is re-projected onto the brush surface and averaged onto the existing brush surfels. Since the sampling density on the brush is high compared to the level of detail one can create with the brush, this is no limitation for the user. Paint transfer has been described in more detail in [1,4].

When new surfels are created, they replace surfels that are part of the object surface. New surfels are assigned to the closest original object surfel. We say they are *child surfels* to their *father surfel*. Surfels that have become unnecessary due to resampling are marked as deleted. If deleted surfels are part of the original surface description, they are retained, but disabled for rendering and paint transfer. Deleted child surfels are removed from the data structure and replaced with new surfels.

5.5.1 Split Brushes

In regions of high curvature, the plane that is fitted to the surface is not a good approximation. This can lead to projection artifacts and distortions if the brush covers larger areas. In such cases, the ability of the brush to split becomes important. A large brush will split when drawn over regions of high curvature. To obtain a better approximation of the surface, a paint buffer is constructed for each of the brush tips, thus approximating the surface with several planes instead of only one. Figure 5.5 (b) shows an illustration.

5.5.2 Downsampling

The paint transfer resamples the object surface to the resolution of the paint buffer. If the brush is of a uniform color, this high resolution is unnecessary. Therefore, a background process permanently checks if parts of the surface can be downsampled. For each original object surfel that is marked as deleted, we compute color variance and normal variance of its children. If these metrics lie below a user-defined threshold, the child surfels are removed and the father surfel is reactivated. Figure 5.7 shows the effect of downsampling. In the interactive application, downsampling occurs in the background, in each downsampling step, only a small part of the surface is processed to guarantee interactivity.

Figure 5.7: The “Fire Dragon” before (left) and after (right) downsampling. The bottom row shows the model with reduced splat size to give an impression of sampling densities. Parts of the surface have been painted, resulting in high sampling density. After downsampling, high sampling density is confined to areas with texture detail. With appropriate variance thresholds, the two versions are visually indistinguishable.

5.6 Paint Model

Mimicking a realistic painting experience necessarily involves creating a realistic model for paint behavior. Several models for interactive painting have been proposed [24–26]. Paint models typically operate on a per-pixel basis. Hence, the model is evaluated in the paint buffer during paint transfer, where information is available in a regular pixel format. However, the adaptive and irregular structure of the underlying surface representation makes modifications and extensions of the paint model necessary once effects other than paint transfer are considered.

The paint model described here is based on the basic paint transfer model by Baxter et al. [25]. Paint is described with a set of attributes determining paint behavior. Most importantly, the visual appearance of the paint is determined by a dry color $\mathbf{C}_d$ and a wet color $\mathbf{C}_w$, as well as an optical density ρ. The optical density determines how transparent the paint is while wet. Dry paint is assumed to be entirely opaque. To determine the visible color of a point given the wet color volume V_w, we evaluate

$$\mathbf{C} = \alpha \cdot \mathbf{C}_w + (1-\alpha) \cdot \mathbf{C}_d, \tag{5.8}$$

where $\alpha = \min(\rho V_w, 1)$ weighs the influence of the wet color on the visible color. While we use simple RGB mixing of paint, more complex models such as the Kubelka-Munk model can be used without incurring major changes [58, 121]. Note that the word color is used to refer to not only an RGB value, but a set of attributes determining paint appearance, such as other rendering parameters like reflectivity and shininess.

When brush and surface interact, paint is exchanged. The paint exchange only affects wet paint. The volume of paint that is mixed depends on the pressure p applied to the surface and a paint-specific transfer rate τ as well as the interaction time Δt. Paint mixing between two surfels i and j can then be described as

$$\Delta V_w^i = (1 - e^{-\tau_{i\to j} p \Delta t}) V_w^i, \tag{5.9}$$

$$\Delta V_w^j = (1 - e^{-\tau_{j\to i} p \Delta t}) V_w^j, \tag{5.10}$$

$$\mathbf{C}_w^{i,\text{new}} = \frac{(V_w^i - \Delta V_w^i)\mathbf{C}_w^i + \Delta V_w^j \mathbf{C}_w^j}{V_w^i - \Delta V_w^i + \Delta V_w^j}. \tag{5.11}$$

$\mathbf{C}_w^{j,\text{new}}$ is computed analogously. If $\tau_{i\to j} \neq \tau_{j\to i}$, paint transfer is not symmetric. This additional degree of freedom can be used to fine-tune paint transfer to make it more usable. Paint flowing back to the brush, while a realistic effect, might not be wanted in some situations. Figure 5.8 shows an example illustrating bidirectional paint transfer.

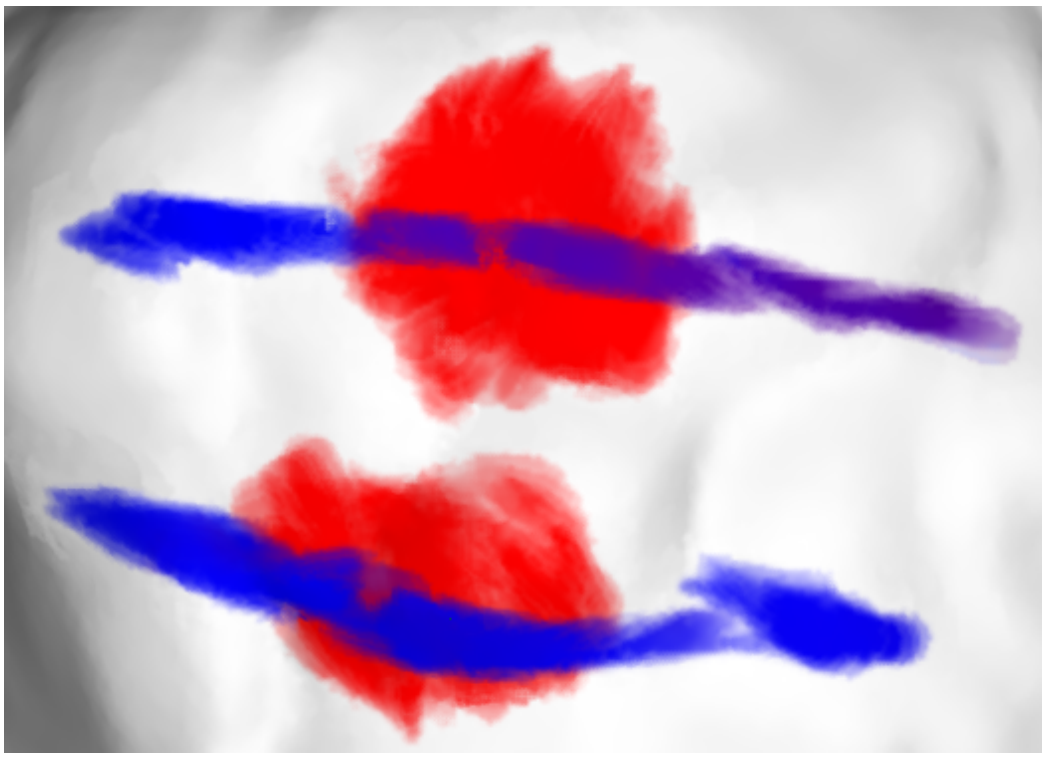


Figure 5.8: A stroke through a wet part of the surface. Top: With bidirectional paint transfer. Bottom: Without transfer from surface to brush ($\tau_{\text{Surface}\rightarrow\text{Brush}} = 0$).

5.6.1 Paint Drying

To simulate the drying of paint, wet paint volume is converted to dry paint with time. This process is performed continuously in the background. For each object surfel, a paint volume of ΔV is transformed from wet to dry paint:

$$\Delta V = (1 - e^{-\kappa \Delta t}) V_w. \tag{5.12}$$

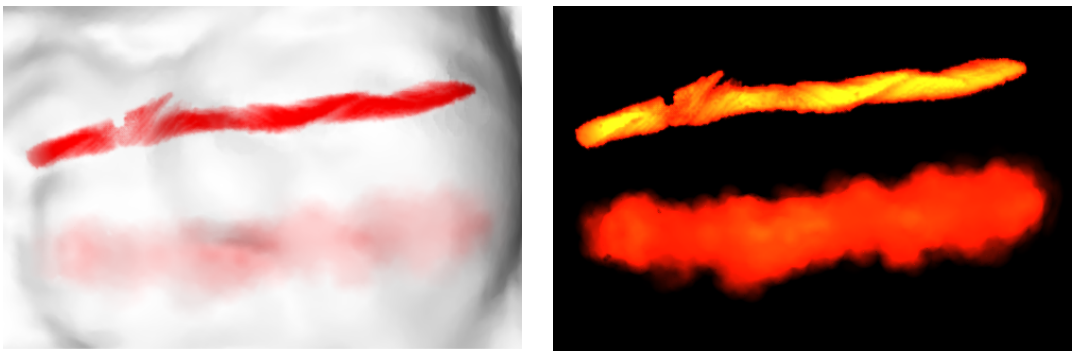
The new dry paint color is then computed similar to (5.11):

$$\mathbf{C}_d^{\text{new}} = \frac{V_d \mathbf{C}_d + \Delta V \mathbf{G}(\mathbf{C}_w)}{V_d + \Delta V}. \tag{5.13}$$

Here, Δt is the time since the surfel was last visited by the drying process, and κ is a parameter defined for each paint type. As wet paint dries, its appearance does not stay the same. This is modeled using the *drying function* $\mathbf{G}(\cdot)$, which reduces reflectivity and shininess, and slightly brightens the color.

Note that since wet and dry paint have different appearance, it is not sufficient to only evaluate paint drying before paint transfer computations, since that would lead to sudden changes in appearance.

Paint on the brush is not subject to drying. While certainly realistic, the effect is generally unwanted, and was found to disturb users.



(a) (b)

Figure 5.9: Surface diffusion. (a) A stroke with and without diffusion. (b) Distribution of wet paint volume after the stroke.

5.6.2 Diffusion

Watercolor or Aquarelle painting heavily relies on transport effects to achieve their typical style. Several researchers have proposed models to simulate the behavior of water-based paint [58, 125, 188, 195, 206]. These range from simple textured strokes to complex physical simulation of the interaction between pigments, water, and paper. In this section, a simple diffusion method is presented that simulates transport of pigments across the surface.

For a function $V(\mathbf{x},t)$, the diffusion equation can be written as

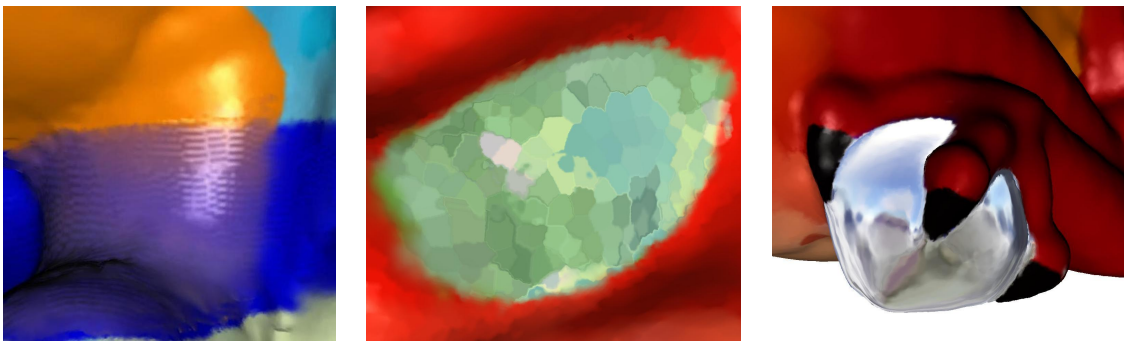
$$\frac{\partial}{\partial t}V(\mathbf{x},t) = -\mathbf{D}\cdot\nabla^2 V(\mathbf{x},t), \tag{5.14}$$

where $\mathbf{D}$ is the diffusion coefficient matrix. For isotropic diffusion, $\mathbf{D}$ reduces to a scalar. A discretization for two surfels i and j at positions $\mathbf{x}_i$ and $\mathbf{x}_j$ yields the paint volume that is exchanged between the two surfels in a given time interval Δt:

$$\Delta V_{ij} = \frac{1}{2}(V_i - V_j)e^{-\frac{1}{D_{ij}\Delta t}}, \tag{5.15}$$

where $D_{ij} = (\mathbf{x}_i - \mathbf{x}_j)^T\mathbf{D}(\mathbf{x}_i - \mathbf{x}_j)$ is the diffusion coefficient between the two surfels. Because of the exponential decay of ΔV_{ij} with distance, we restrict the computations to a small neighborhood around i. The parameter $\mathbf{D}$ depends on the paint type used.

Similar to the paint drying and downsampling computations, surface diffusion is computed asynchronously in the background. The system maintains a list of surfels with non-zero wet paint volume. Whenever a diffusion step is scheduled, a surfel is taken from the list and (5.15) is evaluated for all neighboring surfels. Then, paint volume is exchanged with all of these surfels, and their colors are mixed according to (5.11). Figure 5.9 shows the effect of surface diffusion.



(a) (b) (c)

Figure 5.10: Surface effects. (a) Hair imprints in viscous paint (detail of Figure 5.12). (b) Beaten gold or mosaic pattern. Small patches are restricted to a single normal direction. The detail in (a) and (b) is purely geometric. (c) Metallic paint. Environment mapping is used to give the impression of a reflecting surface. (b) and (c) are details of the “Fire Dragon” model shown in Figure 5.7.

5.6.3 Geometric Detail

Very viscous paint such as oil or acrylic paint is not absorbed by the canvas material, but instead forms layers of finite thickness. The brush or tool used to apply the paint leaves imprints in the paint. In order to simulate these effects, the geometry of the surface has to be changed. Since the surface is resampled during painting, we have complete control over both appearance and geometry of the new surface samples. Depending on the brush type used, we carve a geometric pattern into the newly generated surface, thus adding layers of paint, or molding the applied paint with brush imprints. The surface is shaped using an offset function whenever the brush velocity is aligned with the direction of the brush skeleton. Thus, the illusion of brush hair imprints is generated. Figure 5.10 shows surface patterns that were created by small-scale geometric modifications.

Note that it is not possible to use this technique for modeling larger features. While newly created surface surfels accurately represent the new surface geometry, collision detection relies on the static search data structure based on the original geometry. If the current surface deviates too far from the original surface, this might lead to incorrect collision information.

5.7 Rendering

Except for the brush handle and the palette, which are represented using triangle meshes, all surfaces in the painting system are point-sampled and represented with surfels. While hardware accelerated point rendering has made interactive rendering possible, frame rates of well over 20 frames per second are necessary in order

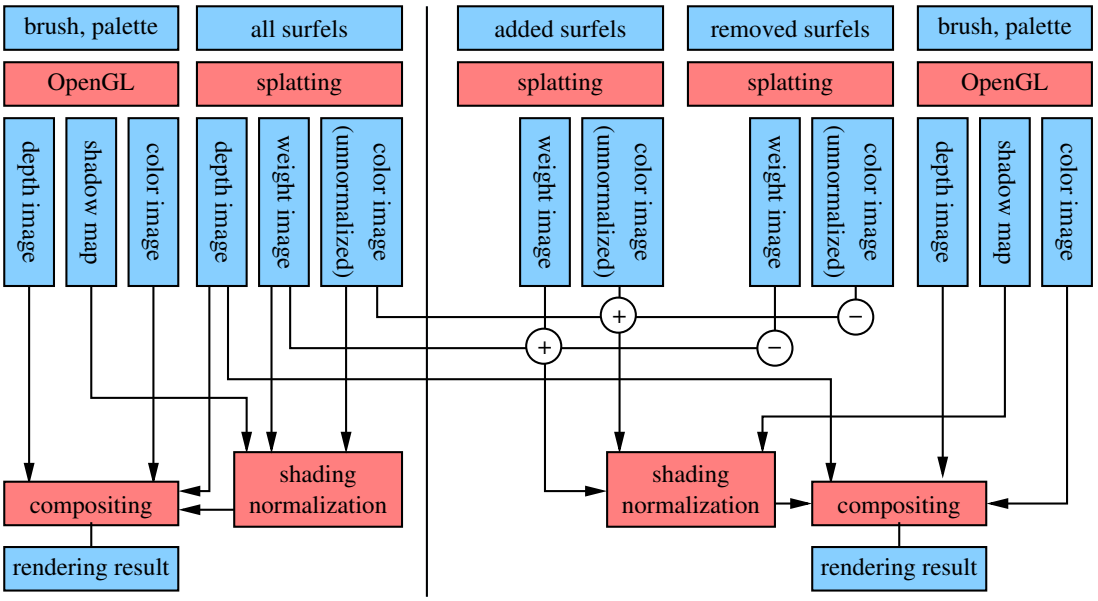


Figure 5.11: The rendering pipeline. Left: Full update. Right: Differential update.

to deliver a believable virtual painting experience. Painted models quickly approach half a million surfels, rendering all of these in each frame is not possible in an interactive setting.

Therefore, only parts of the surface that have changed are redrawn. This includes the brush tip and, if a paint was transferred since the last rendered frame, the parts of the surface that have changed. The renderer supports *un-splatting* of surfels. Thus, changed parts of the surface can be updated without redrawing the complete image. The renderer also supports environment mapping to simulate the visual appearance of reflective paint. Figure 5.11 shows the rendering pipeline.

The point renderer creates a color image, the corresponding accumulated weights, and a depth image. Experience has shown that shadows provide crucial depth cues in the absence of 3D vision. Hence, a shadow map is created from the brush to indicate its position relative to the object. Reflectivity is evaluated per surfel using a cube map, the resulting colors are blended normally. More recently, deferred shading for point-sampled objects has been presented [38]. This technique would make it possible to evaluate reflections on a per-pixel basis, leading to more accurate reflection lines. However, it is only available on graphics hardware supporting floating point framebuffers.

In standard surfel splatting, the accumulated per pixel weights are used to normalize the color image and then discarded. To permit differential updates, the unnormalized color image as well as the weights are retained. In case of a differential update, an image of all deleted surfels and one of all added surfels is created, as well as a weight image for each of these. These are then combined with the source image in the final normalization pass, which additionally draws palette and brush handle. The partial update does not change the depth buffer. For the small-scale

carving that is used to simulate layers of paint, this is not a problem, however, this technique is not well-suited for carving operations modifying the geometry on larger scales.

Lists of surfels to be subtracted and added to the image are kept and updated whenever painting events invalidate parts of the surface. These lists are used to compute partial updates whenever a frame is rendered. Full updates are computed when the object was moved (requiring all visible surfels to be updated), or when the number of surfels in the partial update lists exceeds a threshold, making partial updates unprofitable. Due to numerical inaccuracies in the 8 bit framebuffer, a full update should be performed if certain pixels are updated more than a certain number of times. A per surfel update counter is used to trigger a full update if any region is painted repeatedly.

5.8 Results

The painting system as described above was used to create the painted bunnies shown in Figure 5.12. The models have 300000 to 750000 surfels. The main loop runs at approximately 30 Hz, i. e. 30 paint events are evaluated each second (measured on a Pentium 4 at 3 GHz). A differential rendering update is scheduled after each paint event, such that painted strokes become visible immediately. Due to the possibility of differential updates, the frame rate is almost independent of the number of object samples, while the time for a full update depends linearly on the number of surfels. The only part of the computations during painting that is influenced by the total number of surfels is the collision detection, which takes $O(\log n)$ time for n sample points. However, compared to rendering times, the cost for collision detection and brush simulation is negligible (below 100 μs). This guarantees stable frame rates — the system remains usable even for large input models. Table 5.1 gives rendering times for full updates. For comparison, a scatter plot shows differential update times depending on the number of updated surfels. Typical differential updates affect up to 5000 surfels, and are completed in 15 ms.

Figure	# Surfels	Rendering time
not shown	96518	123 ms
5.12		
middle	347424	497 ms
top-left	398484	610 ms
5.7	572781	844 ms

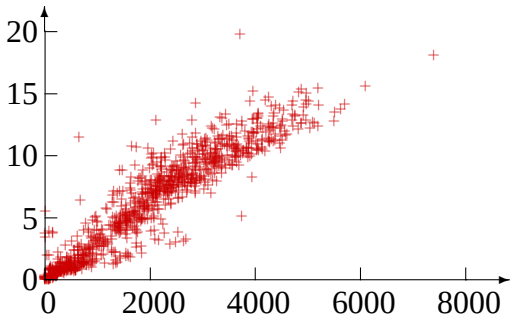


Table 5.1: Rendering performance. The table lists average rendering times (full update) for several models. The graph on the right shows timing for differential updates as a function of the number of updated surfels.

5.9 Discussion

The results show the advantages of using a point-based surface representation. By abandoning a more rigidly structured representation such as textured triangle meshes, we gain the flexibility needed to create a believable virtual painting experience. Several aspects are noteworthy:

- Surfels represent both texture and geometry. As samples of both surface position and color (as well as, in our case, paint attributes), they allow for local modification of both aspects of the object surface in a unified fashion. Since the object surface is resampled during painting, we are free to arbitrarily modify the new surface, not only changing its color, but adding geometric detail as described in Section 5.6.3.

- It is easily possible to locally resample the surface, adding detail where necessary. In mesh-based representations, storing and dynamically and adaptively changing surface detail causes major problems. Algorithms for the maintenance of dynamic texture atlases have been developed that enable adaptive storage of appearance information [46, 104]. However, problems like discontinuities at triangle boundaries and distortion artifacts due to projection errors remain. The system presented here, based entirely on point-sampled surfaces, does not suffer from these problems.

- The surface is resampled when it is changed, giving the system full control over the surface geometry after each paint event. This is used to implement small-scale geometric effects in order to realistically model viscous paint. While in principle, this would be possible to implement for mesh-based editors, it would cause additional complications for the maintenance of adaptive textures. Baxter et al. have implemented geometric manipulations for their painting system, which paints into a fixed-size texture on a plane using displacement textures [26].

When painting, geometric detail can be added to the surface (see Section 5.6.3). These changes to the object geometry are restricted to small surface details. True carving that would significantly change the object shape is not possible. The reason for this is twofold: Due to performance considerations, the kd-tree used for collision detection is entirely static. This is a design choice deemed necessary as stable and interactive frame rates are critical to convey realism. Unfortunately, this means that collision detection cannot reflect the changes to the geometry made during painting. Changing the geometry during painting would lead to divergence of haptic and visual experience, which is obviously unacceptable. The second problem is related to rendering. Significant changes in geometry would require an update of the depth map. This, however, is not easily possible during partial updates. A complete re-rendering would be necessary, which would severely degrade interactivity.

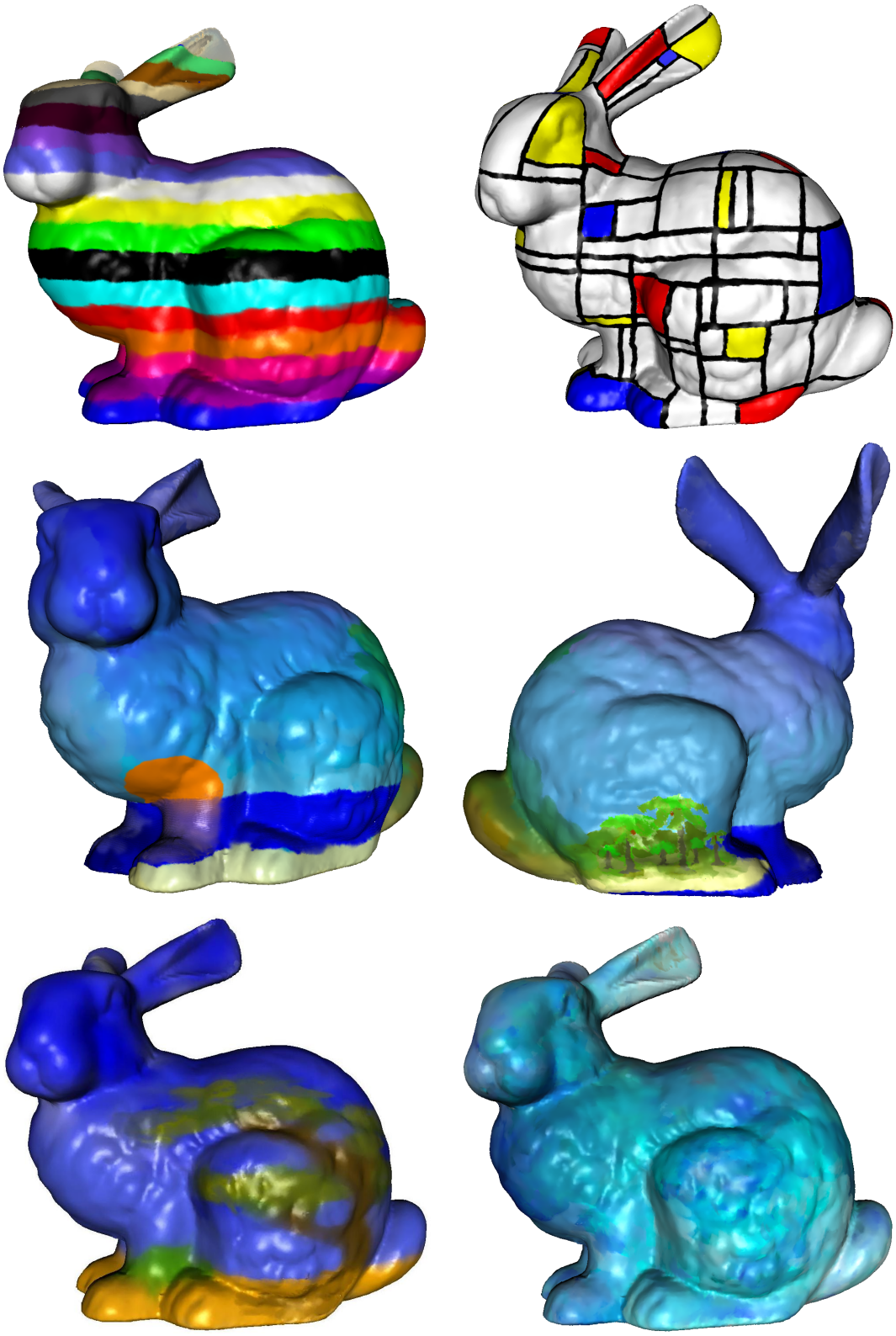


Figure 5.12: Art on Bunnies. Several designs produced using virtual painting. The bunnies shown here took three to five hours to create.

The choice of data representation is a critical design decision for an interactive system as presented here. Due to their flexibility, point-sampled are a near optimal choice for virtual painting. The most noticeable drawback is that rendering is slow compared to mesh-based geometry. On more recent hardware, this is less and less problematic as splat throughput reaches levels where models with surfel counts of half a million splats can be rendered at more than 30 Hz [38], removing the need for workarounds like differential updating.

Using the painting system described here, it is possible to create visually appealing painted models. The user interface is designed to be intuitive and hides the underlying representation. This makes the system useful for non-expert artists. Once the artwork is completed, it is represented as a point cloud. In order to use it in mesh-based systems for further processing, modeling, or animation, it needs to be converted to a triangle mesh. The next chapter presents a method for conversion of point-sampled objects to textured triangle meshes.

Chapter 6

Conversion

While point-sampled surfaces are a good choice for many applications, such as the one described in the last chapter, most algorithms and tools for geometry processing, editing, and rendering work on triangle meshes. To ensure interoperability, conversion between point-sampled surfaces and triangle meshes has to be implemented.

Early work on point-sampled surfaces includes techniques to convert traditional mesh-based or implicit representations to a point-sampled surface [165]. *LDC sampling* [131] is one such technique. Texture and geometry are sampled by orthogonal views, yielding a sampling with guaranteed maximum surfel distance.

Conversion from a point cloud to a triangle mesh is much harder. Algorithms for surface reconstruction from point sets have been developed in computational geometry, for instance [15, 16, 33, 35, 45, 65, 101]. However, these algorithms are only concerned with geometry, not texture. Directly triangulating a point cloud will result in a impracticably large triangle mesh. In triangle meshes, geometry and texture are not sampled in a unified manner, but appearance information and small-scale detail is separated from the geometry and stored in textures, yielding a significantly smaller base mesh and detailed texture information in a compact form.

In this chapter, an algorithm separating geometry and texture information is presented that solves the problem of converting a point-sampled object to a mesh of reasonable size and a texture atlas storing surface detail and appearance information.

The algorithm first generates a mesh by triangulation and reduces its complexity using established simplification techniques. Once a mesh of appropriate size has been computed, texture patches are computed for each of the mesh triangles. The patch size is chosen adaptively to capture detail where present, while conserving space wherever low resolution is sufficient. Finally, the texture patches are packed into a texture atlas. The output mesh can then be used by tools for mesh processing and editing.

6.1 Mesh Generation

In a first step, a mesh is computed that approximates the geometry represented by the input point cloud. Such a mesh can be computed by triangulating the input point cloud using Delaunay triangulation, for example using the Cocone and Tight Cocone algorithms [15, 65, 66]. Tight Cocone always generates a watertight triangulation, while the Cocone algorithm can be used to triangulate surfaces with boundaries. The triangulations produced by these algorithms are generally of good quality. In the presence of noise, a less susceptible triangulation method, for example Robust Cocone [67], is used. The result of a triangulation contains almost every sample point as a mesh vertex. While containing all information present in the point cloud, the resulting mesh is unwieldy and unnecessarily complex in regions of low geometric detail.

Since the goal is to separate textures and geometry, the mesh is simplified in a second step. This ensures that the mesh resolution is adequate to represent the object geometry, and not influenced by the object texture.

Garland and Heckbert's simplification method [80] is steered by a single quality parameter and thus well suited for the task. The algorithm contracts pairs of vertices if the operation only changes the surface geometry within the user-specified threshold. Since in our case, it is important that the topology of the mesh is not changed, only pair contractions along edges of the mesh are allowed. The result of the simplification is a mesh that approximates the original surfel geometry, while having an appropriate triangle count to capture the input geometry.

Without changing the subsequent texture generation algorithm, the triangulation and simplification steps can be replaced with a different surface reconstruction method that is insensitive to texture information present in the set of samples, for instance using an implicit function as intermediate representation [101].

6.2 Texture Generation

The main part of the conversion algorithm deals with the generation of textures for the simplified mesh. The problem can be broken down to individual triangles: A texture patch is created for each triangle in the mesh. Since many small textures are not practical, texture patches are packed into one or more texture atlases that can be used in rendering or further processing.

The patch rendering algorithm uses iterative refinement to adapt the patch size to the detail present in the original model. An error metric controls the texture size.

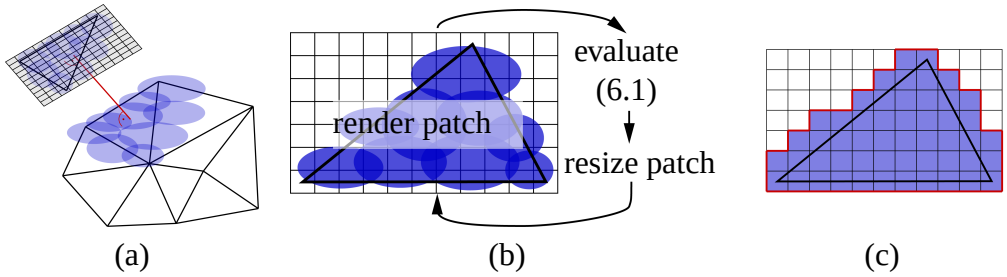


Figure 6.1: Rendering texture patches: (a) Patch rendering setup for a triangle. The viewport is parallel to the triangle, the triangle's longest edge is aligned with the x-axis of the framebuffer. (b) Iterative refinement. (c) Patch extraction. Blue pixels are used in the texture atlas, the outline of the texture patch is highlighted in red.

6.2.1 Patch Rendering

Computing the texture patches for mesh triangles is essentially an image-based technique. EWA splatting [232] is used to render the texture patches.

For each triangle t, the viewing transformation for patch rendering is set up such that the triangle lies in the image plane, with its longest side parallel to the screen-space x-axis. The surfels are projected onto the triangle plane using an orthogonal projection. Hidden surface removal is performed using visibility splatting [165]. Only few surfels will contribute to each texture patch, and an acceleration structure can be used to significantly speed up patch rendering. During splatting, visible surfels whose projected splat ellipses intersect with the triangle are added to a set P_t. Figure 6.1 illustrates the rendering setup.

The viewport size determines the resulting patch size. Starting from the smallest possible patch size, the viewport size is enlarged iteratively until an error function E drops below a user-defined threshold. E measures the difference between the color function reconstructed from the irregular point samples, $\mathbf{C}_{P_t}(x,y)$, and the piecewise linear function represented by the texture, $\mathbf{T}(x,y)$.

$$\begin{aligned} E &= \int_{(x,y)\in t} e\left(\mathbf{C}_{P_t}(x,y), \mathbf{T}(x,y)\right) \mathrm{d}x\mathrm{d}y \\ &\approx \sum_{s\in P_t} A_s \cdot e\left(\mathbf{C}_s, \mathbf{T}(x_s,y_s)\right) \end{aligned} \tag{6.1}$$

Here, (x_s,y_s) is the projected position, and $\mathbf{C}_s$ is the color of the surfel s. The integral is approximated numerically by summing up the errors at the projected surfel centers, weighted with the area of the surfel, A_s. The local error $e(\cdot,\cdot)$ is a metric for colors. We use the metric induced by the L^2 norm in RGB space. The same procedure can be applied to bound the error for normal and depth textures. Note that in general, $\mathbf{C}_{P_t}(x_s,y_s) \neq \mathbf{C}_s$. The reason for this is that the reconstruction of the color function using EWA splatting is not interpolating. Therefore, there can be pathological cases in which the error never drops below the user threshold, independent of the resolution.

The iteration is terminated once the texture error is below a threshold, or when a maximum resolution is reached. This maximum sensible resolution can be derived from the surfel arrangement in P_t: Given the minimum distance $d_{\min}$ between two surfels in P_t, we can safely resample the information contained in P_t onto a regular grid with a pixel spacing of no less than $s = \frac{1}{2\sqrt{2}} d_{\min}$, independent of the grid resolution.

We can thus exit the refinement loop once the pixel spacing falls below s. This defines an upper bound on the texture resolution. In most cases, however, the refinement will exit early as the error (6.1) drops below the user-defined threshold.

Once the patch size has been determined, the triangular texture patch is extracted from the framebuffer by rasterizing the triangle. To make sure that no interpolation errors appear along the triangle edges, the triangle is rasterized with a one pixel boundary to all sides. The resulting texture patch is stored and later packed into a texture atlas.

Normal and Depth Textures

EWA Splatting can be used to interpolate any surfel attribute, such as normals and depths. This data can be used for normal mapping or displacement mapping respectively. The procedure, including the iterative patch size computation, remains largely unchanged, however, a suitable local error function $e(\cdot,\cdot)$ needs to be found for other attributes. For normals,

$$e_{\mathbf{N}}(\mathbf{N}_s, \mathbf{N}_t) = 1 - (\mathbf{N}_s * \mathbf{N}_t) \tag{6.2}$$

is a good choice. Note that in general, normal, color, and depth textures for a given triangle will not be of the same size.

6.3 Texture Packing

Once all texture patches are available, these are compiled into rectangular texture atlases. Previous work in the area has applied packing algorithms that allow shearing [54] and even resizing [189]. However, this would defeat the purpose of computing an optimal patch size as in Section 6.2.1. Methods for optimal placement of polygons into a square exist, but their high complexity quickly becomes prohibitive for more than a few triangles. Heuristics and machine learning techniques have been used to alleviate these problems [20, 50, 122]. The texture packing algorithm described here is a variation of a bottom-left packing strategy [69, 97]. It packs triangular patches into bigger rectangular textures without resizing or shearing them.

The algorithm creates rows of triangles of similar height. Each input triangle t is aligned such that its longest edge is parallel to the x-axis of the texture, with the remaining vertex above this *base edge*. The angles left and right of the *base edge*

are called the *base angles* $\alpha_l(t)$ and $\alpha_r(t)$, respectively. The height of the triangle is denoted $h(t)$.

In each iteration, a row height $\bar{h}$ is chosen and a set of triangles $T_{\bar{h}}$ whose height differs only slightly from $\bar{h}$ is computed. These triangles are candidates to be placed in the current row. From this set, the best-fitting triangle is selected. The quality of a triangle with respect to a row is determined by comparing the base angles of the new triangle with the free base angle β of the last triangle in the row. The triangle that minimizes $\min(|\alpha_l(t)-\beta|, |\alpha_r(t)-\beta|)$ is inserted into this row. It is aligned with the bottom or top border of the current row and flipped horizontally if $|\alpha_l(t)-\beta| > |\alpha_r(t)-\beta|$. It is then pushed as far to the left as possible. Every second triangle is flipped vertically. New rows are started if a row is full.

Input: Set of triangular patches T
1 **set** $y_{\min} = 0$
2 **while** $|\underline{T}| > 0$ **do**
3 **set** $\bar{h} = \max_{t \in T} h(t)$
4 **set** $T_{\bar{h}} = \{t \in T : h(t) > \bar{h} - \Delta h\}$
5 **set** $\beta = 90°$
6 **set** *mirror* $= false$
7 **while** $|T_{\bar{h}}| > 0$ **do**
8 **set** $t = \arg\min_{t \in T_{\bar{h}}} \{\min(|\alpha_l(t)-\beta|, |\alpha_r(t)-\beta|)\}$
9 **if** $|\alpha_l(t)-\beta| > |\alpha_r(t)-\beta|$ **then**
10 `flipHorizontal`(t)
11 **set** $\beta = \alpha_l(t)$
12 **else**
13 **set** $\beta = \alpha_r(t)$
14 **if** *mirror* **then**
15 `flipVertical`(t)
16 **set** *mirror* $= \neg$*mirror*
17 **set** $T_{\bar{h}} = T_{\bar{h}} \setminus \{t\}$
18 **if** `insert`$(y_{\min}, t)$ **then**
19 **set** $T = T \setminus \{t\}$
20 **set** $y_{\min} = y_{\min} + \bar{h}$

Algorithm 6.1: Packing texture patches into a rectangular texture atlas.

Algorithm 6.1 shows a pseudo-code version of the algorithm. The function `insert`$(\cdot,\cdot)$ inserts a triangle at the given y-Location in the texture. The triangle is inserted on the far right and pushed as far left as possible. If there is not enough space in the texture to accommodate the triangle, the function returns $false$. The algorithm then tries to find another triangle in the height range to add to this row.

Figure 6.2 shows the result of this texture packing algorithm. Typically, 85% to 95% of the texture space is used. Texture usage tends to be better if more and smaller texture patches are available.

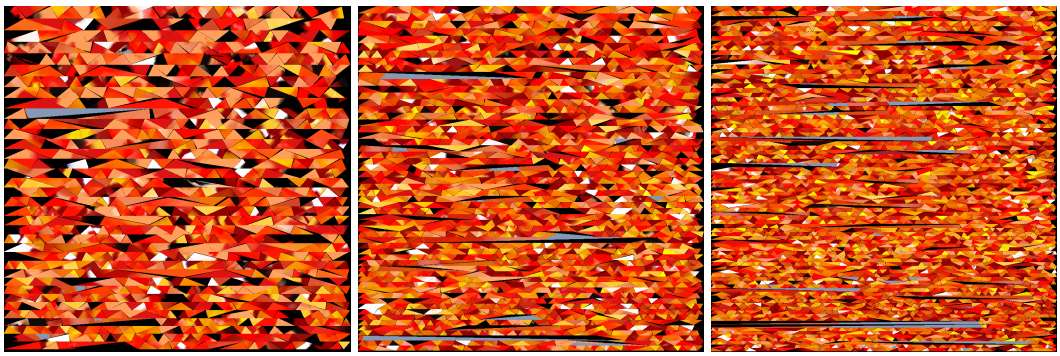


Figure 6.2: Result of the texture packing algorithm for the Fire Dragon model. The texture was split into several $2^n \times 2^n$ pieces. 89% of the texture are used.

6.4 Results

The conversion algorithm was implemented as a Pointshop3D plugin. The only parameters necessary for the conversion are an error bound for the mesh simplification and a error threshold for the texture generation, making conversion simple and intuitive. The algorithm outputs both color and normal textures by default, thus creating a faithful reconstruction of highly detailed surfaces present in the point-sampled original models even for lower resolution meshes.

Figures 6.3 and 6.5 show painted point-sampled models and their corresponding meshes generated by conversion. As can be seen in Figure 6.3 (c) and (d), fine detail present in the original model texture is preserved in the conversion. Surface sampling with surfels and triangles on the back of the bunny model are shown in Figure 6.4.

Since the patch size is computed for each triangle individually, regions with low texture resolution in the input point cloud only take up little space in the output texture. In this example, the texture patch for the triangle containing the bee occupies 5228 pixels in the color texture. The patch of an adjacent, uniformly blue triangle which is of comparable size occupies only 6 pixels. Note that the converted bee patch appears slightly smoother than the surfel rendering. This is due to the additional linear texture interpolation in the mesh-based rendering.

Figure 6.6 shows the effect of simplification on the result. It is hard to visually distinguish between the original point cloud and the output of our conversion algorithm using reasonable simplification parameters. Only after extreme simplification of the mesh, simplification artifacts become visible, especially along the silhouette. These artifacts are due to inappropriate sampling of the geometry, they are not inherent in the conversion algorithm. If generation of normal textures is not enabled, the lack of geometric detail quickly becomes noticeable.

Table 6.1 lists sizes and conversion times for the models.

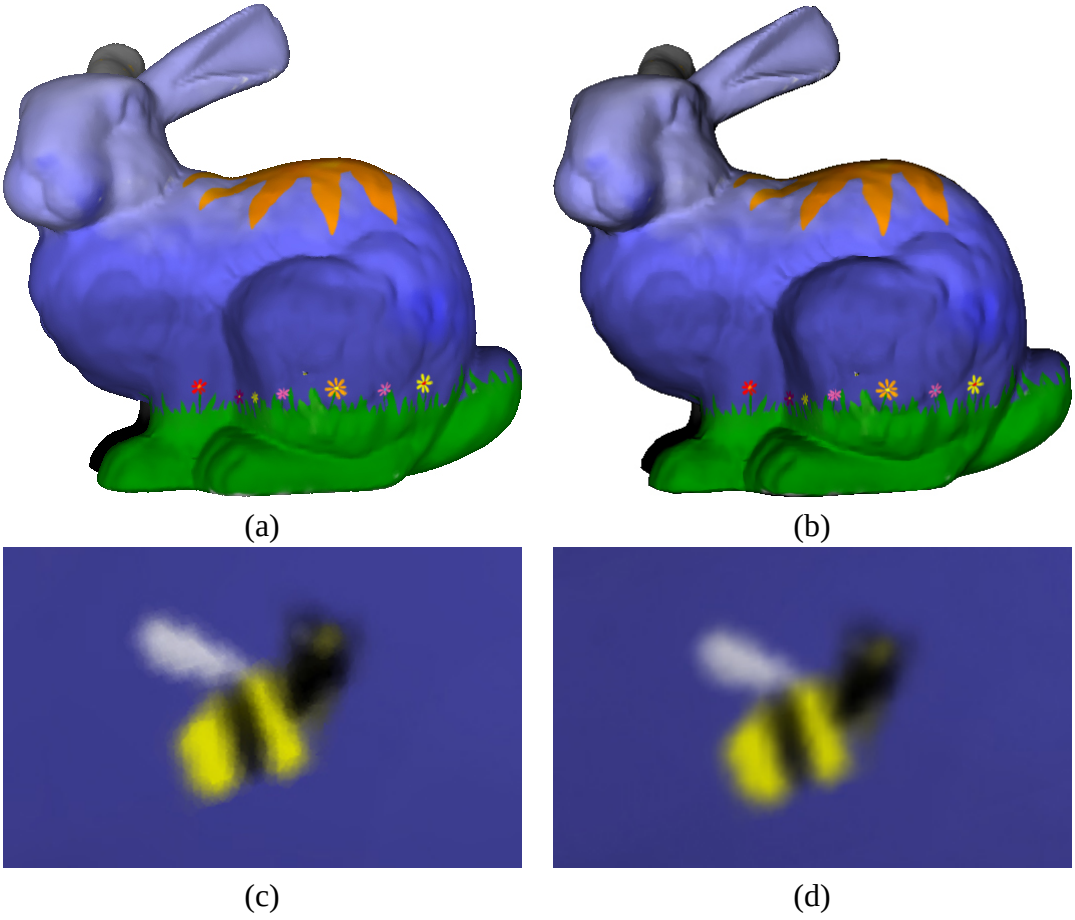


Figure 6.3: A converted painted bunny-model. (a) Surfel model. (b) Textured mesh (10000 triangles). (c) Detail on the surface of the surfel model, (d) on the textured mesh.

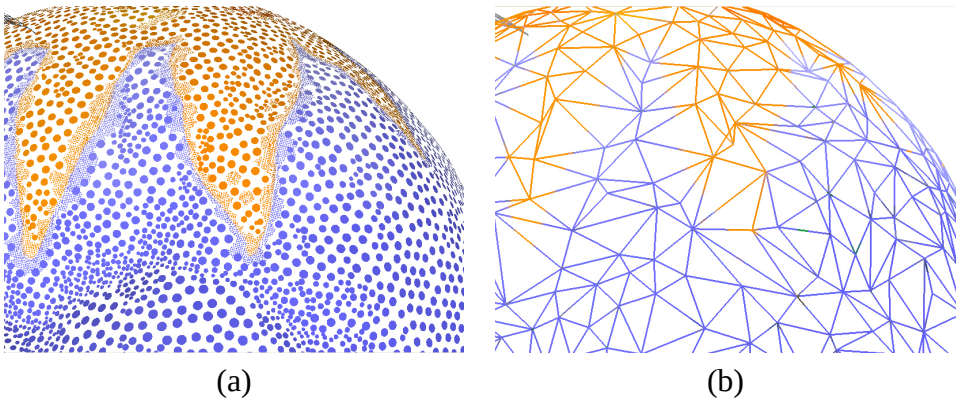
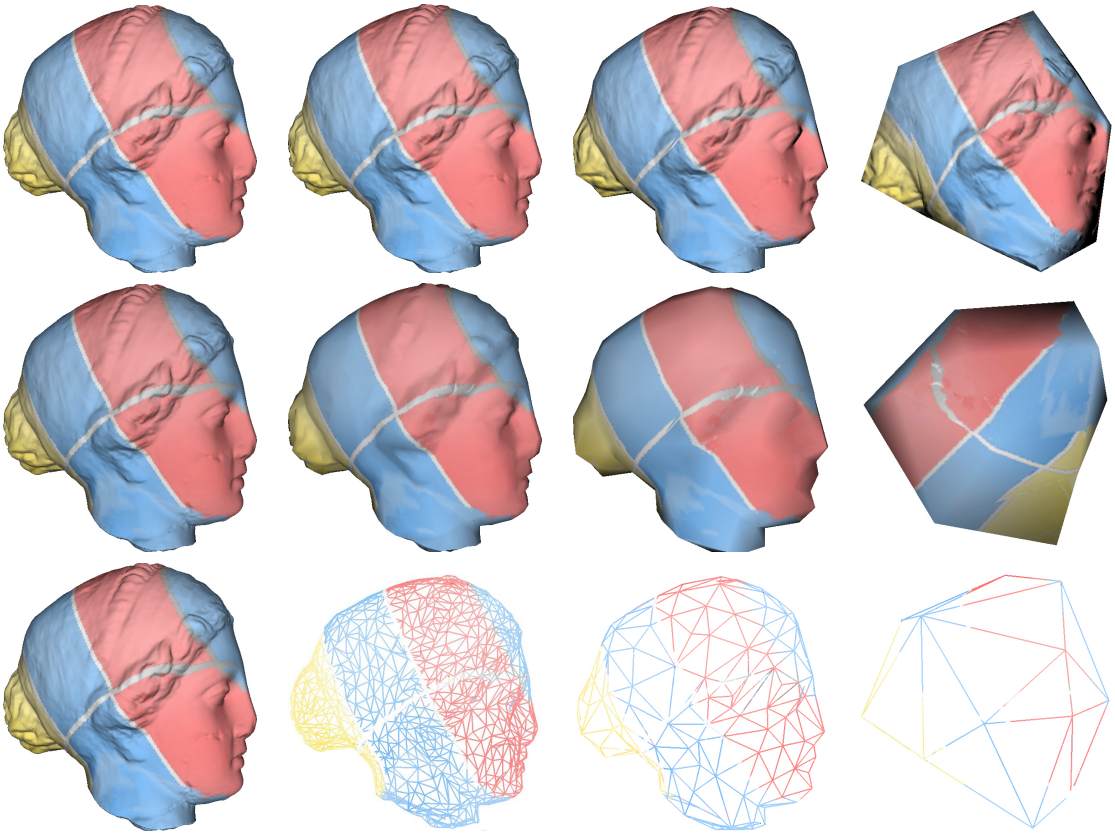


Figure 6.4: Sampling on the back of the bunny model: (a) Point-sampled original. (b) Triangle mesh. The mesh resolution only depends on geometry, the texture detail on the back of the bunny results in upsampling of the corresponding texture patches only.

(a) (b) (c)

Figure 6.5: The (a) original and (b) converted dragon model. (c) Mesh resolution.



(a) (b) (c) (d)

Figure 6.6: Effect of simplification on the converted Igea model. After extreme simplification, discontinuity artifacts become visible in the texture. Top row: (a) 11340 triangles. (b) 3420 triangles. (c) 446 triangles. (d) 30 triangles. Middle row: Models rendered without bump mapping. Bottom Row: (a) Surfel model. (b)-(d) Mesh resolutions.

Model	# Points	# Triangles	Textures	Times [s]
Igea	134345	11340	2.3MB/93%	258,15,35
Bunny	349989	10000	3.8MB/86%	511,28,104
Dragon	553619	30000	12.1MB/89%	907,37,444

Table 6.1: Statistics for converted models. The data shown are: The number of points in the input model, the number of triangles after simplification, number and size of (color) textures, and the timings for triangulation/simplification/texture generation (in seconds).

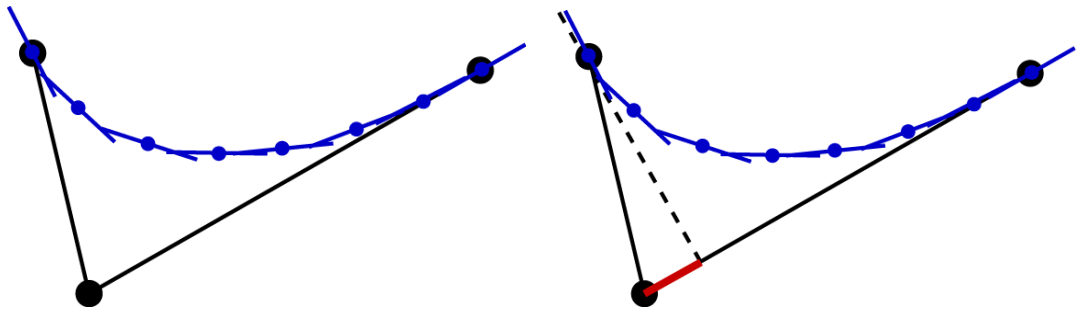
6.5 Discussion

The conversion algorithm proposed in this chapter produces high-quality meshes and textures. If the mesh geometry does not deviate significantly from the surfel geometry, the converted objects are visually indistinguishable from the point-sampled originals. Only under large magnifications, differences become visible. Most of these are caused by linear interpolation between texels, leading to discontinuities at the triangle boundaries if two adjacent triangles have different patch sizes. Higher order texture interpolation [185] alleviates the problem, but it is seldom available in off-the-shelf rendering and editing systems.

Older approaches have used nearest neighbor interpolation [97, 189] or linear interpolation [138] to compute texel values. No texture filtering is performed. Cignoni et al. [54] propose an interesting texture-preserving simplification method that does not assume a specific mesh decimation technique. However, the computed textures are not sensitive to the input texture detail. Both Maruya [138] and Soucy et al. [189] determine patch sizes using the number of vertices projected onto a triangle. This heuristic can result in under-sampling in regions of varying sampling density. If a surface is densely sampled in a uniform color, or densely sampled to represent a linear color gradient, determining the patch size based on the number of vertices leads to large textures where only little information is present on the surface. The adaptive refinement described in Section 6.2.1 works around these problems and guarantees an adequate sampling in all cases.

The resulting texture atlas is tightly packed and does not introduce distortion artifacts due to scaling or shearing of patches. However, it is not well suited for manual editing in an image manipulation program, since adjacent triangles do not have neighboring texture patches in the atlas. It is, however, possible to use editing tools designed to handle texture atlases.

Standard mip-mapping techniques cannot be applied for texture simplification without introducing severe artifacts. However, it is easy to extend the algorithm to produce mip-mapped versions of the texture atlas. Custom tailored mip-mapping can be performed by not only dumping the texture patches at the computed optimal resolution, but also at half and quarter resolutions. The texture packing only needs



(a) (b)

Figure 6.7: Artifacts after extreme simplification. (a) Original geometry and simplified mesh around a saddle point. (b) No texture is available for the lower part of the triangle (red), the texture in the upper part is distorted due to the orthogonal projection.

to be carried out once, the smaller resolution patches are then assembled in the same pattern as the original resolution.

The triangulation of the input model is by far the most time-consuming task. Since the texture generation does not assume anything about how the mesh was acquired, it is possible to substitute any surface reconstruction algorithm for the triangulation and simplification steps. An adaptive version of [101] would be a suitable candidate.

Due to the different interpolation schemes used for textures and point samples, bilinear interpolation and EWA splatting, respectively, the reconstructed color functions look slightly different (e. g. Figure 6.3 (c) and (d)). However, the error metric accounts for these differences and hence the error caused by different interpolation is bounded.

Note that a crucial assumption during texture generation is that the mesh adequately represents the geometry of the input point cloud. Under extreme simplification, textures become distorted and the resulting mesh can even contain triangles that cannot be fully textured using the method described here. In these cases, the orthogonal projection of the object surfels does not entirely cover the triangle area. These triangles typically lie around points of negative Gaussian curvature (see Figure 6.7 for an example).

If the surface deviates from the mesh at the mesh edges, the orthogonal projections either ignore or repeat parts of the surface. Figure 6.8 (a) illustrates the problem. This can cause discontinuity artifacts when the mesh is drastically simplified.

Hale [97] shows how the projection can be adapted to gracefully handle these cases. He interpolates the vertex normals over the area of each triangle in order to find a projection normal for each point on the triangle. Applying the interpolated normals projection to our approach, each surfel has to be splatted using its own projection normal. This method greatly alleviates distortion artifacts, how-

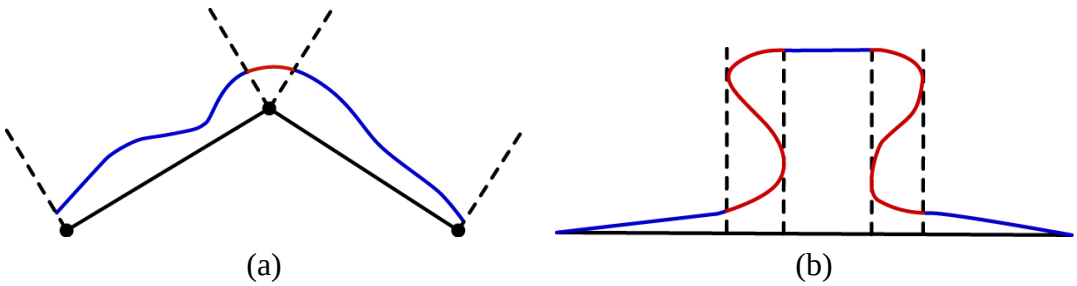


Figure 6.8: (a) Using orthogonal projection of adjacent faces may ignore parts of the original surface (red). (b) A mushroom-shaped geometric detail is discarded by the simplification algorithm. The resulting texture can have discontinuities along the contours of the mushroom.

ever, finding the correct projection normal for a surfel is a nontrivial optimization problem.

Another class of artifacts is introduced by surface patches with depth complexity greater than one. Figure 6.8 (b) shows an example. If a small part of the geometry is entirely discarded by the simplification, the rendering will result in discontinuities along the contours. A remedy to this class of artifacts is to use a parameterization of the original, as done in [138, 189]. It might be possible to modify a method like [138] to use EWA splatting for texture filtering. However, surfels have finite extent and cannot be attributed to only one triangle, complicating the process. A method like this requires triangulation and lacks the flexibility to change the mesh generation method.

Using the method described above, point-sampled models can be converted to triangle meshes. The previous chapters presented work on modeling with point-sampled surfaces, a surface representation with only minimal internal structure. Abandoning consistent connectivity leads to problems, for example with the representation of discontinuities, as discussed in Chapter 4. On the other hand, the lack of structure makes some operations significantly easier, which can be exploited in applications such as the painting system presented in Chapter 5.

In the second part of this thesis, the focus will shift from modeling to animation. The effect of using less structured representations in simulation algorithms will be examined. The next chapter will treat animation of thin shells, using point-sampled surfaces as underlying surface representation, before moving on to algorithms for fluid simulation and continuum elasticity in Chapters 8 and 9, respectively.

Part II

Animation

Chapter 7

Point-Sampled Thin Shells

With the availability of algorithms for shape and appearance modeling that make point-sampled surfaces a fully-fledged general-purpose surface representation, the question of how to animate such surfaces arises. In this chapter, a method for simulation of point-sampled surfaces as thin shells is presented.

Thin shells are almost two-dimensional objects, such as cloth or sheet metal. Many everyday objects are of this type, like clothing, soda cans, and cars. The behavior of these objects is hard to capture with volumetric techniques, since the discretization degrades as the thickness decreases. In order to animate surfaces as thin shells, specialized techniques are called for.

Most thin shell simulation methods use triangle meshes as a material discretization [55, 89, 199]. Since our goal is to animate point-sampled surfaces directly, we avoid triangulation and instead discretize the necessary surface operators without requiring a consistent connectivity.

Other approaches for simulation of point-sampled shells have been proposed. Guo et al. [96] use a global conformal parameterization to construct local frames in which the shell functionals can be evaluated. Since the parameterization is not isometric, the stiffness may vary locally.

Instead of requiring a globally consistent parameterization, we embed splines into the surface. These *fibers* locally measure stretch and curvature. The information measured by the fibers is then consolidated in the nodes of our discretization, yielding a finite-difference type approach.

7.1 Physics of Thin Shells

The fundamental theory underlying thin shells is based on the Kirchhoff-Love theory of thin shells [85]. The thin shell functional depends on surface curvature, and thus contains second order derivatives of the displacement function.

The potential elastic energy stored in a deformed thin shell can be derived as the limit case for an infinitely thin material. The elastic energy then breaks down into a *stretching* (or membrane) term and a *bending* (or flexural) term defined on the surface $\mathcal{S}$ of the thin shell:

$$E_{\text{pot}} = E_s + E_b = \int_{\mathcal{S}} U_s + U_b. \tag{7.1}$$

Here, E_s and E_b are bending and stretching energies and U_b and U_s are the respective energy densities. While the former depends on a first order differential operator, the latter includes a second order term. In addition, the full formulation includes nonlinear geometric differentials, such as curvature, which are very often linearized to become numerically tractable [47].

Finite element solutions for thin shell problems require basis functions with C^1 continuity across elements. Such elements are hard to design and pose significant numerical difficulties, and non-conforming FEM can be used instead [23]. One way of creating conforming higher order terms is to exploiting the specific properties of subdivision surfaces [55].

Following Terzopoulos et al. [199], the elastic energy of the thin shell can be defined in terms of the first and second fundamental forms to penalize deviations from the original shape. Given a surface $\mathcal{S}$ to animate, the energy densities in any point $\mathbf{x} \in \mathcal{S}$ can be written as a functions of the first fundamental tensor $\mathbf{R}$ and the shape operator $\mathbf{S}$:

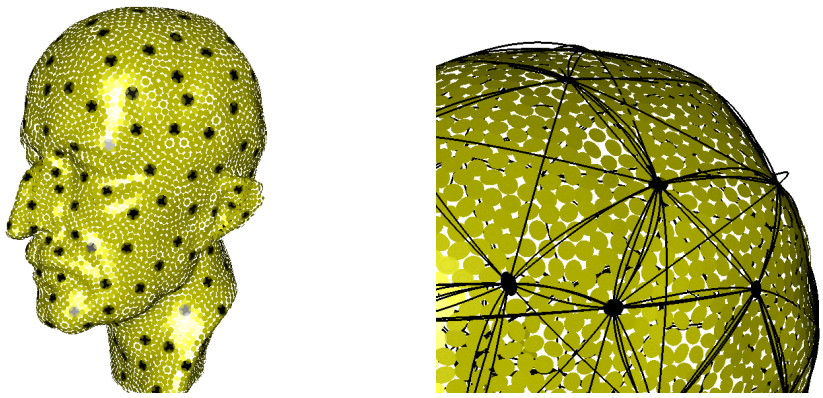
$$U_s = \frac{K_s}{2} \|\mathbf{R} - \mathbf{R}^0\|_p^2, \tag{7.2}$$

$$U_b = \frac{K_b}{2} \|\mathbf{S} - \mathbf{S}^0\|_p^2. \tag{7.3}$$

Here, the superscript 0 denotes the undeformed (rest) value. To measure difference in shape, a pseudo-norm $\|\cdot\|_p$ is used. The parameters K_b and K_s are stiffness constants for bending and stretching deformations respectively. The first fundamental tensor $\mathbf{R}$ measures differential area, while $\mathbf{S}$ measures curvature, thus providing an intuitive interpretation of the energy terms.

7.2 Discretization

The core idea of the method presented here is to sample the surface at distinct points and directions. First, (7.2) and (7.3) are discretized on a set $P \subset \mathcal{S}$ of simulation nodes on the surface. The sampling with nodes does not need to be regular, but it is crucial that the surface is adequately sampled, and that the sampling



(a) (b)

Figure 7.1: (a) Sampling of the surface with simulation nodes. Fibers are created connecting the simulation nodes, the other points defining the surface are not used in the simulation, and are passively animated. (b) The network of fibers on the surface of the object.

density does not vary abruptly. Given a point-sampled input surface, nodes are chosen using a clustering algorithm also used for simplification [158], yielding a good sampling with simulation nodes that adequately represents the surface geometry (see also Section 3.3). The simulation nodes carry the mass of the model, and the energy is discretized in the node positions. In each node, a set of parametric curves is fitted to the surface. These *fibers* measure normal curvature and arc length in their respective directions. Figure 7.1 illustrates the sampling. Surfels that are not used as simulation nodes are passively animated using skinning as described in Section 7.4.

7.2.1 Fibers

Each node represents a small surface patch on the thin shell. The fibers passing through a node are used to calculate the surface area and curvature at that point, which are in turn used to compute the elastic energy density. Fibers are natural cubic splines which are defined by three points: a central node and two of its neighboring nodes. The neighborhood of one such node with embedded fibers is shown in Figure 7.2.

Fiber creation

The fibers sample a three dimensional space: two dimensions for the position on the surface, plus one dimension representing the direction of the fibers. We have to make sure that all these dimensions are adequately sampled. While the clustering algorithm used for node selection ensures a good sampling for the positions on the

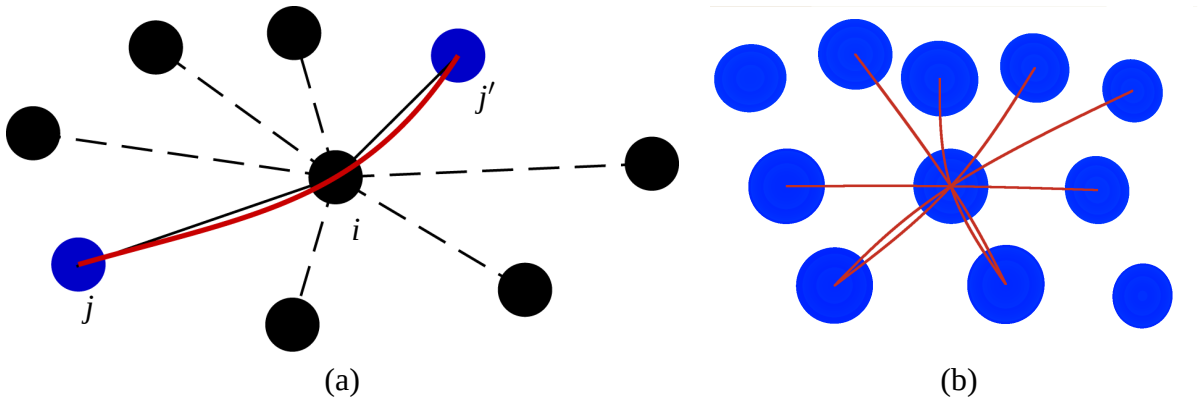


Figure 7.2: Fiber creation. (a) In a neighborhood of nodes N_i around a central node at $\mathbf{x}_i$, one fiber is created for each $j \in N_i$, connecting the nodes j, i, and j', where j' is the most opposite node to j. (b) Fibers created for one neighborhood on an actual surface. Note that several fibers can end in the same point.

surface, a roughly isotropic sampling in the directional domain has to be enforced by the fiber creation algorithm.

A simple heuristic creates such an isotropic sampling. For a node i at position $\mathbf{x}_i$, let $N_i = \{j_1, \ldots, j_{n_i}\}$ be the set of nearest neighbor nodes to i. For each node $j \in N_i$, we find the most opposite point with respect to $\mathbf{x}_i$, i.e.

$$j' = \arg\min_{j' \in N_i} (\mathbf{x}_{j'} - \mathbf{x}_j) * (\mathbf{x}_j - \mathbf{x}_i). \tag{7.4}$$

A fiber connecting the nodes i, j and j' is then created, duplicate fibers are discarded (see Figure 7.2). There are pathological cases in which this heuristic leads to degenerate sampling, in particular when the neighborhood is highly anisotropic. However, since we control which points on the surface become simulation nodes, these cases can be easily avoided. The clustering algorithm used to select simulation nodes produces well-behaved neighborhoods.

Each fiber is a natural cubic spline. Its parametric representation is given as a piecewise cubic polynomial $\mathbf{f}(t)$ whose coefficients linearly depend on the defining points. The exact expressions for the coefficients are given in Appendix B.

Membrane Energy

The surface area represented by each node is determined by the local sampling density. Because of the way the node position on the surface are chosen, the neighborhood shape is roughly circular for each node. This fact can be used to define an approximation of the area that each node represents. Each fiber k provides us with a measurement of the area associated with its central point:

$$A_k = \frac{\pi}{4} l(\mathbf{f}_k)^2, \tag{7.5}$$

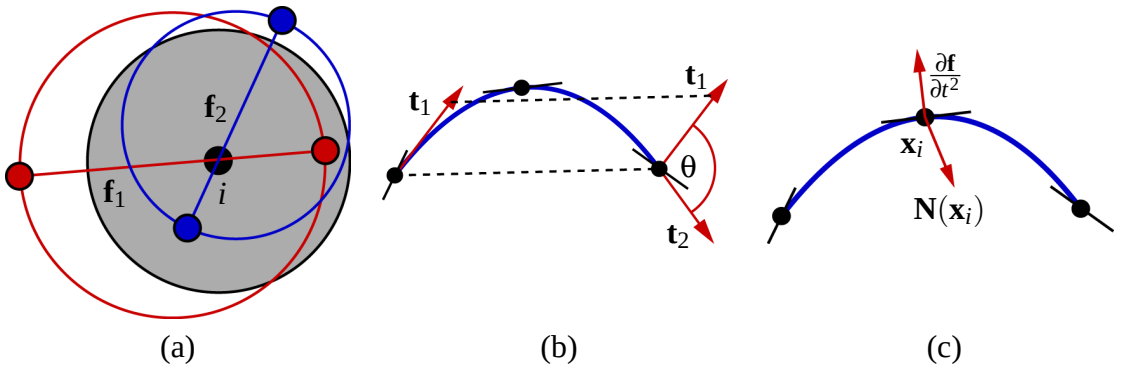


Figure 7.3: (a) Area approximation using two arc length samples from the fibers $\mathbf{f}_1$ and $\mathbf{f}_2$. Each fiber estimates the area of the sample according to its own length only. The area assigned to i is the average of the fiber areas. (b) Approximating the curvature of a fiber through $\mathbf{x}_i$. The angle θ between the tangents $\mathbf{t}_1$ and $\mathbf{t}_2$ at the end points is used as a measure for curvature. (c) θ is oriented according to surface normal $\mathbf{N}(\mathbf{x}_i)$ and the second derivative of the spline defining the fiber, $\frac{\partial \mathbf{f}}{\partial t^2}$.

where $l(\mathbf{f}_k)$ is the arc length of the fiber k. Computing the arc length of a cubic spline involves solving an elliptic integral [21]. Therefore, we use numeric integration to approximate the arc length, yielding a function $\tilde{l}$. The approximation is sufficiently accurate even with a very small number of sample points. It is described in detail in Appendix B.

Using the area estimate by the fibers through a node i, we can define a measure for the current area A_i of a surface patch represented by the node i by simply averaging the fiber estimates:

$$A_i \approx \tilde{A}_i = \frac{1}{m_i} \sum_k \tilde{A}_k = \frac{\pi}{4m_i} \sum_k \tilde{l}(\mathbf{f}_k)^2. \tag{7.6}$$

Here, m_i is the number of fibers through the node i, and all fibers k pass through node i. Figure 7.3 (a) shows an illustration of the approximation. Note that area approximations for all nodes do not add up to the total area of the surface $\mathcal{S}$. However, since the membrane energy only depends on changes in area, this is not a problem during the simulation.

Now, the discrete membrane energy can be defined in terms of the area estimates for each simulation node:

$$\tilde{E}_s(\mathbf{x}_i) = \frac{K_s}{2} \left[\left(\tilde{A}_i - \tilde{A}_i^0\right)^2 + \frac{1}{m_i} \sum_k \left(\tilde{A}_k - \tilde{A}_k^0\right)^2 \right]. \tag{7.7}$$

The energy $\tilde{E}_s(\mathbf{x}_i)$ becomes nonzero when either the total area $\tilde{A}_i$ or the individual fiber areas $\tilde{A}_k$ change. This amounts to preserving the area represented by each point, as well as the shape of the area element.

Flexural Energy

The shape operator $\mathbf{S}$ measures curvature: $Tr(\mathbf{S}) = 2H$, where H denotes the mean curvature and $Tr(\cdot)$ is the matrix trace. Noting that the trace is a pseudo-norm as well as a linear operator, the flexural energy density U_b can be written as

$$U_b = \frac{K_b}{2}\left(H - H^0\right)^2. \tag{7.8}$$

For a sample point i, the mean curvature H_i can be approximated using the directional curvature samples given by the fibers through $\mathbf{x}_i$. We can write:

$$H_i \approx \tilde{H}_i = \frac{1}{m_i}\sum_k \kappa(\mathbf{f}_k), \tag{7.9}$$

where $\kappa(\mathbf{f}_k)$ is the curvature of a fiber k, corresponding to the normal curvature of the surface in its direction.

As a measure for the curvature of the fiber, we use the angle θ between the tangents at the start and end point of the fiber. In order to avoid flipping problems, the angle is oriented according to the surface normal $\mathbf{N}(\mathbf{x}_i)$ in the center point of the fiber. Figure 7.3 shows an illustration.

Specifically, a fiber curvature is negative if the second derivative of its defining spline $\mathbf{f}(t)$ points in the same direction as the surface normal $\mathbf{N}(\mathbf{x})$ in the central point.

$$[\theta < 0] \Leftrightarrow [\mathbf{N}(\mathbf{x}) * \frac{\partial \mathbf{f}}{\partial t^2}(0.5) > 0]. \tag{7.10}$$

To obtain the bending energy, we integrate the energy density over the surface area represented by a sample point. Assuming that the energy density is constant over the surface element, the bending energy becomes

$$\tilde{E}_b(\mathbf{x}_i) = \tilde{U}_b(\mathbf{x}_i)\tilde{A}_i^0 = \frac{\tilde{A}_i^0 K_b}{2}\left(\tilde{H} - \tilde{H}^0\right)^2. \tag{7.11}$$

7.2.2 Dynamic Behavior

With the energies defined as above, the dynamic behavior can be computed by deriving forces for the simulation nodes. The elastic force for each node is given as the negative gradient of the potential energy with respect to the node's position:

$$\mathbf{F}_i = -\nabla_{\mathbf{x}_i}(\tilde{E}_s + \tilde{E}_b). \tag{7.12}$$

In the discrete setting, the force on simulation node i can be written as a sum over the contributions of all simulation nodes:

$$\mathbf{F}_i = \sum_j \mathbf{F}_{ji} = -\sum_j \left(\nabla_{\mathbf{x}_i}\tilde{E}_s(\mathbf{x}_j) + \nabla_{\mathbf{x}_i}\tilde{E}_b(\mathbf{x}_j)\right). \tag{7.13}$$

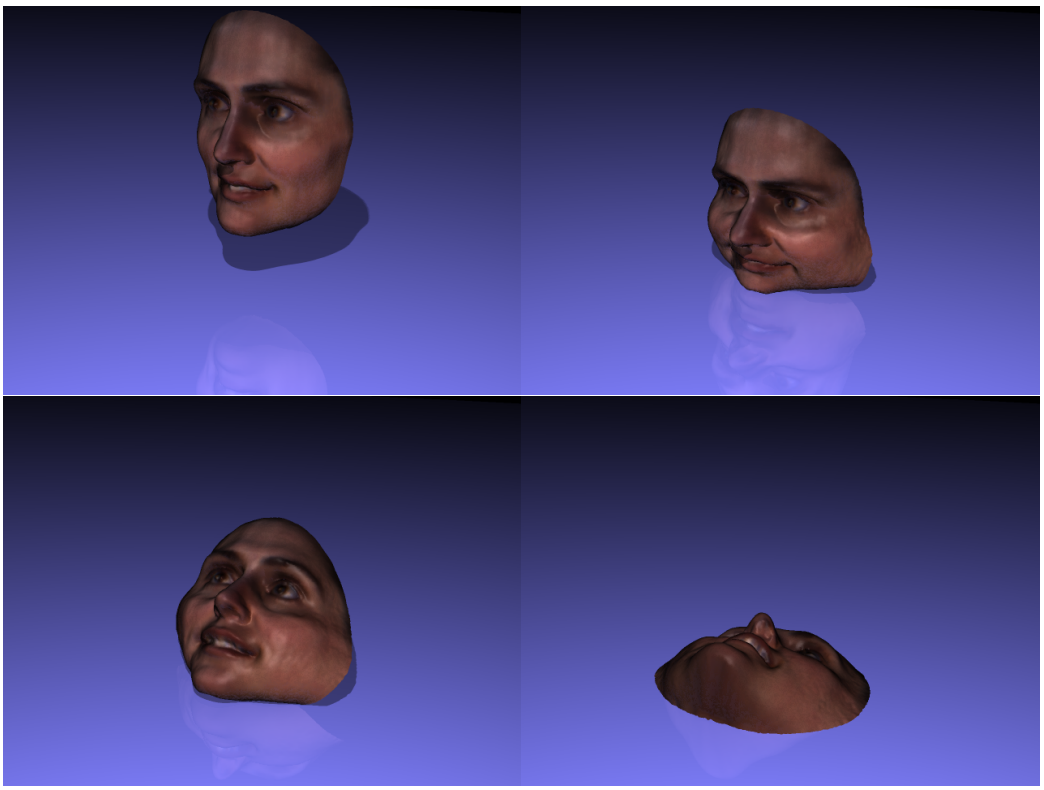


Figure 7.4: An elastic mask is dropped. The shell deforms and bounces, before coming to a rest state on the surface.

Substituting the membrane and flexural energies (7.7) and (7.11) into (7.13) and evaluating the gradients, we obtain

$$\begin{aligned} \mathbf{F}_{ji} &= -\frac{K_s}{m_j}\sum_k \left(\tilde{A}_k - \tilde{A}_k^0\right)\nabla_{\mathbf{x}_i}\tilde{A}_k \\ &\quad -K_s\left(\tilde{A}_j - \tilde{A}_j^0\right)\nabla_{\mathbf{x}_i}\tilde{A}_j \\ &\quad -\tilde{A}_j^0 K_b\left(\tilde{H}_j - \tilde{H}_j^0\right)\nabla_{\mathbf{x}_i}\tilde{H}_j. \end{aligned} \tag{7.14}$$

Noting that both $\nabla_{\mathbf{x}_i}\tilde{A}_j$ and $\nabla_{\mathbf{x}_i}\tilde{H}_j$ are sums over the same fibers, we can split the force $\mathbf{F}_{ji}$ into components induced by individual fibers k:

$$\mathbf{F}_{ji} = \sum_k \mathbf{F}_{ki}, \tag{7.15}$$

$$\begin{aligned} \mathbf{F}_{ki} &= -\frac{K_s}{m_j}\left(\tilde{A}_j + \tilde{A}_k - \tilde{A}_j^0 - \tilde{A}_k^0\right)\nabla_{\mathbf{x}_i}\tilde{A}_k \\ &\quad -\tilde{A}_j^0\frac{K_b}{m_j}\left(\tilde{H}_j - \tilde{H}_j^0\right)\nabla_{\mathbf{x}_i}\kappa(\mathbf{f}_k). \end{aligned} \tag{7.16}$$

$\mathbf{F}_{ki}$ is the force that the fiber k, passing through the node j, exerts on the node i. The gradient of one fiber's area estimate $\tilde{A}_k$ is given by

$$\nabla_{\mathbf{x}_i} \tilde{A}_k = \frac{\pi}{2} \tilde{l}(\mathbf{f}_k) \nabla_{\mathbf{x}_i} \tilde{l}(\mathbf{f}_k). \tag{7.17}$$

Expressions for the gradients of $\tilde{l}(\mathbf{f}_k)$ and $\kappa(\mathbf{f}_k)$ are stated in Appendix B. Taking all surface elements into consideration, the resulting forces add up to zero.

Forces are computed in two passes. In a first pass, $\tilde{l}(\mathbf{f}_k)$ and $\kappa(\mathbf{f}_k)$ and their respective gradients are computed for all fibers. In a second pass over all simulation nodes, (7.16) can be evaluated and the force contributions of the fibers are summed in the nodes to obtain $\mathbf{F}_i$.

The governing equation for our system is

$$-\nabla_{\mathbf{x}} E_{\text{pot}} + \mathbf{M}\ddot{\mathbf{X}} + D\dot{\mathbf{X}} = \mathbf{F}_{\text{ext}}, \tag{7.18}$$

where $\mathbf{M}$ is the mass matrix containing the masses of the simulation nodes in its diagonal, D is a damping coefficient, and $\mathbf{F}_{\text{ext}}$ denotes the external forces acting on the system. The system state $\mathbf{X}$ contains the position of all simulation nodes.

Upon computing $\mathbf{F}_i$ for all simulation nodes, the velocity Verlet integration scheme [207] is used to compute a time step in (7.18) and animate the model. Figure 7.4 shows elastic deformation of a point-sampled mask.

7.3 Material Properties

So far, only the behavior of purely elastic objects has been described. However, most objects exhibit some plasticity, or fracture under high stress. This section describes how plasticity and fracturing can be integrated into the simulation.

7.3.1 Plasticity

Materials that exhibit plastic behavior remember part of their deformation. Elastic forces will not restore the original shape completely. In our model, the rest shape of an object is stored in the rest shape of the fibers used to sample it. It is thus possible to reduce the plasticity computations to single fibers.

The elastic force for a fiber is computed using its rest arc length l^0 and rest tangential angle θ^0, as well as the current length l and current tangential angle θ. To incorporate plasticity, the rest length and rest angle used to compute the elastic force is expressed in terms of the original value and a plastic deformation

$$\begin{aligned} l^0 &= l_{\text{orig}} + l_{\text{plastic}}, \\ \theta^0 &= \theta_{\text{orig}} + \theta_{\text{plastic}}. \end{aligned} \tag{7.19}$$

This approach is equivalent to storing plastic strain in FEM simulations [147].

The plastic deformation represented by l_{plastic} and θ_{plastic} changes whenever the deformation of the fiber becomes too high. The rest arc length of a fiber changes if

$$\beta_s < |l - l^0|, \tag{7.20}$$

where β_s is the plastic yield constant for stretching deformations. Similarly, the curvature θ_{plastic} is updated if

$$\beta_b < |\theta - \theta^0|, \tag{7.21}$$

where β_b is the plastic yield threshold for bending deformations. We update the rest state similar to [154], however accounting for plastic creep by adding time dependence to the update rule:

$$\Delta l_{\text{plastic}} = \alpha_s(l - l_0)\Delta t, \tag{7.22}$$
$$\Delta \theta_{\text{plastic}} = \alpha_b(\theta - \theta_0)\Delta t. \tag{7.23}$$

In each time step in which (7.20) or (7.21) hold, l_{plastic} and θ_{plastic} are changed according to (7.22) and (7.23) respectively. The plastic creep parameters $\alpha_{s,b}$ determine how fast the material can adapt to the new state. In addition to this basic form of plasticity, a variety of other nonlinear effects can be modeled, an example is given in Section 7.3.3.

Figure 7.5 shows a simulation with plastic effects. A heavy bronze bust is simulated as a thin shell. As it hits the floor, it deforms plastically and remains in the deformed state. Note that the expected behavior for a solid object would be to form a flat surface where plastically deformed by the impact. However, due to the lack of volume preservation forces, a shell caves in.

7.3.2 Fracture

The strength of influence a node has on other nodes is determined by the *material distance* between those nodes. By connecting neighboring nodes with fibers, we implicitly assume that the Euclidean distance is a good approximation to material distance. When a material fractures or tears, parts of the material that are separated by a crack do not influence each other, even though they might are close. Euclidean distance alone is no longer useful in determining the coupling between nodes.

In order to model fracture in the simulation framework, the notion of spatial neighborhood needs to be redefined to give a better approximation to *material distance*. This is achieved by introducing a visibility criterion when determining neighborhoods once fracture has occurred.

A crack is initiated if the stress on some fiber in the model becomes too high, specifically if one of

$$\begin{aligned} \gamma_s &< l/l^0, \\ \gamma_b &< |\theta - \theta^0|, \end{aligned} \tag{7.24}$$

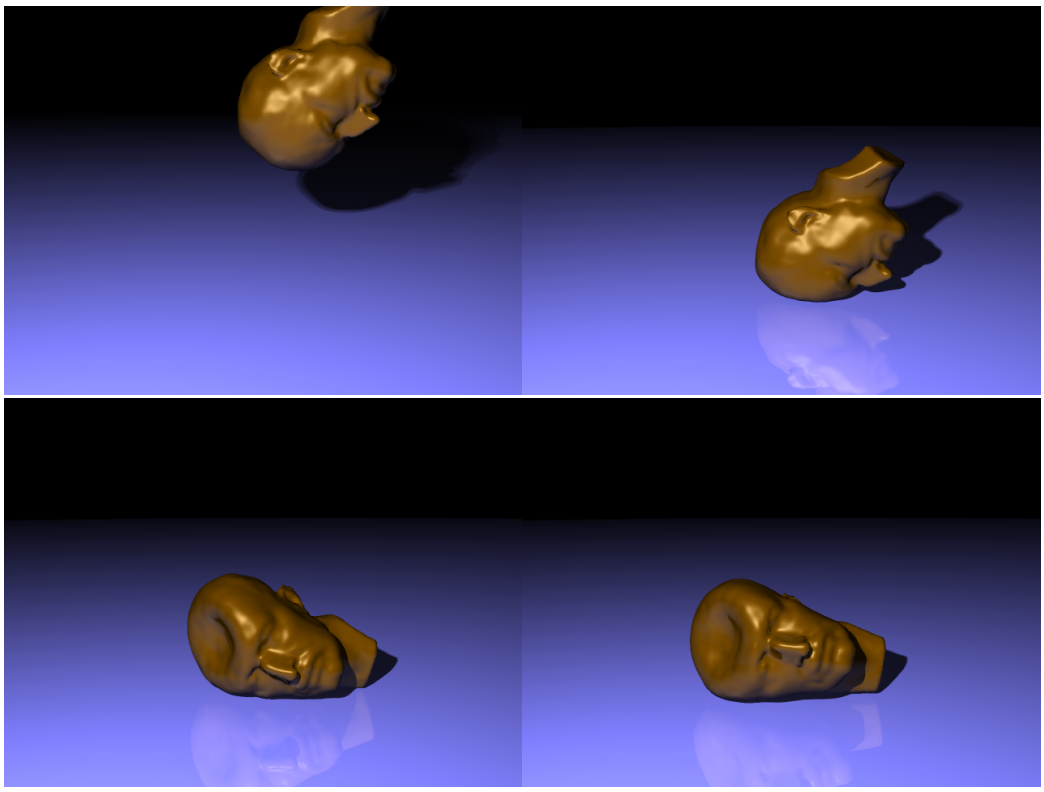


Figure 7.5: A heavy bronze bust is dropped on the floor and deforms plastically on impact.

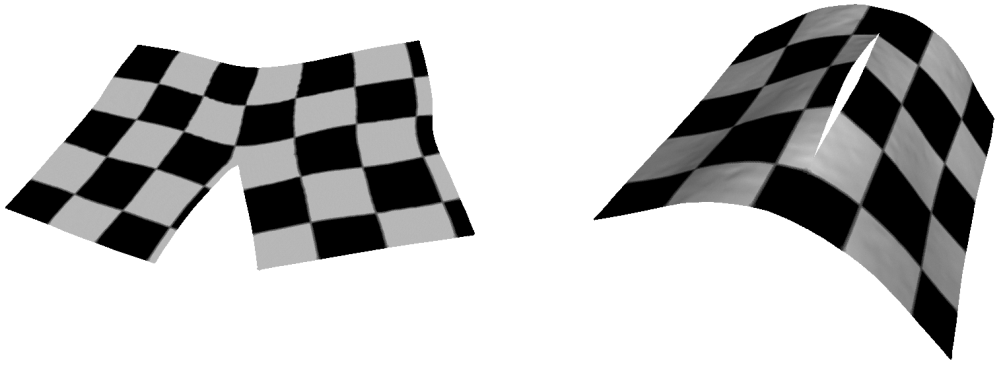
holds for any fiber. γ_s and γ_b are fracture thresholds for tensile stretching and bending fracture respectively. Figure 7.6 shows the two cases.

Once a fiber is chosen, a small disc, the crack plane, is placed at the center node $\mathbf{x}$ of the fiber. Its normal equals the fiber's tangent in $\mathbf{x}$, and its radius is equal to the average node distance in the neighborhood.

The neighborhoods of nearby simulation nodes are then recomputed accounting for crack planes. Only nodes that are not separated by a crack plane can be in the same neighborhood. This yields a better approximation to material distance in the presence of cracks. Fibers within those new neighborhoods are initialized as described in Section 7.2.1, using the original positions of the simulation nodes.

This procedure effectively cuts any fibers that would intersect with a crack plane. The recomputation of neighborhoods makes sure that the material remains adequately sampled with fibers, also along the crack surfaces. Figure 7.7 illustrates the procedure.

The crack plane can be used to create discontinuity surfels for accurate edge rendering as described in Section 4.3. In this case, two discontinuity surfels with opposing normals are created. These new surfels are animated with the other object surfels as described in Section 7.4. Figure 7.8 shows tearing of a thin shell.



(a) (b)

Figure 7.6: Different fracture modes. (a) A crack introduced by tensile stress. (b) Bending stress.

7.3.3 Nonlinear Effects

Many materials change their material properties depending on the current strain state or the amount of plastic deformation they undergo. It is easy to integrate such effects into the simulation framework. Since all material properties are stored with the individual fibers, the parameters can be varied depending on the current state of the fiber (the local strain state of the material), or past deformations of the fiber.

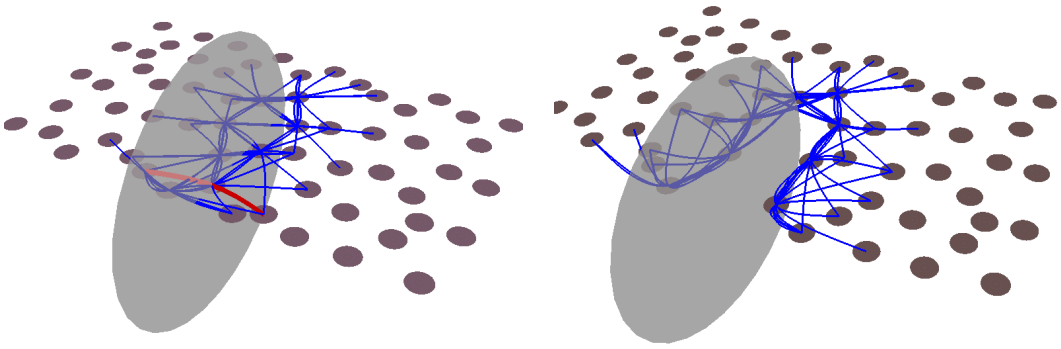
As an example, the following update rule for the fracture thresholds implements a behavior often found in hard plastics and metals. The fracture threshold depends on the deformation history of the material, the material weakens under repeated plastic bending:

$$\Delta\gamma_b = -\chi\gamma_b|\Delta\theta_{\text{plastic}}|. \tag{7.25}$$

Here, the parameter χ determines how much the material is weakened by plastic deformation. $\Delta\theta_{\text{plastic}}$ is the change of plastic deformation as defined in (7.23), which is evaluated in each time step.

7.3.4 Anisotropy

The fiber-based energy as described above can easily be extended to handle anisotropic and inhomogeneous materials. Such materials exhibit variable stiffness depending on stress direction, or position. The best way to integrate anisotropy into the fiber-based energy is to use a roughly regular and isotropic sampling throughout the model, and vary the stiffness constants of the individual fibers according to their position or direction. It would be possible to achieve anisotropic behavior by deliberately creating an anisotropic fiber sampling. However, as the approximations used in the derivation of the energy rely on isotropic sampling, this assumption should not be violated.



(a) (b)

Figure 7.7: (a) When the stress along some fiber (red) exceeds a fracturing threshold, a new crack plane orthogonal to the fiber tangent is created. (b) Fiber sampling after neighborhood recomputation. Fibers intersecting the crack plane have been deleted, the edges have been resampled.

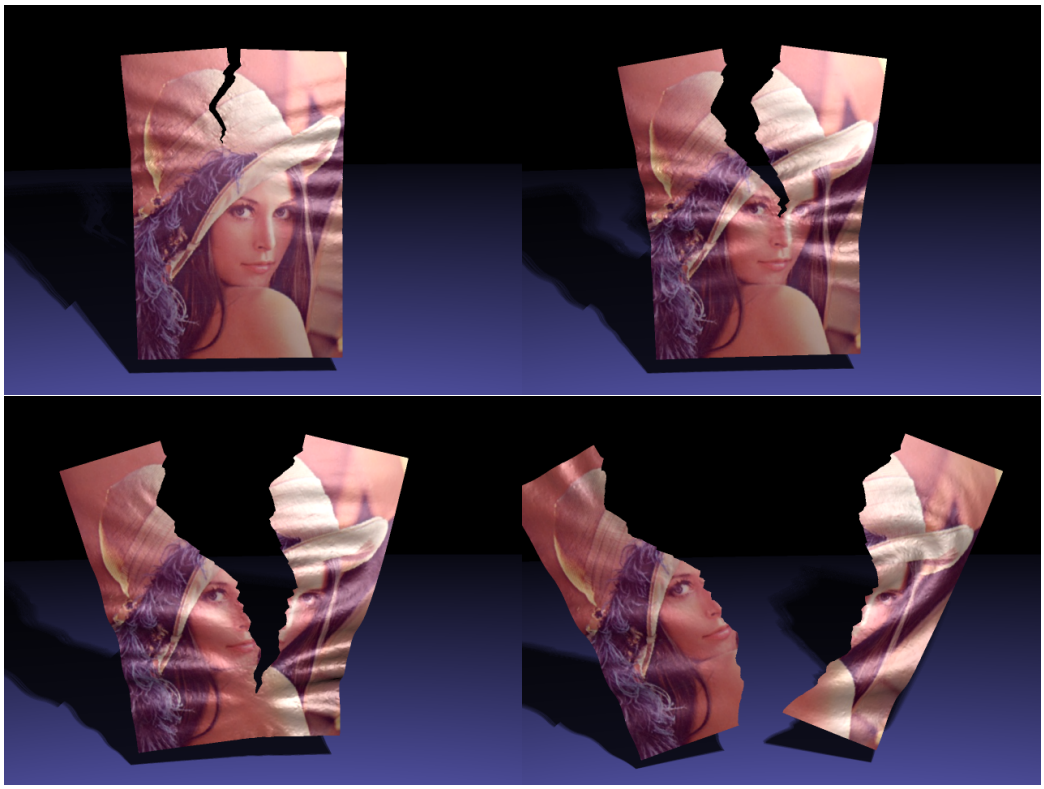


Figure 7.8: A large poster is torn apart by external forces. The crack starting point is controlled by adding a weak point to the top edge of the sheet. Since the material is not brittle and the forces are moderate, the crack propagates slowly, and the material tears.

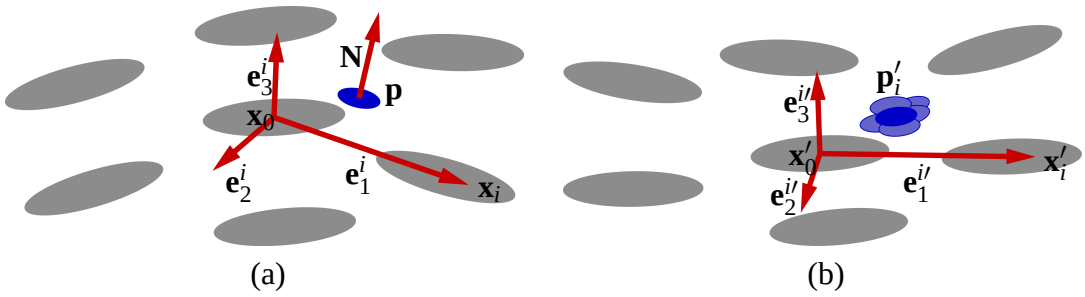


Figure 7.9: Deforming the surface using local coordinate frames. The deformation field computed for the simulation nodes is interpolated to a surface point **p** (blue). (a) In the rest state, a local coordinate frame is computed for each neighboring node (gray) of the nearest simulation node $\mathbf{x}_0$. (b) In the transformed configuration, the local coordinates are used to reconstruct a deformed position for **p** by blending the deformed local positions obtained using the local coordinates for each neighboring node.

7.4 Surface Animation

The surface of the thin shell is only sparsely sampled with simulation nodes. The deformation computed for these nodes has to be re-mapped onto the much denser surfel sampling that is used for rendering. The method applied here is somewhat similar to the skinning approach proposed by Singh and Kokkevis in [187]. While they compute barycentric coordinates and a normal displacement on a triangle mesh, we use coordinate systems defined by neighboring simulation nodes to encode the distance to the closest active node.

For each *passive point* on the surface that needs to be deformed, local coordinates with respect to its neighborhood of simulation nodes are computed. When the object deforms and the simulation nodes move, an updated position of the passive point can be computed using these local coordinates.

Let **p** be the position of a passive point and $P_\mathbf{p} = \{\mathbf{x}_i, i = 0 \dots n_\mathbf{p}\}$ be the set of neighboring simulation nodes in their undeformed state, ordered such that $\|\mathbf{x}_i - \mathbf{p}\| \leq \|\mathbf{x}_{i+1} - \mathbf{p}\|$. In a first step, a least squares plane through the $\mathbf{x}_i$ is computed, yielding a (normalized) plane normal **N**. Using the plane normal and the position of one of the neighbors $\mathbf{x}_i \in P_\mathbf{p}$ relative to the nearest neighbor $\mathbf{x}_0$, $n_\mathbf{p}$ coordinate systems $L_i = \{\mathbf{e}_1^i, \mathbf{e}_2^i, \mathbf{e}_3^i\}$ are computed:

$$\mathbf{e}_1^i = \mathbf{x}_i - \mathbf{x}_0, \qquad \mathbf{e}_2^i = \frac{\mathbf{N} \times \mathbf{e}_1^i}{\|\mathbf{e}_1^i\|}, \qquad \mathbf{e}_3^i = \frac{\mathbf{e}_2^i \times \mathbf{e}_1^i}{\|\mathbf{e}_1^i\|}, \tag{7.26}$$

see Figure 7.9 for an illustration. Each of these coordinate systems measures surface stretch in the direction $\mathbf{e}_1^i$, and accounts for rotations around $\mathbf{e}_1^i$. Due to the normalization of $\mathbf{e}_2^i$ and $\mathbf{e}_3^i$, the coordinate system L_i does not capture surface stretch in other directions.

Finally, $\mathbf{p}-\mathbf{x}_0$ is transformed into each of the local frames L_i, yielding coordinates $\mathbf{y}^i = \{y_1^i, y_2^i, y_3^i\}$. Local coordinates are computed as

$$y_k^i = \frac{(\mathbf{p}-\mathbf{x}_0) * \mathbf{e}_k^i}{\|\mathbf{e}_k^i\|^2}. \tag{7.27}$$

The y_k^i are stored with the passive point. When the surface deforms, deformed local frames $L_i' = \{\mathbf{e}_1^{i\prime}, \mathbf{e}_2^{i\prime}, \mathbf{e}_3^{i\prime}\}$ can be computed analogously to (7.26). Then, the stored local coordinates are transformed back to world coordinates, yielding $n_\mathbf{p}$ positions

$$\mathbf{p}_i' = \mathbf{x}_0' + y_1^i \mathbf{e}_1^{i\prime} + y_2^i \mathbf{e}_2^{i\prime} + y_3^i \mathbf{e}_3^{i\prime}. \tag{7.28}$$

The final deformed position $\mathbf{p}'$ is a computed as a weighted sum of the points $\mathbf{p}_i'$:

$$\mathbf{p}' = \frac{\sum_i w_i \mathbf{p}_i'}{\sum_i w_i}, \tag{7.29}$$

where the weights are given by

$$w_i = w\left(\frac{\|\mathbf{x}_i - \mathbf{p}\|}{\|\mathbf{x}_{n_\mathbf{p}} - \mathbf{p}\|}\right), \tag{7.30}$$

using some smooth weight function $w(\cdot)$ satisfying $w(0) = \infty$ and $w(1) = 0$. Thus, the surface deformation is smooth and interpolates the deformation given by the nodes.

Alternative methods for the animation of point-sampled surfaces from sampled deformation fields have been proposed [111, 112]. Keiser's method deforms surface points using first order approximations of the deformation fields computed in each simulation node. In the thin shell setting, a full least squares approximation of the Jacobian of the deformation field, as required by [112], is not straightforward to compute. Since the geometry is (locally) planar, the sampling with simulation nodes is degenerate for the purposes of computing a least squares approximation. Therefore, no full first-order approximations to the Jacobian are available. Instead, approximations to directional deformation gradients are only computed in a finite difference fashion. Each local coordinate system measures a deformation gradient in direction of $\mathbf{e}_1^i$.

7.5 Results

The animation method described above can be used to animate point-sampled models without requiring a triangulation of the surface. It supports a variety of physical effects such as plasticity, and fracture. Nonlinear material behavior such as material fatigue can be modeled on a per-fiber level.

			Average time/frame [ms]			
Figure	# Nodes	# Points	Forces	Fracture	Surface	Total
7.5	931	9835	7	—	64	71
7.10	1170	24578	24	23	166	212
7.4	1987	40880	222	—	325	547
7.8	11860	240000	1533	6066	1459	9058

Table 7.1: Simulation times. Shown are (from left to right): number of simulation nodes, number of surface points, time for force computation and integration, time for fracture testing and neighborhood recomputation, time for high-resolution surface animation, total animation time per frame.

Figure 7.4 shows an elastic mask dropping on the floor. The mask is sampled with 9835 surfels, approximately every tenth surfel is used as a simulation node. The material is fully elastic.

Figure 7.5 illustrates plasticity. The additional plasticity computations have no significant impact on simulation times.

In Figure 7.8, a poster is torn apart. A weak point in the middle of the top edge determines the starting point of the crack. The material has a very high stretching stiffness. The temporal discontinuities introduced by the fracture process lead to small-scale oscillations along the tear. The geometry is initially planar, and its texture is represented using 240000 surfels. The dynamic behavior is computed using 11860 nodes. Fracture computations are quite expensive as neighborhoods have to be recomputed while considering visibility.

Even though the sampling with nodes and fibers is quite coarse in the models shown, the method yields surprisingly realistic results. Figure 7.10 shows an animation of a tearing balloon, sampled with 1170 nodes.

The simulation parameters used to produce the simulations shown in the examples are given in Appendix F.1. Table 7.1 summarizes the timing data for the scenes shown in Figures 7.4, 7.5, 7.8, and 7.10. Simulation times were measured on a Pentium 4 at 3 GHz. As can be seen from the data, the animation of the high-resolution surface is relatively expensive. Since it is only performed once per frame, it is a good alternative to a full simulation if the simulation time step is small compared to the frame time. For slow motion animations, such as the balloon sequence, surface reconstruction becomes the limiting factor.

7.6 Discussion

The discretization approach described in Section 7.2 will lead to different material behavior depending on the sampling chosen. Since the forces are normalized, increasing the number of fibers per point does not increase material stiffness, however, anisotropic sampling results in anisotropic material behavior. This can also

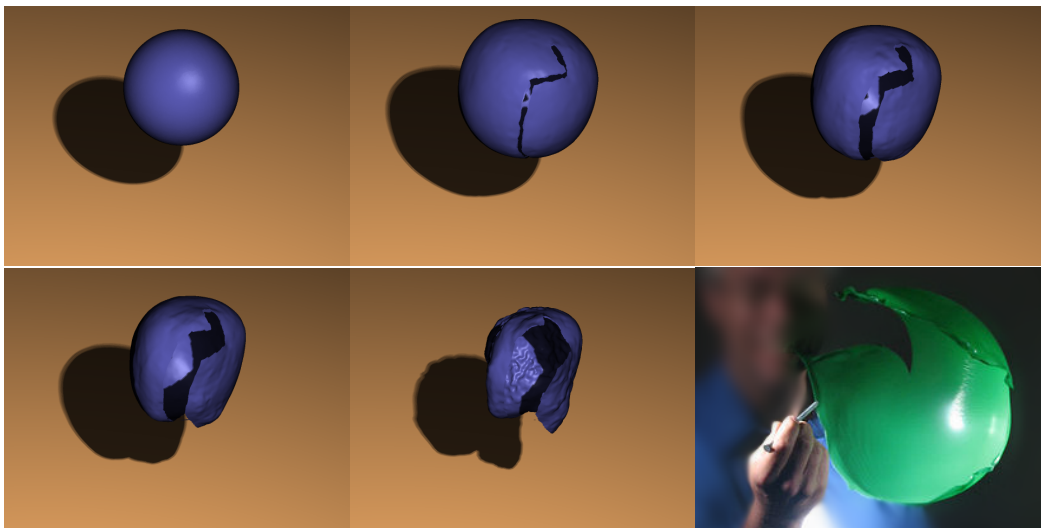


Figure 7.10: A balloon tears when inflated too much. Once the tear is opened, the shell can relax and the tearing process stops. Bottom right: a picture taken with a high-speed camera.

occur in regions of highly irregular sampling, where the heuristic described in Section 7.2.1 breaks down.

This dependence on the sampling pattern is the drawback that is incurred by abandoning a consistent connectivity. An alternative simulation method based on global parameterization of the object [96] avoids these issues. However, since the parameterization is not isometric in general, the material behavior is then dependent on the parameterization chosen.

Under very large stretching deformations, the initial sampling degrades. Hence, simulation quality suffers after applying large plastic deformations. This could be alleviated by dynamically resampling the surface in areas that were stretched. While resampling and activating new nodes on the surface is easy, transferring the plastic deformation onto the newly created fibers without creating temporal discontinuities is challenging.

As demonstrated in the previous chapters, using point samples without connectivity is a viable approach for surface modeling. Reintroducing persistent, but not necessarily consistent connectivity makes it possible to animate point-sampled surfaces as thin shells. Naturally, the question of how sampling approaches would perform for volumetric data arises. Such approaches have been studied in the context of rendering and data representation [88, 230]. One of the preferred methods for fluid animation is also a particle-based approach. An extension to the popular *smoothed particle hydrodynamics* approach for fluid simulation shall be treated in the next chapter.

Chapter 8

Visco-Elastic Fluid Simulation

The behavior of fluids is governed by the Navier-Stokes equations. Most commonly, the incompressible version of these equations is discretized on either regular or tetrahedral meshes for both water and smoke simulations [70, 118, 134, 190].

An alternative to these Eulerian approaches is to discretize the material instead of its embedding space. Using a tetrahedral or hexahedral grid, this is the standard method for simulations involving continuum elasticity (see also Chapter 9). However, fluid simulations in computer graphics involve free surfaces and complex topological changes. Hence, a grid-based Lagrangian discretization is not feasible, as the grid would have to be restructured in every time step. Instead, Lagrangian fluid simulations are particle-based. Similar to the connectivity-free sampling used for point-sampled surfaces, particles without persistent connectivity are used to sample the fluid volume. The velocity field is sampled at the particle locations, and differential operators capable of handling irregular samples are used to evaluate the Navier-Stokes equations.

The following sections first introduce *smoothed particle hydrodynamics* (SPH), the most popular particle method for fluid simulation. Then, an extension to SPH is proposed that adds elastic forces to the fluid simulations without requiring additional connectivity information. Thus, elastic objects, fluids, as well as phase transitions between these states can be simulated. The goal is to offer a truly connectivity-free method that can compute a believable approximation to the behavior of visco-elastic materials.

8.1 Smoothed Particle Hydrodynamics

At the core of smoothed particle hydrodynamics is a method for function approximation from scattered samples [81, 137]. Originally invented for astrophysical

simulations, the method has been applied to a variety of problems [145]. A good introduction to the method can also be found in [132]. It has been introduced to computer graphics for the simulation of soft deformable objects [64], and has since mainly been used for fluid simulations [112, 146, 228]. As the particles carry the mass of the fluid, boundary conditions can be formulated on the basis of individual particles, making interaction between the fluid and its surrounding simple and intuitive [152].

In the following, the framework for function approximation used by SPH is discussed, before treating the specifics of fluid simulation using the SPH framework.

8.1.1 Kernel Functions

In SPH, a number of particles represents the material. Each particle carries mass, velocity and other attributes.

A kernel function $W_h(\cdot)$ describes the influence of each particle on its surroundings. The kernels share some characteristics with the weight functions for surface reconstruction discussed in Section 3.1.5. They are defined as smooth, radially symmetric functions from $\mathbb{R}^3 \to \mathbb{R}$, mapping a distance vector to a corresponding weight. SPH kernels are required to be normalized, i. e.

$$\int W_h(\mathbf{x})\mathrm{d}\mathbf{x} = 1. \tag{8.1}$$

As the kernel functions are radially symmetric, they can also be written as $W_h(\mathbf{x}) = w_h(\|\mathbf{x}\|)$. The normalization criterion is then modified to $\int 2\pi r w_h(r)\mathrm{d}r = 1$ for two-dimensional kernels, and $\int 4\pi r^2 w_h(r)\mathrm{d}r = 1$ in three dimensions.

The *smoothing length* h is a constant of the particle determining the maximum influence radius of a particle. For $h \to 0$, the limit of any SPH kernel should be a Dirac δ-function. In computer graphics, the smoothing length is usually assumed to be constant over all particles, and does not change during the simulation, yielding equal-sized particles that are much easier to handle algorithmically.

Gingold and Monaghan initially proposed normalized Gaussian kernels [81], while Lucy proposed to use polynomial splines [137]. Polynomial spline kernels, while of lower smoothness, have local support and their derivatives vanish at the boundary of the support region. Additionally, a well-chosen polynomial kernel is computationally cheaper. Good polynomial kernels for 2D and 3D simulation, as well as their derivatives, are given in Appendix C. Figure 8.1 shows a typical kernel function.

If the kernels have local support, acceleration data structures can be used to quickly find neighboring particles that are of interest at a specific location. Since the smoothing radius h is usually constant throughout space, a hash grid is ideally suited for SPH simulations. The cell spacing should be chosen to optimize range queries of radius h as described in Section 3.4.

Often, several kernel functions are used for different interpolation tasks within the same simulation [56, 146].

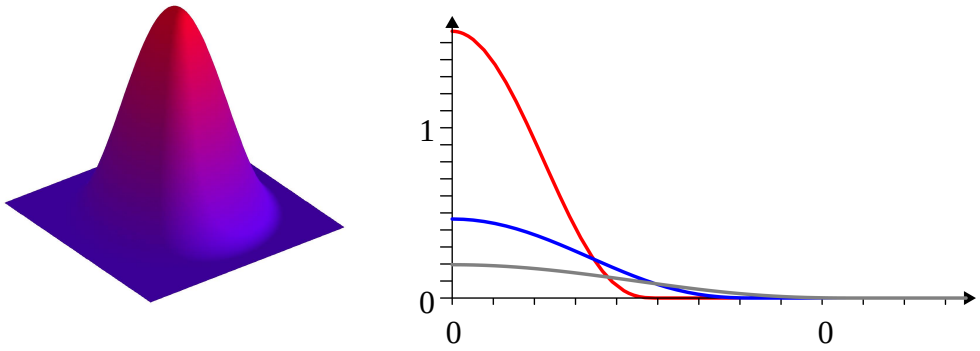


Figure 8.1: (a) A polynomial 3D kernel function $W_h(\mathbf{x})$ (see Appendix C for a definition). (b) The corresponding function $w_h(\|\mathbf{x}\|)$ for several values of $h = 1$ (red), $h = \frac{3}{2}$ (blue), and $h = 2$ (gray).

8.1.2 Function Approximation using SPH

Using a kernel function as defined above, any continuous function $f(\mathbf{x})$ can be smoothed by convolution with W_h to obtain $\tilde{f}(\mathbf{x})$:

$$\tilde{f}(\mathbf{x}) = \int W_h(\mathbf{x}-\mathbf{y}')\, f(\mathbf{y}')\mathrm{d}\mathbf{y}'. \tag{8.2}$$

In the discrete setting, only samples of $f(\mathbf{x})$ are available at discrete particle locations: $f_i = f(\mathbf{x}_i)$. Each particle represents a small volume fraction V_i. Thus the integral in (8.2) can be approximated by a sum over the particles:

$$\tilde{f}(\mathbf{x}) \approx \langle f(\mathbf{x})\rangle = \sum_i W_h(\mathbf{x}-\mathbf{x}_i)\, f_i V_i \tag{8.3}$$

We will call $\langle f(\mathbf{x})\rangle$ the SPH approximation of the function $f(\mathbf{x})$.

Every particle carries a fixed mass m_i, but since we make no assumption on the distribution of the particles, the density at each sample point, ρ_i, and thus the particle volume $V_i = m_i/\rho_i$ may vary. However, we can estimate the density using the SPH approximation of the unknown density function $\rho(\mathbf{x})$:

$$\langle \rho(\mathbf{x})\rangle = \sum_i W_h(\mathbf{x}-\mathbf{x}_i)\rho_i V_i = \sum_i W_h(\mathbf{x}-\mathbf{x}_i) m_i \tag{8.4}$$

We can then define $\rho_i := \langle \rho(\mathbf{x}_i)\rangle$. Now that both m_i and ρ_i are known, (8.3) can be used to interpolate any function from samples given at the particle positions.

In a simulation, SPH approximations are mostly evaluated at particle positions $\mathbf{x}_i$. We will therefore introduce the following short-hand notation. For the SPH approximation of the function f, evaluated at a point $\mathbf{x}_i$, we write

$$\langle f\rangle_i := \langle f(\mathbf{x}_i)\rangle. \tag{8.5}$$

For the kernel weight of a particle at $\mathbf{x}_j$ with respect to $\mathbf{x}_i$, we introduce the shorthand

$$W_{ij} = W_{ji} := W_h(\mathbf{x}_i - \mathbf{x}_j), \tag{8.6}$$

assuming the smoothing length h to be constant throughout the simulation.

8.1.3 Approximations of Differential Operators

To obtain approximations of differential operators, they are applied to the approximation $\langle f(\mathbf{x})\rangle$. Linear operators can thus be reduced to the operations on the kernel functions. For instance, a simple approximation to the gradient of $f(\mathbf{x})$ is

$$\langle \nabla f(\mathbf{x})\rangle = \sum_i \nabla W_h(\mathbf{x} - \mathbf{x}_i) f_i V_i, \tag{8.7}$$

where the gradient $\nabla W_h(\mathbf{x} - \mathbf{x}_i)$ can be written in terms of the derivative of the one-dimensional kernel function w_h:

$$\nabla W_h(\mathbf{x} - \mathbf{x}_i) = \frac{\mathbf{x} - \mathbf{x}_i}{\|\mathbf{x} - \mathbf{x}_i\|} w_h'(\|\mathbf{x} - \mathbf{x}_i\|). \tag{8.8}$$

The naïve gradient operator (8.7) is quite sensitive to the distribution of particles. If the distribution is not symmetric, (8.7) can yield non-zero gradients even if the samples f_i are all equal. In case the gradient is evaluated at a sample point $\mathbf{x}_i$, a better approximation is available. Since the gradient of any function remains unchanged if we subtract a constant, we can rewrite the gradient approximation at the sample point $\mathbf{x}_i$ by using the SPH approximation of $\nabla(f - f_i)$:

$$\langle \nabla f\rangle_i := \langle \nabla(f - f_i)\rangle_i = \sum_j \nabla W_{ij}(f_j - f_i) V_j. \tag{8.9}$$

Eq. 8.9 correctly produces zero gradients for constant functions. There are different methods to derive the above result, for a more general derivation, see [145]. A similar argument could be used at arbitrary evaluation points. However, evaluating $\langle \nabla(f - \langle f\rangle)\rangle$ does not yield a significantly more stable gradient estimate between sample points.

Similarly to (8.8), a Laplace operator for f and a divergence operator for a vector-valued function $\mathbf{f}$ can be defined:

$$\langle \nabla^2 f(x)\rangle = \sum_i \nabla^2 W_h(\mathbf{x} - \mathbf{x}_j) f_i V_i \tag{8.10}$$

$$\langle \nabla * \mathbf{f}(\mathbf{x})\rangle = \sum_i \nabla W_h(\mathbf{x} - \mathbf{x}_j) * \mathbf{f}_i V_i. \tag{8.11}$$

For evaluation at the sample points, we can apply the same reasoning as above and obtain

$$\langle \nabla^2 f\rangle_i = \sum_j \nabla^2 W_{ij}(\mathbf{f}_j - \mathbf{f}_i) V_j, \tag{8.12}$$

$$\langle \nabla * \mathbf{f}\rangle_i = \sum_j \nabla W_{ij} * (\mathbf{f}_j - \mathbf{f}_i) V_j. \tag{8.13}$$

The Laplacian of the weight function is given by $\nabla^2 W_h(\mathbf{x}) = w_h''(\mathbf{x})$.

In general, mathematical identities known from vector calculus do not carry over to SPH approximations. It is hence worth carefully selecting the right function to approximate, as results might be radically different, especially for low sampling densities and irregular sampling.

8.1.4 Fluid Simulation using SPH

Fluid behavior is governed by the Navier-Stokes equations, derived from the momentum and mass conservation laws. For compressible fluids, the momentum equation can be written as

$$\frac{D\mathbf{v}}{Dt} = \frac{1}{\rho}\left(-\nabla p + \mathbf{F}_{\text{viscous}} + \mathbf{F}_{\text{ext}}\right), \tag{8.14}$$

where $\frac{D\mathbf{v}}{Dt} = \frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} * \nabla \mathbf{v}$ is the *material derivative* of $\mathbf{v}$, i. e. the derivative of $\mathbf{v}$ at a point moving with the material. In a Lagrangian setting, the particles move with the material they represent. Thus, the material derivative can simply be computed by looking at the time derivative of a value stored with a given particle. p denotes the pressure field, $\mathbf{F}_{\text{ext}}$ are external forces acting on the fluid, and $\mathbf{F}_{\text{viscous}}$ are forces due to internal friction.

The continuity equation, stating the conservation of mass, is not needed in the Lagrangian framework, since the mass is carried by the particles, and it hence preserved automatically if no particles are deleted or inserted, and the particle masses are not changed.

The pressure p is a function of the density of the fluid. A common choice for the pressure function is [144]

$$p = K\left(\left(\frac{\rho}{\rho_0}\right)^{\Gamma} - 1\right). \tag{8.15}$$

The parameter K is a stiffness constant. Monaghan proposed $\Gamma = 7$, whereas in computer graphics, a value of $\Gamma = 1$ is typically used [64, 146]. Low values of Γ and K make the fluid more compressible, but allow for larger time steps. For simulations involving free boundaries, the pressure can be clamped to positive values, or reduced by a factor ζ for negative values:

$$p' = \begin{cases} p & p \geq 0, \\ \zeta p & p < 0. \end{cases} \tag{8.16}$$

Clamping or reducing the pressure at the boundaries reduces the cohesive forces inside the fluid and makes splashes and spray more likely.

Directly computing the SPH approximation of the pressure force $-\nabla p$ yields non-symmetric forces that violate the conservation of linear and angular momentum, leading to ghost forces during the simulation. Instead, we can approxi-

mate the acceleration due to pressure forces directly and obtain a symmetric and momentum-preserving expression at the particle locations:

$$\begin{aligned}\left[\frac{-\nabla p}{\rho}\right](\mathbf{x}_i) &= -\left[\nabla\left(\frac{p}{\rho}\right)\right](\mathbf{x}_i) - \frac{p_i}{\rho_i^2}\nabla\rho(\mathbf{x}_i) \\ &\approx -\left\langle\nabla\left(\frac{p}{\rho}\right)\right\rangle_i - \frac{p_i}{\rho_i^2}\langle\nabla\rho\rangle_i \\ &= -\sum_j \nabla W_{ij}\left(\frac{p_j}{\rho_j^2} + \frac{p_i}{\rho_i^2}\right) m_j. \end{aligned} \tag{8.17}$$

In computer graphics, most fluid simulations deal with (almost) inviscid liquids like water, or gases. In SPH simulations, viscosity is mostly an unwanted side-effect of the simulation method. If it is modeled at all, the bulk viscosity, measuring viscous forces due to compression, is ignored [146]. The viscosity forces then reduce to the viscosity term for a Newtonian, incompressible fluid:

$$\mathbf{F}_{\text{viscous}} = \mu\nabla^2\mathbf{v} \approx \mu\left\langle\nabla^2\mathbf{v}\right\rangle, \tag{8.18}$$

where μ is the shear viscosity coefficient. Moderate viscosity regularizes the velocities of neighboring particles, and has a stabilizing effect. Simulations with very high or very low viscosity values require smaller time steps. Usually, viscosity is only needed for numerical stability, and should be set as low as possible. Methods for artificial viscosity make this easier. One example is the XSPH technique [142]. To simulate the effect of viscosity, the velocities are smoothed in each time step:

$$\tilde{\mathbf{v}}_i = (1-\xi)\mathbf{v}_i + \xi\langle\mathbf{v}\rangle_i. \tag{8.19}$$

The parameter $0 \leq \xi \leq 1$ determines the amount of artificial viscosity. XSPH uses the smoothed velocities $\tilde{\mathbf{v}}_i$ only for advection. If $\tilde{\mathbf{v}}_i$ is also stored with the particles and used in all subsequent computations, the viscosity is higher. The advantage of artificial viscosity is that the parameter ξ can be set as high as necessary: high artificial viscosity does not require a smaller time step.

Using the forces computed as above, velocity Verlet integration [207] can be used to animate the fluid.

8.2 Elastic Forces

An SPH fluid simulation requires only the current spatial neighborhoods for its computations. Methods that add elastic forces to SPH simulations require persistent connectivity information to evaluate elastic stresses [112]. In this section, an alternative approach is presented that uses only the current simulation state to compute elastic forces.

Inspired by crystallography, the material is initially sampled regularly, using a closest sphere packing — a hexagonal grid in two dimensions, and a cubic closest packed lattice in three dimensions. Since the rest state that is necessary for elasticity computations is not stored, it has to be reconstructed from the current simulation state alone. To this end, shape matching is used to locally fit a lattice to the current neighbors of each particle. The orientation and scale of the local lattice is determined using shape matching similar to [148]. Hence, the spatial neighborhood relationships between particles in the current simulation state implicitly define a connectivity that is used for computation of elastic forces.

The absence of a simulation mesh or rest state simplifies the simulation of melting and freezing processes. The neighborhood information computed during the simulation can be also used to provide a simple penalty-based collision handling scheme at no additional computational cost.

8.2.1 Implicit Rest State

In a crystal lattice, the rest state of the neighbors of any atom is determined by the orientation of the lattice and the lattice type. Since we choose the initial sampling to be a closest sphere packing, the lattice type is known, and only the orientation of the lattice has to be determined. An important property of a closest sphere packing is that it is a stable state of particle-based fluid simulations, such that there is no conflict between the fluid simulation and the elastic forces.

In 2D, the only closest sphere packing is a regular hexagonal grid. In three dimensions, there are two possible closest sphere packings: hexagonally closest packed (HCP) and cubic closest packed (CCP) [224]. As initial sampling, we sample the material using CCP, as the set of neighbors to each point has higher symmetry. A CCP lattice around the origin consists of the points

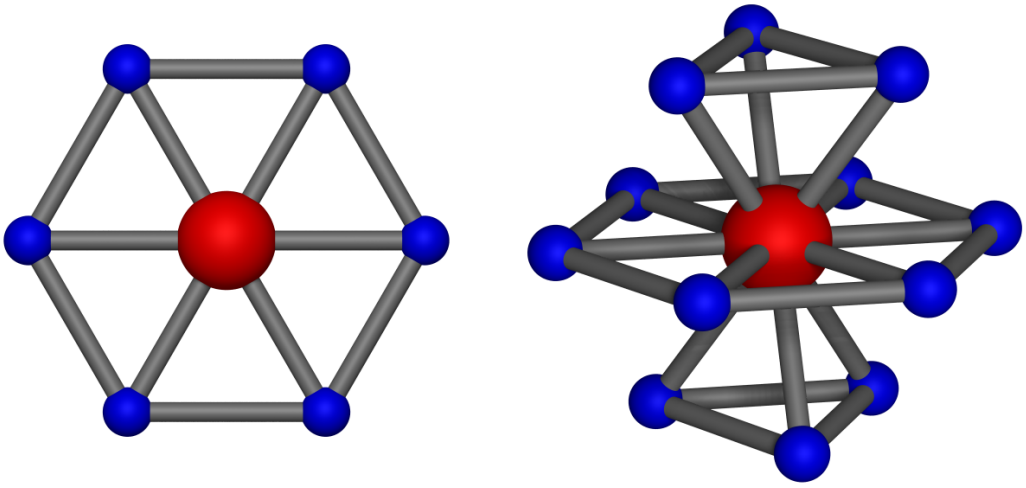
$$L = \left\{ d \left[i\mathbf{e}_1 + j\left(\frac{1}{2}\mathbf{e}_1 + \sqrt{\frac{3}{4}}\mathbf{e}_2\right) + k\left(\frac{1}{2}\mathbf{e}_1 + \sqrt{\frac{1}{12}}\mathbf{e}_2 + \sqrt{\frac{2}{3}}\mathbf{e}_3\right) \right] \right\}, \tag{8.20}$$

with integers $i, j, k \in \mathbb{Z}$. The vectors $\mathbf{e}_{1,2,3}$ form an orthonormal basis of $\mathbb{R}^3$. d denotes the inter-particle distance in the lattice. Figure 8.2 shows the nearest neighbors to a lattice point in 2D and 3D.

Computing the Rest State

In each simulation step, the orientation and deformation of the lattice has to be reconstructed from the current particle positions. This is done for each particle separately. The procedure computes a shape matching of the neighborhood particles with the lattice points. See Figure 8.3 for an illustration.

In a first step, lattice points are assigned to particles in the neighborhood. Then, a linear transformation $\mathbf{A}$ is computed such that the transformed grid best matches the particle locations.



(a) (b)

Figure 8.2: Closest lattice points to a particle. (a) 2D lattice. (b) 3D lattice. Shown are the center particle (red) as well as the lattice positions of its immediate neighbors (blue).

The grid transformation $\mathbf{A}^{(t-1)}$ from the last time step is used to assign lattice points to particles. For each particle $\mathbf{x}_j$ in the neighborhood of $\mathbf{x}_i$, we assign the lattice point $\mathbf{l}_{ij}$ which is closest to $\mathbf{r}_{ij} = (\mathbf{x}_j - \mathbf{x}_i)$ in the untransformed lattice.

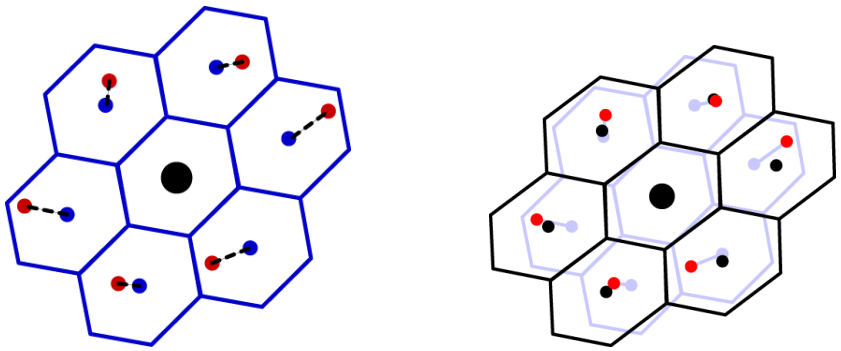
$$\mathbf{l}_{ij} = \arg\min_{\mathbf{l} \in L} \left\| \mathbf{l} - \left(\mathbf{A}_i^{(t-1)}\right)^{-1} \mathbf{r}_{ij} \right\|^2 . \tag{8.21}$$

Due to the structure of the grid, this minimization can be easily solved by enumerating the nearest points in L. Note that it is possible that several neighboring particles are assigned to the same lattice point. In such a case, only the closest particle is assigned to the lattice point. The forces generated by the neighborhood of the particle i will not be applied to the free particle.

Erroneous assignment mainly happens due to two reasons: When the material is deformed rapidly, the lattice structure in the last time step is not a good approximation to the lattice structure in the current time step, and the “true” assignment of particles to lattice points might be lost. Under large compressive deformations, the lattice positions lie very close to each other, such that even small disturbances in the particle positions can cause neighboring particles to be assigned to the same lattice position.

After all points are assigned, the transformation $\mathbf{A}$ is computed such that the transformed grid best matches the actual particle positions in a least squares sense:

$$\mathbf{A}_i = \arg\min_{\mathbf{A} \in \mathbb{R}^{3\times 3}} \sum_j W_{ij} \|\mathbf{A}^{-1}\mathbf{r}_{ij} - \mathbf{l}_{ij}\|^2 . \tag{8.22}$$



(a) (b)

Figure 8.3: Assigning lattice points and shape matching. (a) Neighboring particles (red) are assigned to lattice points (blue) using the local grid transformation $\mathbf{A}_i^{(t-1)}$. (b) Shape matching: $\mathbf{A}_i$ is computed such that the deformed lattice points (black) best match the assigned particle positions (red).

As pointed out in [148], the solution to this minimization is

$$\mathbf{A}_i = \left(\sum_j W_{ij} \mathbf{r}_{ij} \mathbf{l}_{ij}^T \right) \left(\sum_j W_{ij} \mathbf{l}_{ij} \mathbf{l}_{ij}^T \right)^{-1}. \tag{8.23}$$

We then use a singular value decomposition to compute the rotational part of the lattice transformation $\mathbf{A}_i = \mathbf{U}_i \Sigma_i \mathbf{V}_i^T$:

$$\mathbf{R}_i = \mathbf{U}_i \mathbf{V}_i^T. \tag{8.24}$$

The rotation $\mathbf{R}_i$ gives the rigid transformation of the lattice around the particle i, the translational part of the transformation is already accounted for by the movement of the particle itself. The rest position $\mathbf{g}_{ij}$ of a particle at $\mathbf{x}_j$ relative to $\mathbf{x}_i$ is

$$\mathbf{g}_{ij} = \mathbf{R}_i \mathbf{l}_{ij}. \tag{8.25}$$

The singular values Σ_i give additional information about the deformation of the lattice.

We can use the implicit rest state of surrounding particles in two ways. The next section details how a general strain tensor can be computed using the rest positions $\mathbf{g}_{ij}$. Section 8.2.3 presents a method to compute approximate forces. The latter method is more robust against bad particle distributions, however, physical accuracy is lost in the approximation.

8.2.2 Computing Strain

Having computed rest states for the neighboring particles j, we can compute an estimate for the strain tensor at a particle i. The linear (Cauchy) strain tensor is defined as

$$\varepsilon = \frac{1}{2}\left(\nabla \mathbf{u} + (\nabla \mathbf{u})^T\right), \tag{8.26}$$

where $\mathbf{u} = [u_x, u_y, u_z]^T$ is the displacement of the material. We discretize the partial derivatives of the displacement using one-sided differences. For the β component of the gradient of u_α at $\mathbf{x}_i$, a particle $\mathbf{x}_j$ contributes

$$\left(\frac{\partial u_\alpha}{\partial \beta}\right)_{ij} = \frac{(\mathbf{r}_{ij} - \mathbf{g}_{ij}) * \mathbf{e}_\alpha}{\mathbf{g}_{ij} * \mathbf{e}_\beta}, \tag{8.27}$$

for $\alpha = x,y,z$. The orthonormal vectors $\mathbf{e}_{x,y,z}$ form a basis of $\mathbb{R}^3$. As all particles in the neighborhood of particle i influence the strain state at $\mathbf{x}_i$, we weight the contributions of the particles:

$$\left(\frac{\partial u_\alpha}{\partial \beta}\right)_i = \frac{\sum_j \left[W_{ij}(\mathbf{g}_{ij} * \mathbf{e}_\beta)\left(\frac{\partial u_\alpha}{\partial \beta}\right)_{ij}\right]}{\sum_j \left[W_{ij}(\mathbf{g}_{ij} * \mathbf{e}_\beta)\right]} = \frac{\sum_j \left[W_{ij}(\mathbf{r}_{ij} - \mathbf{g}_{ij}) * \mathbf{e}_\alpha\right]}{\sum_j \left[W_{ij}(\mathbf{g}_{ij} * \mathbf{e}_\beta)\right]} \tag{8.28}$$

Thus, the strain can be computed from the knowledge of rest state and current particle positions. Note that in a regular setting, (8.28) yields central differences. The strain can be used to apply any standard elasticity model.

8.2.3 Direct Force Estimate

The strain estimate is only reliable if a relatively large number of neighbors is available. When only few neighbors contribute to the strain estimate, the results strongly depend on the particle distribution.

An alternative to computing the strain in order to obtain forces is to use purely geometric reasoning to derive elastic forces. The basic idea is that each neighborhood i compels particles j to move towards their assigned rest state positions $\mathbf{g}_{ij}$. This corresponds to the behavior of a material with a Poisson ratio of zero. In order to include local volume preservation forces into the framework, we use a method similar to [148]. We make use of the additional information contained in the singular values of the lattice transformation $\mathbf{A}_i$. We allow some deformation of the lattice, while keeping the volume of the lattice constant. This yields goal positions

$$\mathbf{g}'_{ij} = \mathbf{U}_i \mathbf{S} \mathbf{V}_i^T \mathbf{l}_{ij}. \tag{8.29}$$

Here, the volume-preserving scale matrix

$$\mathbf{S} = \frac{s\Sigma_i + (1-s)\mathbf{I}}{\sqrt[3]{\det(s\Sigma_i + (1-s)\mathbf{I})}} \tag{8.30}$$

deforms the lattice half-way from the identity $\mathbf{I}$ to its current best match. The normalization in the denominator ensures that volume is preserved by the transformation. The parameter s controls how much deformation is allowed when computing goal positions. Once the goal positions are computed, we obtain forces that pull each particle to its assigned goal position:

$$\mathbf{F}_{ij} = W_{ij} K_e (\mathbf{g}'_{ij} - \mathbf{r}_{ij}), \tag{8.31}$$

where K_e is a stiffness constant.

The elastic forces are not necessarily symmetric, i. e. $\mathbf{F}_{ij} \neq \mathbf{F}_{ji}$, and do not preserve linear and angular momentum. In order to preserve linear momentum, $\frac{1}{2}\mathbf{F}_{ij}$ is applied to particle j and $-\frac{1}{2}\mathbf{F}_{ij}$ to i.

Also the torque introduced by the elastic forces has to be compensated for. The spurious torque around the center of mass of the neighborhood $\mathbf{c}_i$ can be measured as

$$\tau_i = \frac{1}{2} \sum_j (\mathbf{x}_j - \mathbf{c}_i) \times \mathbf{F}_{ij} - (\mathbf{x}_i - \mathbf{c}_i) \times \mathbf{F}_{ij}. \tag{8.32}$$

As this torque would violate the preservation of angular momentum, we redistribute it onto the $\mathbf{x}_j$ by adding a torque correction force to $\mathbf{x}_j$:

$$\mathbf{F}^{\tau}_{ij} = \frac{W_{ij}}{\sum_j W_{ij} \|\mathbf{x}_j - \mathbf{c}_i\|} \tau_i \times \frac{\mathbf{x}_j - \mathbf{c}_i}{\|\mathbf{x}_j - \mathbf{c}_i\|}. \tag{8.33}$$

Thus, the total torque incurred by the elastic forces is zero, as required.

8.2.4 Phase Transitions

The only difference between elastic solids and fluids in our framework is the fact that particles in a solid are subject to elastic restoring forces while particles in the fluid phase are not. Thus, phase transitions can be implemented by activating or deactivating the elastic restoring forces described in the previous sections.

As both fluids and elastic solids can be handled within our framework, phase transitions can be implemented without transferring material between representations. Since no connectivity or explicit rest state is stored, this information does not have to be generated in case of freezing. Instead, the fluid particles crystallize when freezing. Note that the phase transition model presented here does in no way aim for a correct simulation of the physical process of phase transitions. Instead, melting and freezing are modeled as stochastic processes: A Poisson process gives the probability ω that a particle changes melts or freezes within the current time step:

$$\omega = 1 - e^{-\lambda \Delta t}. \tag{8.34}$$

The transition rate λ can be determined from criteria like position in space, structure of the neighborhood, or physical properties like temperature. For temperature-based melting or freezing, heat conduction can be simulated using the SPH framework [145].

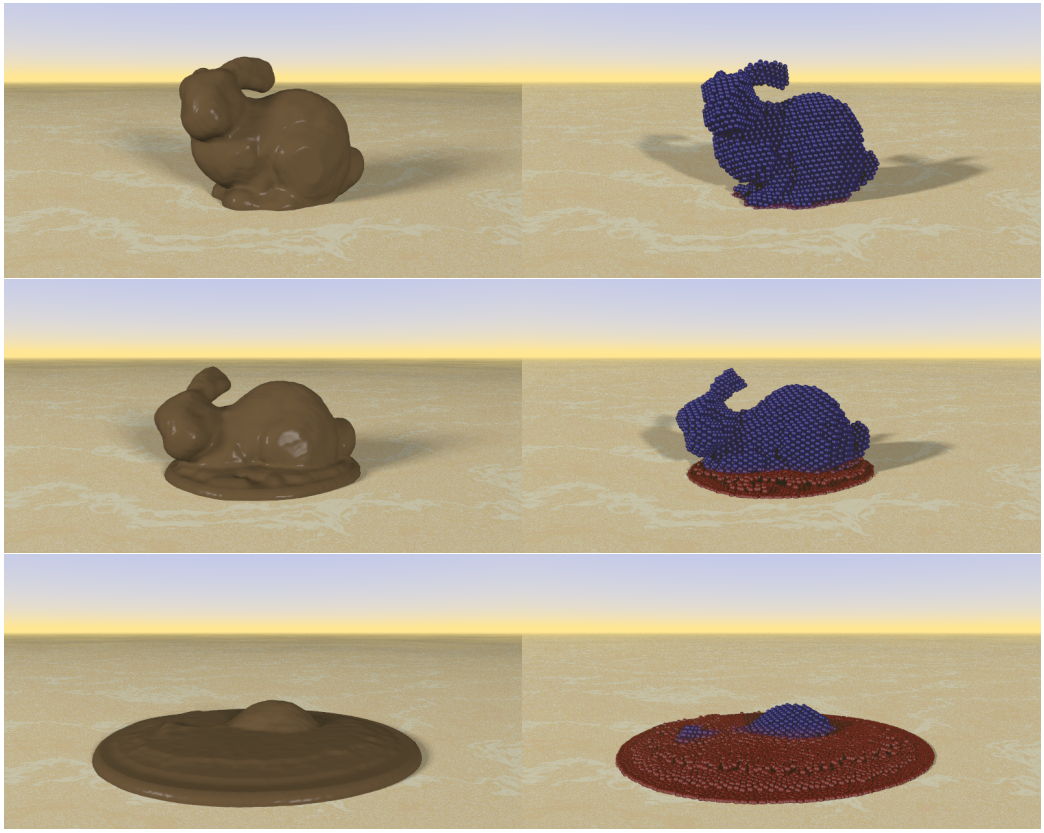


Figure 8.4: Frames from an animation involving a solid to fluid phase transition. The bunny drops on the floor where it deforms and melts. The particle rendering on the right shows the temperature distribution.

Melting

To simulate melting, the *melting rate* λ_m for each particle is computed from the temperature:

$$\lambda_m = \max(0, \lambda_m^0(T - T_m)), \tag{8.35}$$

where T is the temperature of a particle and T_m is the melting point of the material. Figure 8.4 shows frames from an animation that involves melting.

Freezing

The inverse process of freezing is modeled similarly. If a fluid particle is close to a solid, it has a probability to freeze and thus integrate into the lattice of the solid. This probability is dependent on the relative velocity of the particle to its solid neighbors, and the distance of the fluid particle to the next available lattice point, as well as the temperature. This statistical approach mimics the freezing process without computing a full-blown physical simulation [116, 117].

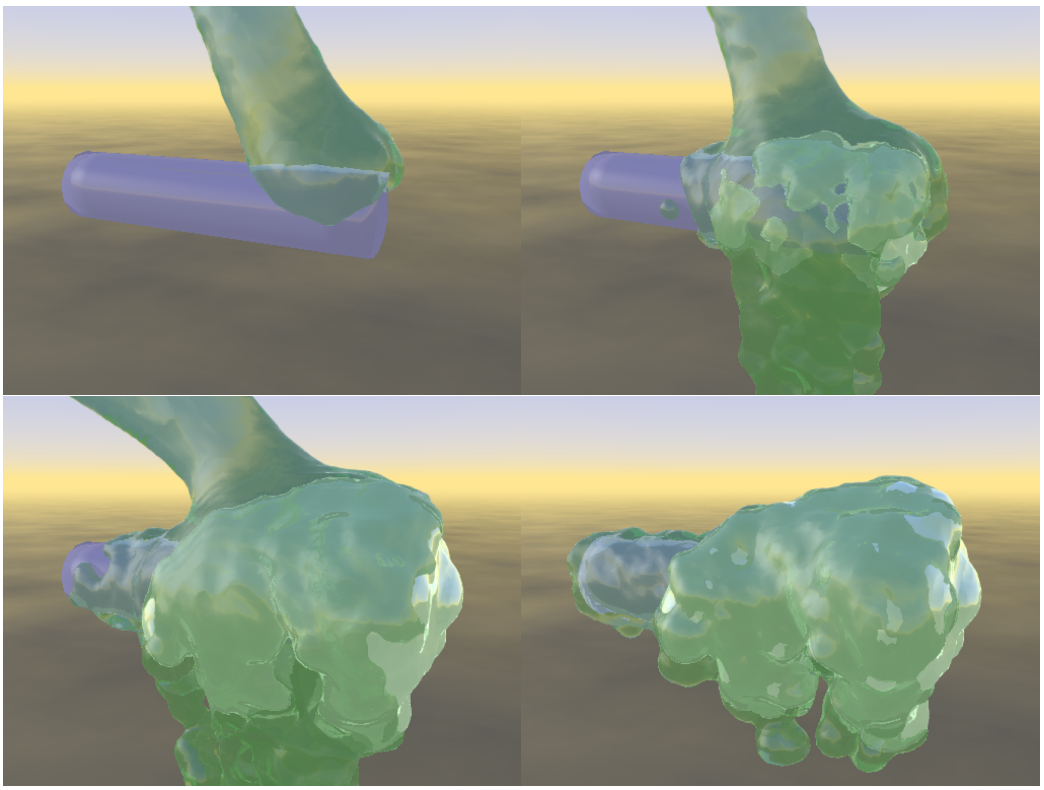


Figure 8.5: Frames from an animation involving a fluid to solid phase transition. The green slime is a viscous fluid simulated using SPH with artificial viscosity. “Ice” forms on the rod as fluid particles solidify and are subjected to elastic forces.

To compute the *freezing rate* λ_f for a particle i with neighbors j, we use a criterion based on temperature T, number of solid neighbors n_s, their average velocity $\bar{\mathbf{v}}$, and the accumulated lattice forces:

$$\lambda_f = \max\left(0, \lambda_f^0 (T_f - T)(n_s - n_{\min}) \min(1, c_{\mathbf{v}}) \min(1, c_{\mathbf{F}})\right) \tag{8.36}$$

Here, T_f is the freezing temperature of the fluid and $n_{\min}$ denotes the minimum number of solid neighbors. If this parameter is set to zero, particles can freeze spontaneously, otherwise, particles can only freeze to already solid material. The velocity regularization term $c_{\mathbf{v}} = v_{max}/\|\mathbf{v}_i - \bar{\mathbf{v}}\|$ decreases the freezing probability when the velocity of the particle differs significantly from the velocity of its neighborhood. The last term $c_{\mathbf{F}} = F_{max}/\|\sum_j \mathbf{F}_{ij}\|$ diminishes the freezing rate of a particle if the (hypothetical) lattice forces exceed a given maximum. This is a simple measure of how good the current position of the particle fits into the lattice of a neighboring solid.

Figure 8.5 shows an animation involving freezing. The fluid is cooled when it comes in contact with the rod, and freezes.

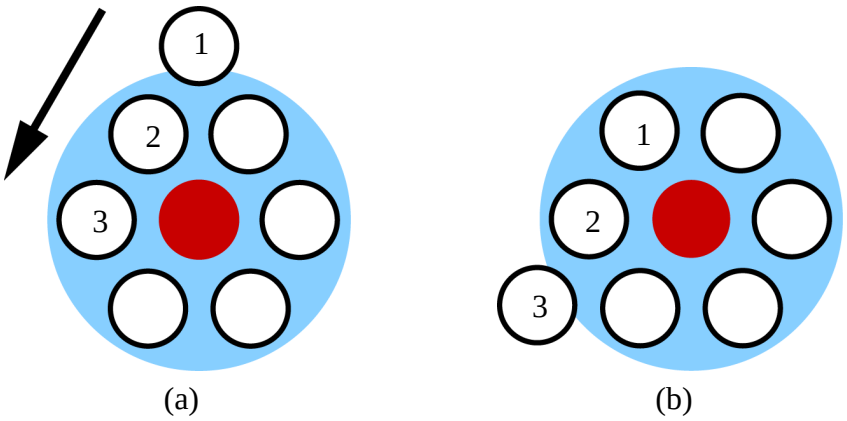


Figure 8.6: Inherent plasticity: Since we do not store connectivity, the center particle has no way of distinguishing between the situations (a) and (b). Its neighborhood is limited to the blue region. There will be no restoring forces for particles 1, 2 and 3. Particle 1 is integrated into the neighborhood, while particle 3 leaves. These deformations are plastic.

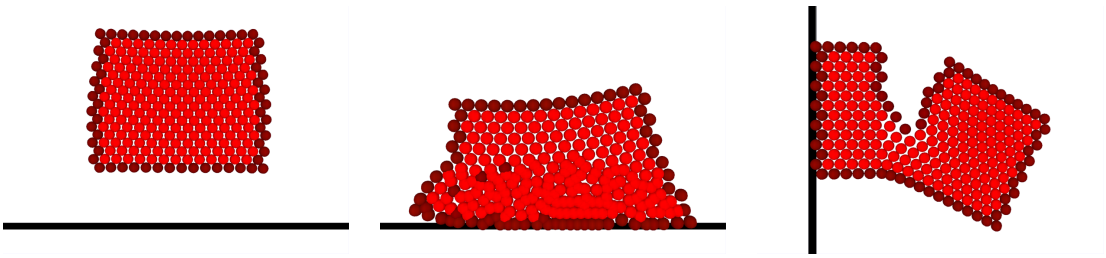
8.2.5 Material Properties

The materials that can be modeled using the enhanced SPH method range from very stiff elastic materials to fluids. However, abandoning stored connectivity gives rise to some inherent traits that apply to all possible simulations using this method.

Plasticity

Since no rest state information is stored, the rest state of the material has to be inferred from the current state. Thus, all information on the rest state is contained in the positions of the current neighbors of a particle. The particle does not store which particles were its initial neighbors, and hence the algorithm has no way of distinguishing particles. Even if the positions of neighboring particles stay the same, the particles occupying these position may have changed. In that case, there are no restoring forces for the original particles. Instead, they are integrated into their new neighborhoods. Figure 8.6 shows an illustration.

The shape matching process assumes that the local lattice deformation does not change abruptly. This means that the material cannot be deformed too quickly without acting plastic: Consider a pair of neighboring particles i and j. A permanent plastic deformation occurs whenever j is displaced far enough such that different lattice point is closest to its current position in the next time step. As the material has no memory and does not know the “true” rest state of the particle j with respect to i, these changes are not counteracted by restoring forces.



(a) (b) (c)

Figure 8.7: Particles in 2D simulations of different material properties. (a) An elastic cube bounces off the ground plane. (b) Rest state of a plastic cube after falling. (c) Fracture. The cube is fixed to the Wall and fractures under the influence of gravity. Darker particles have a docking rate $\lambda_d = 0$ for some of their lattice points. They form the boundary of the solid.

The inherent plasticity due to particle ambiguity acts independently of any plasticity computations that are used in the simulation, for example using stored plastic strain. Figure 8.7 (b) shows plastic deformation in a 2D simulation.

Fracture

If a particle leaves a neighborhood of an adjacent particle due to large deformations, it is not considered for shape matching any longer, and no restoring forces are generated — the material has fractured. However, without further processing, the crack is closed as soon as the stray particle (or another particle) re-enters the neighborhood. This causes the material to behave sticky.

To avoid this, each particle remembers which of its lattice points are occupied by other particles. We use a probabilistic model to account for faults and weaknesses in the material. Similar to the probabilistic model for melting and freezing, each particle stores a *docking rate* for each lattice point q. If the shape matching assigns a particle to q, the probability that it is accepted and forces are computed is given by (8.34), using the docking rate λ_d^q. In every time step that a lattice point q is not occupied by a particle, its docking rate is reduced by $\Delta\lambda_d^-$. Conversely, the docking rate of an occupied lattice point is increased by $\Delta\lambda_d^+$ in each time step. A stress criterion can be used to additionally modify λ_d. Figure 8.7 (c) shows fracturing in a 2D simulation.

The parameters $\Delta\lambda_d^{+/-}$ determine how easy it is for the material to close once opened cracks. High values of $\Delta\lambda_d$ create a brittle material, while lower values allow for ductile fracture.

8.2.6 Multiple Objects

Starting from a fluid simulation, we have so far only considered one object. If elastic forces are considered, several distinct objects are possible. The algorithm can be easily extended to handle multiple objects. Each particle carries an object ID, and the shape matching as well as the force computation are confined to particles with the same object ID.

If forces acting between different objects are restricted to repulsive forces instead of disallowing them altogether, the result is a simple penalty-based collision handling scheme. If i and j are particles from different objects, we apply a modified interaction force $\mathbf{F}'_{ij}$

$$\mathbf{F}'_{ij} = \begin{cases} (\frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|} * \mathbf{F}_{ij}) \frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|} & (\mathbf{x}_j - \mathbf{x}_i) * \mathbf{F}_{ij} > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{8.37}$$

Of course, more elaborate collision handling schemes can be implemented, but the implicit collision handling provided comes at no additional cost and is sufficient in most situations. Figure 8.8 shows an example animation of stiff-elastic objects. Collisions were resolved using the penalty-based collision described above.

8.2.7 Surface Animation

In order to render the simulation result, a surface needs to be extracted from the simulation data. The surface of fluids in the examples shown in this thesis was extracted using marching cubes [133] to extract an iso-surface of the fluid density.

In order to enable texturing of solid objects, a simple skinning approach is used to advect a surface with the particles. It is discussed in terms of a surface mesh, but works without modification for point-sampled surfaces. Each particle i stores the relative rest state positions $\mathbf{s}^0_{ij} = \mathbf{x}^0_j - \mathbf{x}^0_i$ of nearby surface mesh vertices j. The lattice transformation $\mathbf{A}_i$ that are computed in each time step is used to transform the relative positions, yielding estimates of the deformed position $\mathbf{s}_{ij} = \mathbf{x}_i + \mathbf{A}_i \mathbf{s}^0_{ij}$. The position estimates of all particles are then averaged using the SPH smoothing operator, yielding a deformed position for the mesh vertex

$$\mathbf{s}_i = \sum_j W_h(\mathbf{s}_{ij}) \mathbf{s}_{ij} V_j. \tag{8.38}$$

Since we make use of the SPH framework to interpolate the positions, the resulting surface is smooth. If the surface is represented as a point cloud, the same transformation can be applied to warp the tangents or normals of the surfels.

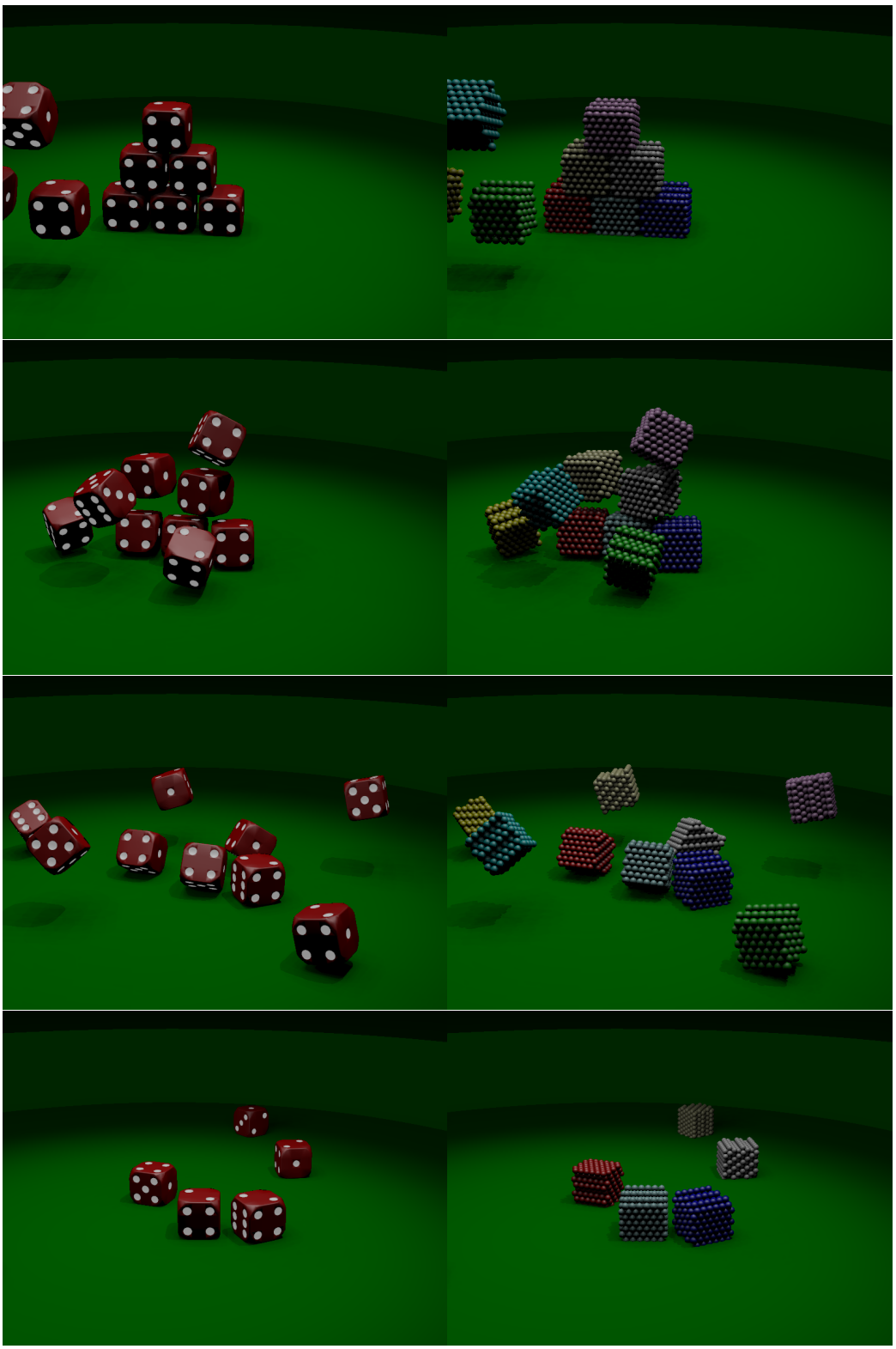


Figure 8.8: Several colliding stiff elastic objects. Inter-object collisions are handled as described in Section 8.2.6, no additional collision handling is necessary. On the right, the particle sampling is shown.

Figure	# Particles	# Objects	Time/Frame [s]
8.4	9871	1	17
8.5	1606 - 7000	2 (1 solid, 1 fluid)	18.2
8.8	2214	9	18.7

Table 8.1: Statistics for the examples shown in this chapter.

8.3 Results

Figures 8.4 and 8.5 show phase transitions between solid and fluid phase. Both phenomena are simulated using an SPH simulation enhanced with elastic forces as described above.

Figure 8.4 shows an elastic bunny being dropped and melting on the ground. The ground is heated, and a simple diffusion algorithm computed heat transfer between the ground and the particles. The surface is reconstructed using marching cubes.

In Figure 8.5, a viscous fluid freezes to a cooled rod. Fluid, ice and rod are modeled using the presented method. The surface of the cylinder is attached to its particles using skinning as described in Section 8.2.7, the surfaces for fluid and ice are computed using marching cubes. The collisions between fluid and solid are handled as described in Section 8.2.6, no additional collision handling mechanism is necessary.

Figure 8.8 shows nine stiff-elastic dice. Each die is sampled with 246 particles. The dice are rendered using a textured surface mesh which is moved along with the particles using the skinning method described in Section 8.2.7. The high stiffness of the material limits the time step size, making the animation more expensive to compute. The collision handling is performed using the method described in this chapter. Thus, collision handling does not incur additional costs for the simulation.

The simulation parameters used in the examples shown in this chapter are summarized in Appendix F.2. Simulation times are given in Table 8.1. The timings were measured on a Pentium 4 at 3 GHz and do not include rendering time.

8.4 Discussion

The method for enhancing an SPH simulation with elastic forces described in this chapter does not use any persistent connectivity. This is an advantage in situations where maintaining connectivity of any kind is problematic. From an algorithmic point of view, simulations involving melting or freezing are simplified significantly if no connectivity has to be considered.

A simulation method not requiring connectivity is an advantage on architectures where no complex data structures are available, in particular highly parallel processing units. Here, maintaining persistent neighborhoods is a major compu-

tational burden compared to streaming particles and their current neighborhoods, requiring only local computations.

The materials that can be modeled using the enhanced SPH method range from regular fluids to stiff elastic objects, such as seen in Figure 8.8. However, all of these materials exhibit the material properties described in Section 8.2.5. In particular, this means that materials simulated using this model deform plastically under large or abrupt deformations and fracture if stretched too much.

These specific material properties are a direct consequence of abandoning persistent connectivity, other approaches with no persistent connectivity exhibit the same behavior [204]. A comparison to simulations using Lennard-Jones potentials is instructive. These also rely on current spatial relationships alone to compute forces, and hence share many of the characteristics of the method described here. The rest state of a Lennard-Jones material is also implicit: For each particle, its rest state is given by the nearest minimum of the potential energy, which is a simple superposition of potential fields evaluated at the current particle positions. However, the potential energy may have many more minima than there are particles, such that particles can more easily move to a different minimum, leading to plastic deformation analogous to the one described in Section 8.2.5. While this is in fact the way plasticity works for crystalline materials, such as metals, plasticity derived from continuum mechanics is a better approximation for sampling densities such as found in typical computer graphics simulations.

The shape matching that replaces the implicit assignment to the nearest energy minimum is a smarter way of finding suitable rest state positions, making the material significantly more robust against inherent plasticity, even for large deformations. In order to simulate true elasticity that does not exhibit the inherent plasticity incurred by erroneous particle assignments, the initial particle neighborhoods need to be stored throughout the simulation. The resulting algorithm is very similar to [148]. Müller et al. organize the material into overlapping clusters of points for which shape matching is performed. If the particle assignment step is replaced with stored connectivity information, the algorithm described above is roughly equivalent to [148], where each neighborhood forms one cluster.

Müller et al. do not use forces in their simulation, but directly move the particles towards goal positions computed using shape matching. This leads to a material stiffness that is time step dependent. Their α-parameter can be translated to a stiffness K_e using $K_e = \alpha/\Delta t^2$. Thus, the stability criterion from [148], $\alpha \leq 1$ yields a upper bound on the time step depending on the material stiffness.

The major difference between the approaches is that abandoning persistent connectivity requires dynamic particle assignment in each time step. If large, non-plastic deformations are considered, the method presented here cannot be used due to its inherent restrictions. Connectivity information is crucial for these deformations. The next chapter introduces a finite element method that is able to model the full spectrum of linear continuum elasticity.

Chapter 9

Finite Elements on Irregular Meshes

The last chapter described a method for integrating elasticity into a fluid simulation. While this technique is useful for enhancing fluid simulations, it is not general enough to be used for arbitrary simulations involving elastic materials.

By far the most popular simulation method for elasticity computations is the finite element method. It requires that the simulation domain is partitioned into a finite number of elements. Hence, a consistent mesh that divides the simulation domain into disjoint primitives is needed. A function sampled at the vertices of the mesh can then be approximated by a sum over basis functions defined within each element. A comprehensive treatment of the theory behind FEM can be found in [23].

Most three-dimensional finite element methods partition the domain into tetrahedral or hexahedral elements. Thus, linear or trilinear shape functions can be used to interpolate within elements and the computations are simplified significantly. This chapter will explore a way to relax the regularity requirements for the discretization, allowing for arbitrary convex elements. As has been shown in previous chapters, abandoning guarantees on the connectivity can lead to more flexible methods, while increasing the cost of basic operators. In the case of a finite element method for arbitrary convex polyhedra, the interpolation functions become more cumbersome, while the ability to handle non-tetrahedral elements makes the simulation method more flexible.

In order to use arbitrary convex elements, basis functions based on mean value coordinates [73, 108, 109] are proposed. Therefore, a purely tetrahedral or hexahedral grid is no longer necessary. This method is a true generalization of linear tetrahedral finite elements: If the domain happens to be meshed with tetrahedral elements, the new formulation yields regular linear basis functions.

In the following sections, a finite element method based on new basis functions using mean value coordinates is introduced. The advantages become appar-

ent when the topology of the simulation domain changes during the simulation. This is demonstrated in simulations involving cutting, where costly and numerically challenging remeshing can be avoided.

9.1 Elastic Deformation

Deformation of elastic material is governed by the equations of continuum elasticity, which can be discretized using the finite element method. In the following, we consider an object with material coordinates $\mathbf{x} = [x,y,z]^T$ deformed by a displacement field $\mathbf{u}(\mathbf{x}) = [u_x(\mathbf{x}), u_y(\mathbf{x}), u_z(\mathbf{x})]^T$.

The elastic energy density of a deformable body is defined in terms of stress and strain within the object. We use Cauchy strain ε, which linearly depends on the Jacobian $\nabla\mathbf{u}$ of the deformation field $\mathbf{u}$. Recalling Eq. 8.26, we can write

$$\varepsilon = \frac{1}{2}\left(\nabla\mathbf{u} + (\nabla\mathbf{u})^T\right). \tag{9.1}$$

The strain of the material causes elastic restoring forces, represented by the 3×3 stress tensor σ. We assume a Hookean material, i.e. a linear stress-strain relationship, yielding

$$\sigma_{ij} = \sum_{k,l=1}^{3} \mathbf{C}_{ijkl}\,\varepsilon_{kl}, \quad i,j \in \{1,2,3\}, \tag{9.2}$$

with a tensor $\mathbf{C}$ containing the elastic coefficients of the material. As ε and σ are symmetric 3×3 matrices, their independent coefficients can be written as 6D vectors. For the strain, this yields

$$\varepsilon = \left[\frac{\partial u_x}{\partial x}, \frac{\partial u_y}{\partial y}, \frac{\partial u_z}{\partial z}, \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x}, \frac{\partial u_x}{\partial z} + \frac{\partial u_z}{\partial x}, \frac{\partial u_y}{\partial z} + \frac{\partial u_z}{\partial y}\right]^T. \tag{9.3}$$

Thus, the strain-stress relationship (9.2) becomes a 6×6 matrix product

$$\sigma = \mathbf{C}\varepsilon. \tag{9.4}$$

For an isotropic material, the constitutive matrix $\mathbf{C}$ only depends on the material's elasticity modulus (also Young's modulus) Y and Poisson ratio ν, controlling stiffness and volume preservation, respectively.

With stress and strain defined throughout the material, the total elastic energy $E(\mathbf{u})$ can be computed as the integral of stress times strain over the object's volume V:

$$E(\mathbf{u}) = \frac{1}{2}\int_V \sigma^T\varepsilon = \frac{1}{2}\int_V \varepsilon^T\mathbf{C}\varepsilon. \tag{9.5}$$

In a finite element discretization, the displacement is given at discrete locations. The space in between these nodes is divided into elements, such that the integral (9.5) can be written as a sum of per-element energies E_e:

$$E(\mathbf{u}) = \sum_e E_e = \sum_e \frac{1}{2} \int_{V_e} \varepsilon^T \mathbf{C} \varepsilon. \tag{9.6}$$

The following sections detail how the displacement function $\mathbf{u}$ (and thus, the strain ε) can be interpolated within the elements, and how the per-element energies are computed.

9.1.1 Interpolation Functions for Convex Polyhedra

In order to discretize the energy equation (9.5) the continuum object is decomposed into a finite number of elements, and each node i of this decomposition is associated with a material position $\mathbf{x}_i$, a displacement value $\mathbf{u}_i = \mathbf{u}(\mathbf{x}_i)$, and a scalar shape function $\phi_i(\mathbf{x})$. Now, the continuous function $\mathbf{u}(\mathbf{x})$ can be approximated by

$$\mathbf{u}(\mathbf{x}) = \sum_i \mathbf{u}_i \phi_i(\mathbf{x}). \tag{9.7}$$

Since the goal is to discretize the material into arbitrary convex elements, we require basis functions suitable to interpolate within an arbitrary convex polyhedron.

For 2D simulations, Wachspress coordinates [208, 209] and two-dimensional mean value coordinates [72, 196] have been used for finite element simulations on convex polygons. Generalizations of Wachspress and mean value coordinates to three dimensions are available [73, 107–109, 211]. In the following, mean value coordinates are used.

The generalized mean value coordinates are only defined on convex polyhedra with triangular faces. The faces of the elements are therefore triangulated in order to compute the weight functions. Triangulating non-triangular faces is unproblematic compared to computing a tetrahedralization of the element. Note that if two convex polyhedra share a common face, this face is necessarily planar, such that the exact nature of the triangulation does not change the shape of the elements. Therefore, a triangulation that avoids degenerate triangles can be chosen.

Given the triangulation of the surface of the element, we consider the vertex $\mathbf{x}_i$ and its edge-incident one-ring neighbors $\mathbf{x}_j$. The weight w_i is defined as a weighted sum of ratios of signed tetrahedron volumes

$$w_i(\mathbf{x}) = \sum_j \left[\frac{c_{j,j+1}}{V_{i,j,j+1}} + \frac{c_{i,j} V_{j-1,j+1,j}}{V_{i,j-1,j} V_{i,j,j+1}} \right], \tag{9.8}$$

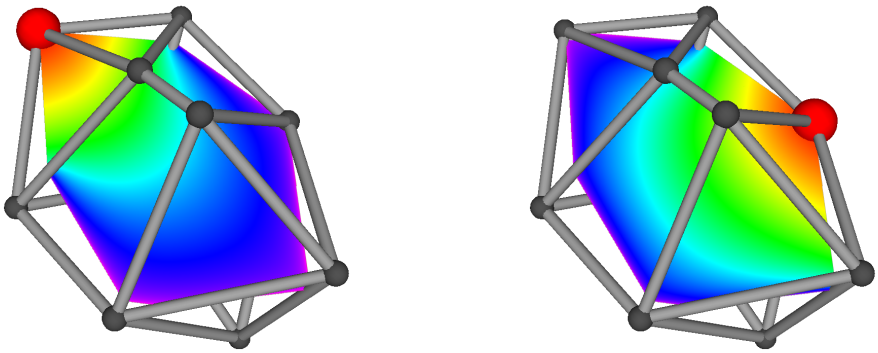


Figure 9.1: Basis functions for the highlighted nodes visualized by hue interpolation on a plane through the element.

where $V_{a,b,c} = V(\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c, \mathbf{x})$ is the signed volume of a tetrahedron with the nodes a, b, c, and the evaluation point $\mathbf{x}$ as vertices. The per-edge weights $c_{a,b}$ are given by

$$c_{a,b} = \frac{\|(\mathbf{x}_a - \mathbf{x}) \times (\mathbf{x}_b - \mathbf{x})\|}{6} \arccos\left[\frac{(\mathbf{x}_a - \mathbf{x}) * (\mathbf{x}_b - \mathbf{x})}{\|\mathbf{x}_a - \mathbf{x}\|\|\mathbf{x}_b - \mathbf{x}\|}\right]. \tag{9.9}$$

The mean value shape function ϕ_i within the element is finally obtained by normalizing the weight function w_i:

$$\phi_i(\mathbf{x}) = \frac{w_i(\mathbf{x})}{\sum_k w_k(\mathbf{x})}. \tag{9.10}$$

See Figure 9.1 for a visualization of ϕ_i.

The functions ϕ_i are true barycentric coordinates for convex polyhedra in the sense that they are positive inside the polyhedron (if it is convex), and that each point $\mathbf{x}$ inside the polyhedron can be written as a weighted sum of the vertices $\mathbf{x}_i$ with its coordinates as weights:

$$\mathbf{x} = \sum_i \phi_i(\mathbf{x})\,\mathbf{x}_i. \tag{9.11}$$

This property implies partition of unity and reproduction of linear functions: Let $f(\mathbf{x})$ be a linear function. Then, it can be easily shown that the approximation reproduces $f(\mathbf{x})$ exactly:

$$\sum_i \phi_i(\mathbf{x})\, f(\mathbf{x}_i) \overset{f \text{ linear}}{=} f(\phi_i(\mathbf{x})\,\mathbf{x}_i) \overset{(9.11)}{=} f(\mathbf{x}). \tag{9.12}$$

Setting $f(\mathbf{x}) = 1$ yields partition of unity.

Therefore, the functions ϕ_i fulfill all properties necessary to prove convergence of the finite element approximation: The basis functions are positive and reproduce linear functions. Their support is limited to incident elements. Continuity is C^1 within elements, and since they reduce to linear barycentric coordinates on the faces of the triangulation, the functions are C^0 continuous across element boundaries. Hence, the basis functions ϕ_i are in the Sobolev space H^1, and the finite element approximation converges [23].

In order to evaluate the strain, we require the first order partial derivatives of the shape functions. The derivatives of (9.8) and (9.10) can be computed analytically, the corresponding expressions are given in Appendix D.

Note that if the element is tetrahedral, the ϕ_i are linear, and equal to the shape functions commonly used in tetrahedral element discretizations. Hence, optimized code can be used for tetrahedral elements.

Numerical Issues

The shape functions defined as above are sums of volume ratios, which are problematic to compute if the volumes in (9.8) approach zero. This can occur in two situations: 1) if the point $\mathbf{x}$ lies on the boundary of the element, or 2) if a surface triangle has zero area.

The ϕ_i are not well defined on the boundary of the element, however, they converge to barycentric coordinates on the faces. Thus, this special case can easily be resolved. In practice, we can choose where to evaluate the interpolant during per element integration (see Section 9.1.3), and avoid evaluating ϕ_i or $\nabla\phi_i$ on the faces, except in the case of sliver elements. Slivers are almost planar elements that are known to pose numerical difficulties in FEM simulations. Owing to the flexibility of the discretization, it is possible to remove such slivers during the simulation. This technique is described in Section 9.2.

Triangles with zero area make no net contribution to the weights w_i. Although not obvious from Equations (9.8) or (9.10), this can be easily seen in the equivalent formulation presented in [109].

9.1.2 Finite Element Discretization

With the shape functions defined in the last section, the continuous energy (9.5) can be discretized using the approximation (9.7). In any particular element e with vertices $i = 1,\ldots,n_e$, only the shape functions $\phi_1,\ldots,\phi_{n_e}$ of its vertices are non-zero. Hence, within e, the displacement interpolation (9.7) is

$$\mathbf{u}(\mathbf{x}) = \underbrace{\begin{bmatrix} \phi_1(\mathbf{x}) & & & \phi_{n_e}(\mathbf{x}) & & \\ & \phi_1(\mathbf{x}) & \cdots & & \phi_{n_e}(\mathbf{x}) & \\ & & \phi_1(\mathbf{x}) & & & \phi_{n_e}(\mathbf{x}) \end{bmatrix}}_{=:\mathbf{H}_e(\mathbf{x})} \begin{bmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_{n_e} \end{bmatrix} = \mathbf{H}_e\mathbf{U}_e, \quad (9.13)$$

with a $3 \times 3n_e$ interpolation matrix $\mathbf{H}_e(\mathbf{x})$ and the $3n_e$ element displacement vector $\mathbf{U}_e$. The 6D strain vector (9.3) inside this element can then be written as

$$\boldsymbol{\varepsilon}(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix} \mathbf{H}_e(\mathbf{x})\,\mathbf{U}_e =: \mathbf{B}_e(\mathbf{x})\,\mathbf{U}_e. \tag{9.14}$$

From the stress-strain relationship (9.4) we get the per element energy element's energy density:

$$E_e = \frac{1}{2}\mathbf{U}_e^T \left(\int_{V_e} \mathbf{B}_e^T \mathbf{C} \mathbf{B}_e \right) \mathbf{U}_e =: \frac{1}{2}\mathbf{U}_e^T \mathbf{K}_e \mathbf{U}_e. \tag{9.15}$$

The element's $3n_e \times 3n_e$ stiffness matrix $\mathbf{K}_e$ is assembled by integrating products of partial derivatives of the shape functions, as described below.

Note that for linear elasticity, $\mathbf{K}_e$ is only computed in the rest state of the material, i. e. in a state where all elements are guaranteed to be convex. In particular, this means that the shape functions are always evaluated on convex polyhedra, even if the elements are non-convex in the deformed shape. Also, the expensive stiffness matrix setup is only necessary whenever the topology of the simulation domain changes.

Once the element stiffness matrices $\mathbf{K}_e$ are computed, the global $3n \times 3n$ stiffness matrix $\mathbf{K}$ is assembled [23]. If we denote by $\mathbf{U} = [\mathbf{u}_1^T, \ldots, \mathbf{u}_n^T]^T$ the vector of nodal displacements, the discrete version of the total elastic energy (9.5) becomes

$$E(\mathbf{U}) = \frac{1}{2}\mathbf{U}^T \mathbf{K} \mathbf{U}. \tag{9.16}$$

9.1.3 Integration

In order to compute the per element stiffness matrix $\mathbf{K}_e = \int_{V_e} \mathbf{B}_e^T \mathbf{C} \mathbf{B}_e$, we have to integrate over each element. For linear tetrahedral elements, these integrals are trivial to compute since $\mathbf{B}_e$ is constant over the element. For other simple element shapes, for example hexahedral elements, integrals can be evaluated using Gauss quadrature. In the more general case of irregular convex elements, such quadrature rules are unwieldy. Instead, the integral is approximated with a low number of sample points $\mathbf{p}$ heuristically placed throughout the element.

For integration, one sample per vertex of the element, plus one sample for each face of the triangulation of the element surface is used. The vertex samples are placed between the element centroid $\mathbf{c}$ and the vertex $\mathbf{x}_i$, at $\mathbf{p}_i = 0.8\mathbf{x}_i + 0.2\mathbf{c}$. The face samples are placed similarly, at $\mathbf{p}_f = 0.9\mathbf{c}_f + 0.1\mathbf{c}$, where $\mathbf{c}_f$ is the face

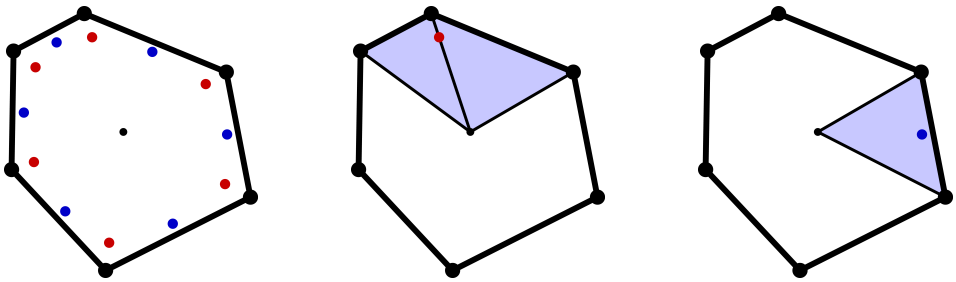


Figure 9.2: (a) Integration samples for a 2D element. Face samples are shown in blue, vertex samples in red. Is is important that the corners of the element are sampled appropriately. (b) The weight for a vertex sample is proportional to the volume between the incident faces with the element centroid. (c) The weight for a face sample is proportional to the volume between the face and the element centroid.

centroid. The heuristic ensures that there are samples near the boundaries of the element. Experiments have shown that it is important that there are integration samples near all nodes to make sure that all nodes have sufficient influence on the element energy. However, the exact location of the sample points does not have a critical influence on the simulation result. A 2D example of sample position in an irregular element is shown in Figure 9.2.

The integration samples are weighted proportional to the volume between the element centroid $\mathbf{c}$ and incident faces (for vertex samples), or the corresponding face (for face samples). With the same one-rings as in Section 9.1.1, we define the volume fraction μ_i^e of element e associated with the vertex i as

$$\mu_i^e = \frac{\sum_j V\left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_{j+1}, \mathbf{c}\right)}{3V_e}. \tag{9.17}$$

Similarly, the volume fraction for the face sample of face f with vertices j_1, j_2, and j_3 is

$$\nu_f^e = \frac{V\left(\mathbf{x}_{j_1}, \mathbf{x}_{j_2}, \mathbf{x}_{j_3}, \mathbf{c}\right)}{V_e}. \tag{9.18}$$

Using the weights μ_i^e and ν_f^e, the element stiffness matrix $\mathbf{K}_e$ is then computed as

$$\mathbf{K}_e = \sum_i \frac{\mu_i^e}{2} \mathbf{B}_e^T\left(\mathbf{p}_i\right) \mathbf{C} \mathbf{B}_e\left(\mathbf{p}_i\right) + \sum_f \frac{\nu_f^e}{2} \mathbf{B}_e^T\left(\mathbf{p}_f\right) \mathbf{C} \mathbf{B}_e\left(\mathbf{p}_f\right). \tag{9.19}$$

In the special case of a tetrahedral element, only one integration point is necessary. Note that while computing the element stiffness matrix for arbitrary elements by integration is more complex than in the tetrahedral case, this has only minor impact on the overall simulation complexity. For linear elasticity, the stiffness matrices of the elements are constant throughout the simulation, and can be

pre-computed. Recomputation is necessary only if the discretization is changed. This can happen in simulations involving adaptive refinement, or due to element splitting after fracture or cutting. Thus, the computational complexity during the actual simulation is mainly dependent on the total number of nodes.

9.1.4 Simulation Loop

The discrete energy (9.16) leads to the discrete equations of motion

$$\mathbf{M}\ddot{\mathbf{U}} + \mathbf{D}\dot{\mathbf{U}} + \mathbf{K}\mathbf{U} = \mathbf{F}, \tag{9.20}$$

where **M** and **D** are the mass and damping matrices, respectively, and **F** represents external forces.

In order to simplify the computations, we employ mass lumping to obtain a diagonal mass matrix **M**. Assuming constant density per element, the mass of each element can be trivially computed from its volume. We then use the volume fractions μ_i^e to obtain a mass for each node by summation over all incident elements:

$$m_i = \sum_e \mu_i^e V_e \rho_e, \tag{9.21}$$

where ρ_e is the density of element e. The volume ratio μ_i^e is a good approximation of the integral over the shape function ϕ_i within the element.

The linear elasticity model (9.3) is not invariant to rotations. Therefore, stiffness warping [147] is used to control linearization artifacts. Stiffness warping estimates a rotation for each element, and rotates the displacement values back to best match the rest shape. Strain is then evaluated on the rotated displacements, and the resulting forces are re-transformed into the current state.

This method requires that we compute per-element rotation matrices $\mathbf{R}_e$. The shape matching method described by Müller et al. [147] only works for tetrahedral elements. Therefore, we adopt the registration method presented by Horn [103] instead, which has the additional advantage of being stable even for degenerate planar elements. Once the per-element rotations are known, the global stiffness matrix **K** has to be reassembled using the rotated element stiffness matrices $\mathbf{K}'_e = \mathbf{R}_e^T \mathbf{K}_e \mathbf{R}_e$. Using stiffness warping, only the element rotations have to be recomputed in each time step, the un-rotated stiffness matrix $\mathbf{K}_e$ which is computed by costly integration is constant.

Implicit Euler integration is used to solve for the dynamic behavior of the object. Since most real-world deformable objects are strongly damped, the numerical damping introduced by the integration scheme is acceptable. For undamped simulations, symplectic integration can be used [114].

Figure 9.3 shows deformation of a single element with 12 nodes. Since the basis functions are nonlinear, the deformations are nonlinear as well, even though the object is sampled with only a single element.

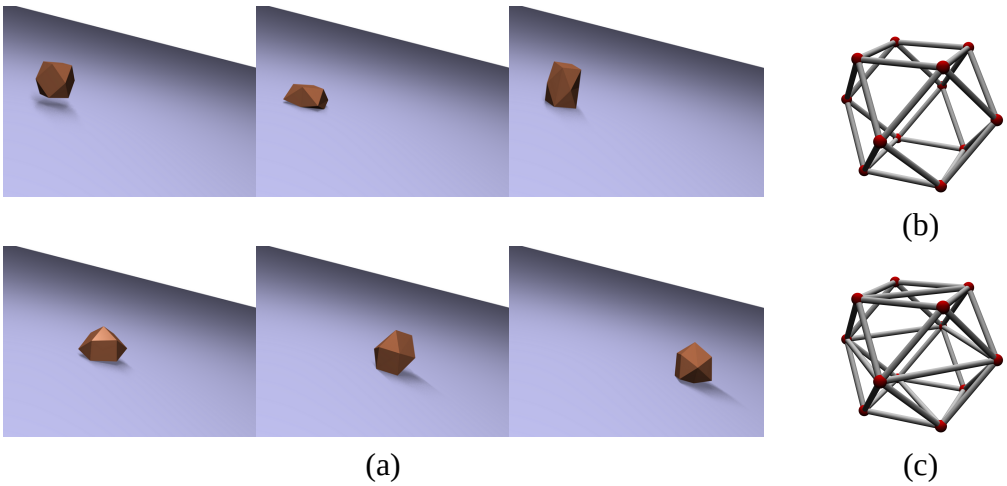


Figure 9.3: (a) A single element deforms on impact with the ground. The deformations of the element are nonlinear. (b) Wireframe of the element. (c) Wireframe view with triangulated faces.

Figure 9.4 shows a cube that deforms on impact. The cube is partitioned into elements by 30 randomly chosen planes. For each plane, an element is chosen which is then split as described in Section 9.3.1. The initial sampling is a single hexahedral element, yielding a consistent mesh of 31 elements after the plane splits are complete. The elements are highly irregular, and have between 8 and 32 nodes.

9.2 Sliver Removal

The accuracy of a finite element solution depends on the quality of the discretization. For tetrahedral discretizations, several quality criteria for the shape of elements have been proposed as a quality measure [181, 182]. Contrary to tetrahedral meshes, it is unclear what criteria determine the quality of a convex polyhedral element. Experiments suggest that bad elements are almost planar. These elements give rise to numerical problems during simulation, because the gradients of basis functions inside such elements cannot be evaluated robustly. Following Shewchuck [181, 182], we call elements that are (almost) planar due to degenerated edges *needles*, while planar elements without degenerated edges are called *slivers*.

Needles are easy to avoid: Whenever a node is created (during initial meshing or remeshing during the simulation, it is snapped to existing nodes if incident edges would become too short. Most widely available meshing software, such as TetGen, can control the minimum edge length, and checking for nearby nodes

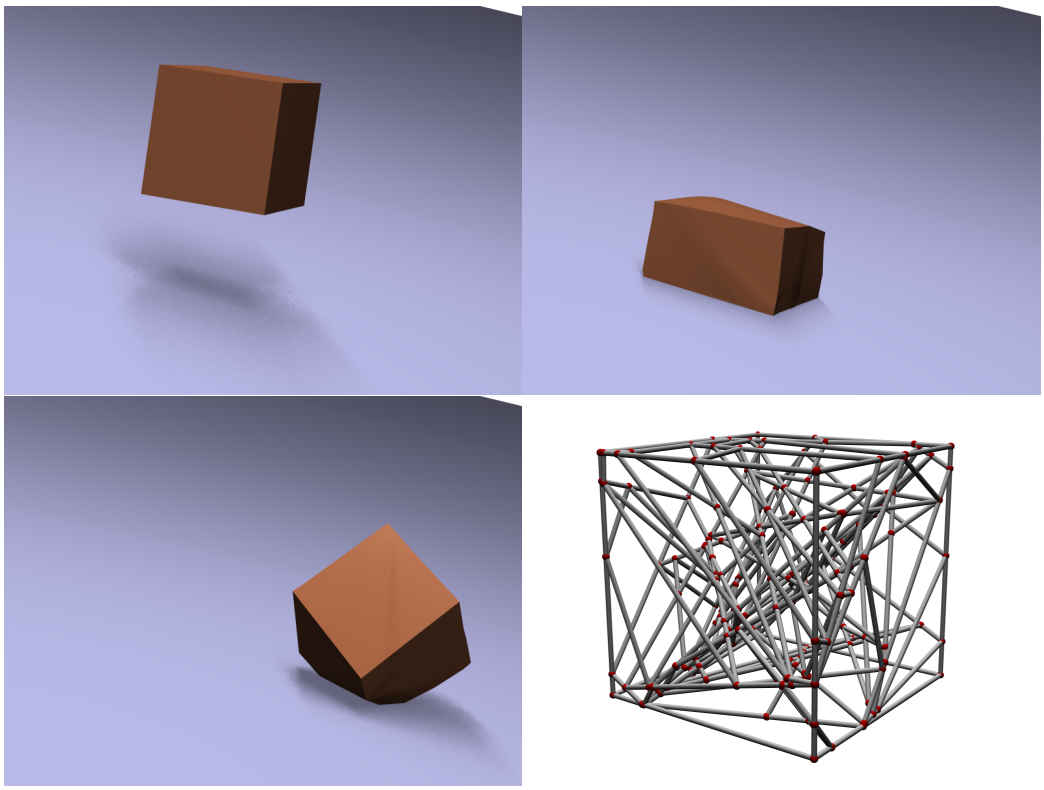
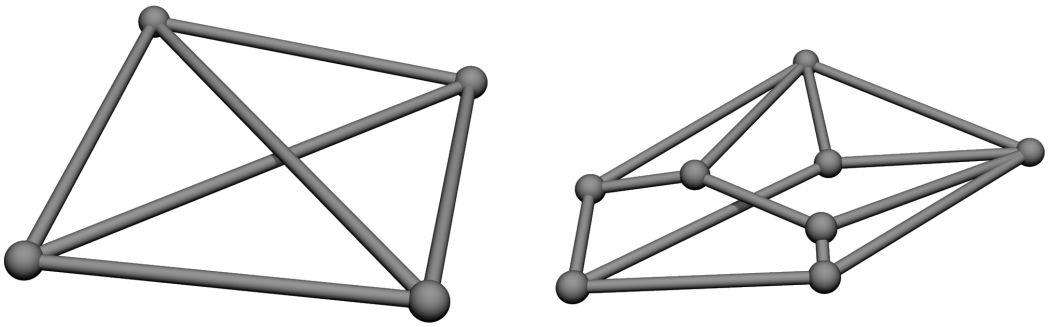


Figure 9.4: A cube deforms on impact with the ground. Bottom right: Wireframe view showing the tessellation into elements.

during cutting is relatively easy, such that needles do not cause problems during simulations.

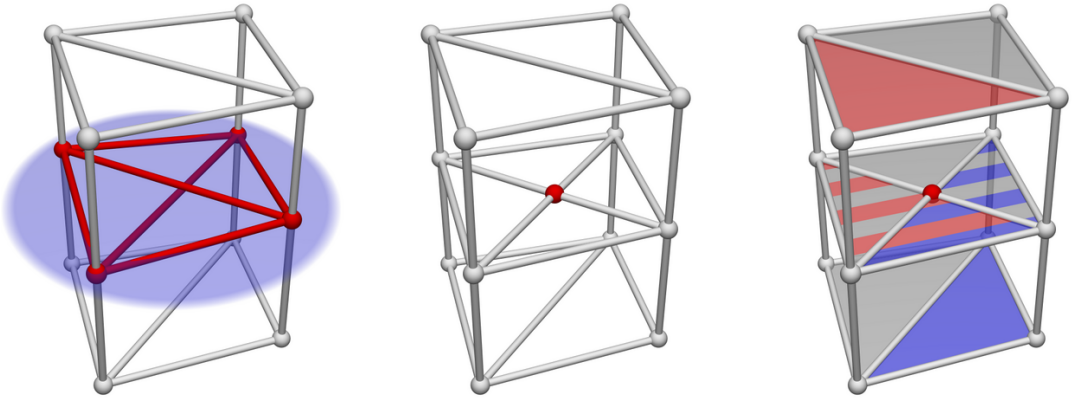
Slivers are more problematic. See Figure 9.5 for examples. Creating slivers is notoriously hard to avoid, and robustly removing them from a discretization is a daunting task. For tetrahedral meshes, remeshing algorithms based on constrained Delaunay tetrahedralization insert more nodes into adjacent elements and remesh the neighborhood of the sliver element. This process is costly, and is not guaranteed to be local [183].

When using FEM on convex polyhedra, there is no restriction to tetrahedral elements, and sliver elements can be merged with neighboring elements. This process consists of the following steps (see Figure 9.6 for an illustration): First, a least-squares plane through the element is computed, which we call the *sliver plane*. All vertices of the sliver element are projected onto the sliver plane. Note that this does not significantly change the geometry of the surrounding elements, as the sliver element is almost planar by definition. We can hence guarantee that sliver removal does not create more sliver elements, and all elements in the discretization stay convex.



(a) (b)

Figure 9.5: Sliver elements: (a) A tetrahedral sliver element. Note that all faces can have reasonable areas, and no edge is too short. (b) Allowing arbitrary convex polyhedra can lead to more complex slivers.



(a) (b) (c)

Figure 9.6: Removing sliver elements: (a) A sliver element and its neighbors. (b) New nodes are created at edge intersections. (c) After tessellating the sliver plane, new faces are connected to their neighboring elements. The faces in the sliver plane are colored with the color of both elements they are connected to. Note that the shape of the adjacent elements is not changed, only their connectivity is modified to eliminate the sliver.

If the projection moves nodes too close together, those nodes are merged to avoid the creation of needles. Then, all edges in the sliver plane are intersected and new nodes are inserted at the intersection points. Finally, the sliver plane is retessellated and each new face is connected to the two elements that were attached to the old faces it intersects. Thus, the sliver element is removed.

If the elements were tetrahedral before the sliver was removed, exactly one new node is created using this technique. The neighboring elements have five nodes each after the sliver is deleted. In more complex cases, more nodes might be inserted.

Although the geometry does not change significantly during sliver removal, projecting the nodes of a sliver element onto the sliver plane may lead to slightly non-convex elements in the neighborhood. The mean value coordinates used to construct basis functions are well-behaved in the case of slightly non-convex elements [109]. Hence, the simulation is stable in these cases. On the other hand, generalized Wachspress coordinates [211] are much more sensitive to non-convex elements.

9.3 Cutting

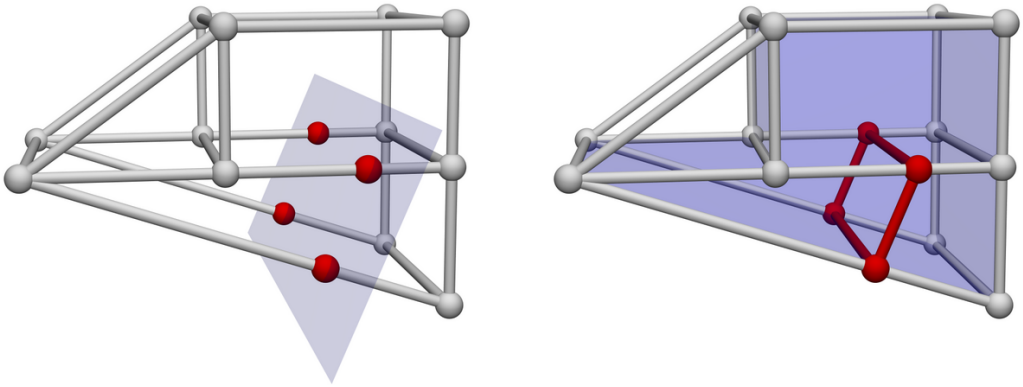
Changing the topology of the simulation mesh shows the advantages of discretizing the domain into convex elements instead of tetrahedral cells. As our only requirement to the mesh is that all elements must be convex, maintaining a valid simulation mesh after cutting operations is significantly easier.

This section first describes how elements are split by a single plane. In Section 9.3.2, the more general case of progressive cuts is considered.

9.3.1 Splitting Elements

Splitting a convex polyhedron along a plane results in two convex polyhedra. Thus, after planar element split operations, no remeshing is necessary in order to maintain a valid discretization of the simulation domain.

The splitting process works as follows: Wherever the splitting plane intersects existing edges of the element to be split, new simulation nodes are created. We compute displacement samples for the new nodes using our interpolant $\mathbf{u}$, which is linear on the edges of the discretization. During these edge splits, care has to be taken not to create nodes too close to existing nodes. In practice, nodes that would be created too close to existing nodes are snapped to the existing geometry. This avoids needle elements mentioned in Section 9.2. For all elements that were changed in the process, new integration samples are computed. In order to avoid computing integration samples multiple times, reinitialization of element integration samples is deferred until the end of the time step. Figure 9.7 illustrates the procedure.



(a) (b)

Figure 9.7: (a) The bottom element is split along a plane. New simulation nodes (red) are added where the cutting plane intersects the original geometry of the element. (b) New integration samples are created in all elements that were changed by the split (shaded blue). Not all neighboring elements need to be updated.

Note that t-junctions might be created during splitting, as seen in Figure 9.7. This is not a problem, since the only requirement to the discretization is that all elements are convex. At a t-junction, one element has two coplanar faces.

9.3.2 Progressive Cuts

When progressive cuts are considered, elements are not necessarily split entirely in a single time step. Instead, we have to deal with an arbitrary surface intersecting the element [32, 193]. This surface is assumed to be tessellated into polygons. It is therefore sufficient to show how one element can be cut by a polygon, yielding a valid mesh whose boundary contains the polygon. The simulation mesh can then be cut by an arbitrary cut surface by sequentially cutting it with each of the surface polygons.

As the *cut shape*, consider a polygon with vertices $\mathbf{p}_1 \dots \mathbf{p}_k$, lying in a common *cutting plane*. First, all elements that intersect the cut shape are split along the cutting plane, as described in Section 9.3.1. Then, simulation nodes are created at the polygon points $\mathbf{p}_1 \dots \mathbf{p}_k$, which are connected with edges. The polygon edges are intersected with the faces of the simulation mesh and all intersection points are added as new simulation nodes. The faces in the cut plane are split to accommodate the new nodes and edges. If this process generates non-convex faces, those faces are split again to enforce that all generated faces are convex. Finally, all simulation nodes inside the cut shape are duplicated, and their incident elements are separated along the cutting plane. Figure 9.8 illustrates the necessary steps. After the partial cut, the element has been split along the plane, creating two ele-

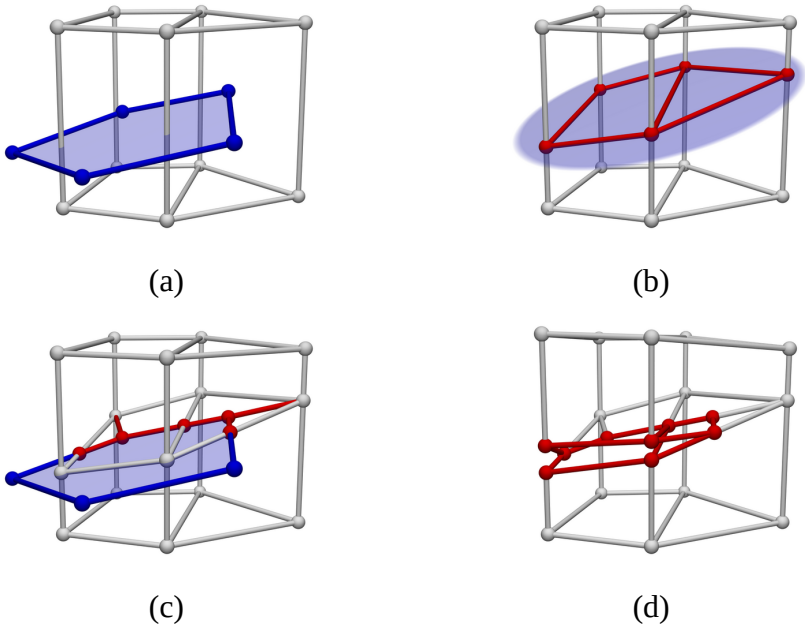


Figure 9.8: (a) Two elements are cut with a polygonal cut shape (blue). (b) All intersected elements are split along the cutting plane. (c) The polygon edges are intersected with the existing edges of the mesh, and new nodes are inserted at intersections. Faces in the cutting plane are split to create a consistent tessellation. (d) Nodes inside the cut shape are duplicated, the material is separated along the cut.

ments. These element are separated in the area of the cut shape, but stay connected outside the cut shape, creating a partial cut.

If there are no simulation nodes inside the cut shape, the cut cannot open. In such cases, one additional simulation node is created at the centroid of the cut shape, and connected with edges to the polygon nodes at $\mathbf{p}_1 \dots \mathbf{p}_k$, thus creating a pocket in the material.

Note that for non-tetrahedral elements, cuts through a single element might not be planar. In these cases, the surface within the element has to be tessellated after its edges have been intersected. The element is then sequentially cut with all faces of the tessellation.

9.4 Results

Figure 9.9 shows progressive cuts. The cube is initially sampled with $3 \times 3 \times 3$ hexahedral cells. During the cuts, only nodes that are necessary to sample the cut surface are created. As elements are cut repeatedly, the number of nodes created would be significantly higher if elements had to be subdivided to maintain a

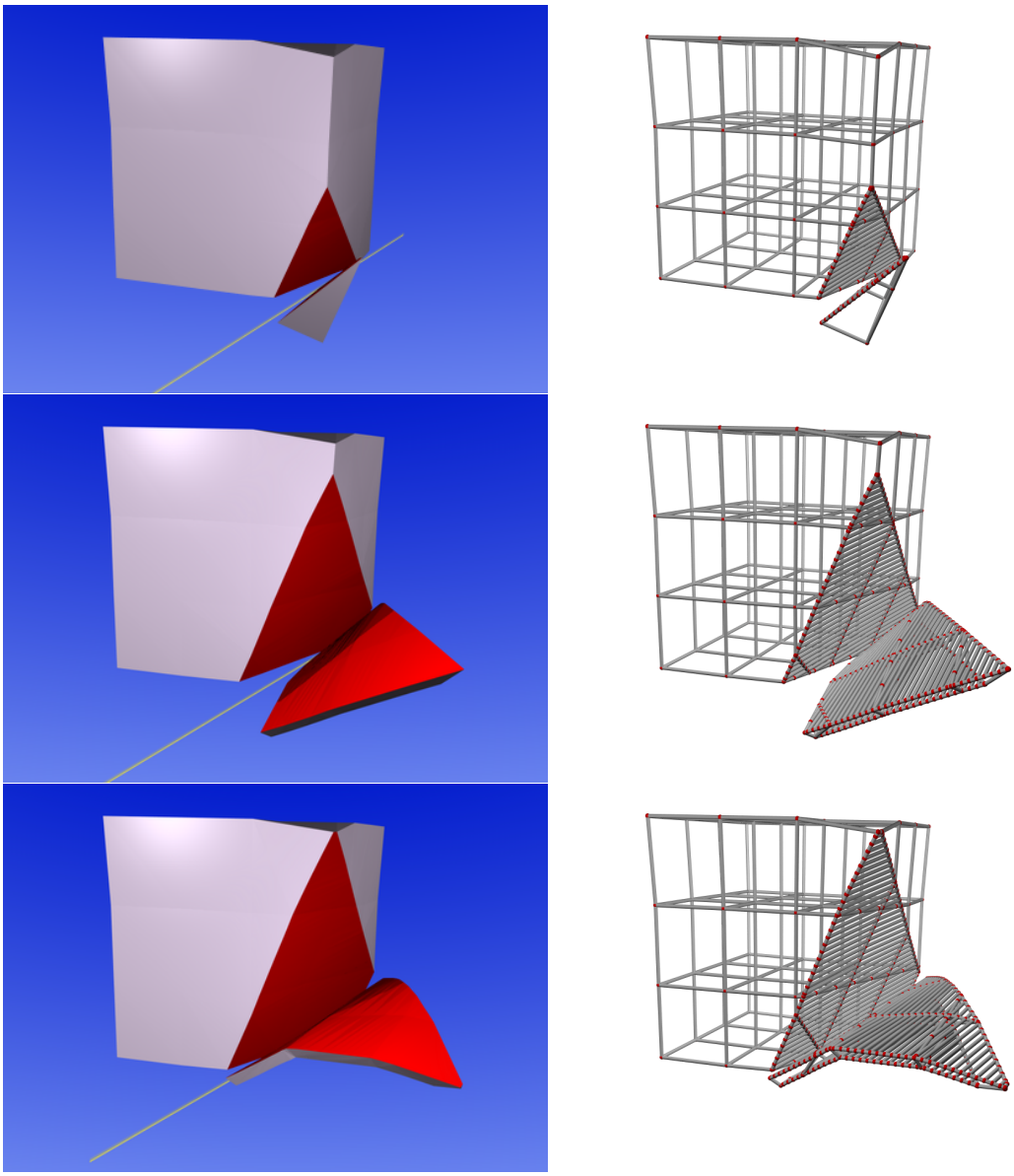


Figure 9.9: A block of material is sliced. Very few additional elements are created as tetrahedral subdivision is not necessary. The bottom row shows the elements. Note that even though many nodes are created to accurately represent the cut surface, elements do not need to be split.

	Start		End		Avg. time/
Figure	# Nodes	# Elements	# Nodes	# Elements	frame [s]
9.3	12	1	12	1	0.012
9.4	184	31	184	31	0.22
9.9	64	27	1214	60	0.8 (0.04)
9.10	24079	8278	46132	44118	6.08 (3.56)

Table 9.1: Node count, element count, and computation time. The time in parenthesis is the computation time for the dynamic update not including recomputation of basis functions.

tetrahedral mesh. Edges added in order during tetrahedral remeshing would be cut again, leading to more elements and simulation nodes.

More complex, non-planar cuts are shown in Figure 9.10. The initial sampling is a tetrahedral mesh created with the meshing software TetGen. After all cuts are complete, 51% of the elements are still tetrahedral, the elements with the highest number of nodes has 27 nodes. Performing the same sequence of cuts using a state of the art tetrahedral subdivision method [193] results in more than 75000 nodes and more than 300000 elements, even if the cuts are executed non-progressively, i. e. each element is split at most once per cut. The exact number of nodes and elements in the tetrahedral setting depends on snapping thresholds.

Simulation parameters for the examples shown in this chapter can be found in Appendix F.3. Table 9.1 summarizes the computation times, as well as node and element counts for the examples shown here. The timings were measured on a Pentium 4 at 3 GHz, and exclude rendering time. The simulation time excluding recomputation of basis functions after cuts depends mostly on the total number of nodes. Due to the shape matching needed for stiffness warping, there is a linear dependence on the number of elements. Highly optimized code is available for tetrahedral FE simulation. On purely tetrahedral models, the proposed method is clearly slower than specialized implementations. However, the theoretical complexity in these cases is of the same order. As soon as topological changes are considered, the number of elements and nodes in the discretization grows faster when only tetrahedral elements are allowed.

9.5 Discussion

The finite method presented in this chapter is a true generalization of linear tetrahedral finite elements. If the simulation mesh composed of tetrahedral elements, the methods are identical. However, since we are no longer restricted to tetrahedral elements, we have more flexibility in choosing the domain discretization. This is

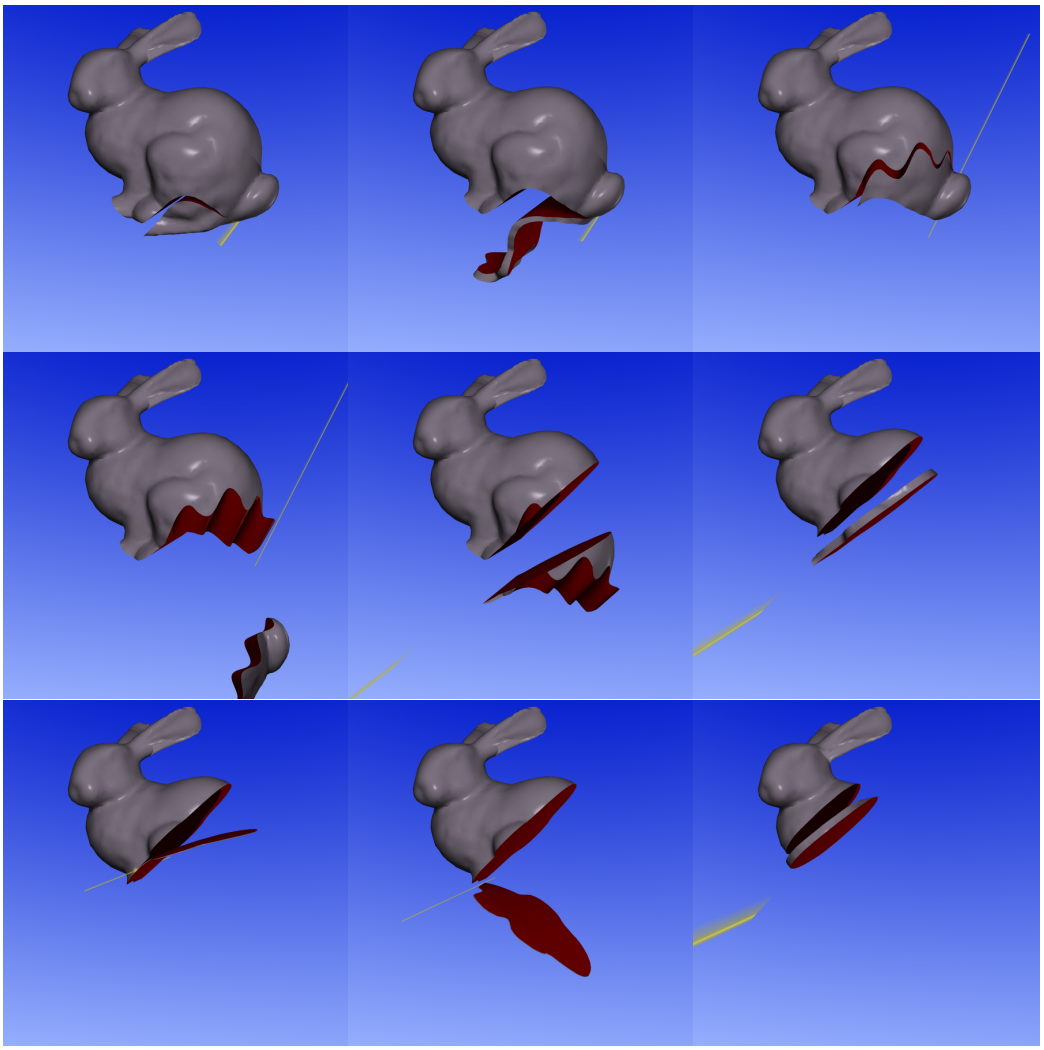


Figure 9.10: Slicing the Stanford Bunny. The cut trajectories are accurately represented. We can cut extremely thin slices without mesh restructuring — the simulation method is stable in these cases.

particularly useful when the domain topology is changed, for example by cutting, or fracture.

Because the shape functions based on mean value coordinates are only positive for convex polyhedra, non-convex elements are not allowed in the discretization. Slightly non-convex shapes can occur in the mesh, for example due to numerical inaccuracies or after sliver removal. This is not a problem, since the basis functions are smooth, and smoothly change depending on the element vertex positions. However, severe concavities would be problematic. This is why nonlinear strain cannot be simulated using the method presented here. Nonlinear elasticity would require evaluating the basis functions and their gradients also for the deformed state of the simulation mesh. For large deformations, the elements might not be convex in the deformed state. One possible remedy might be to subdivide the deformed elements into convex parts in each time step. Irving et al. [105] treat similar problems arising for nonlinear strain in hexahedral elements, however, their method is not directly applicable to arbitrary convex polyhedra. DeRose and Meyer [63] propose a method to compute barycentric (and in particular, positive) coordinates for arbitrary, even non-convex meshes. However, their technique involves discretizing the mesh into a grid and solving a Laplacian. Doing so in each time step, and for each element would be prohibitively expensive, and certainly not competitive with tetrahedral subdivision.

Chapter 10

Conclusion

In this thesis, several algorithms with reduced connectivity requirements have been presented. Compared to the state of the art in computer graphics, these techniques require less structure within the computational domain. The methods treat problems in two major fields of computer graphics: Geometric modeling and physically-based animation.

In the first part of this thesis, methods for modeling with point-sampled geometry were examined. Point-sampled surfaces are an alternative to triangle meshes which are the most commonly used surface representation in modeling. Their inherent lack of connectivity significantly complicates some operations. In Chapter 4, a method for representing and rendering discontinuities in point-sampled objects was presented. For edge rendering, the surfaces are considered to consist of overlapping disks. The actual clipping operations are deferred until rendering, which makes the approach ideally suited for dynamic data, where costly preprocessing cannot be performed. Unlike previous approaches for representation of geometric discontinuities in point-sampled objects, the proposed method can represent arbitrarily complex edges and corners.

Chapter 5 described a virtual painting system that makes extensive use of point-sampled surfaces as its internal surface representation. The system uses the superior resampling capabilities of point-sampled surfaces to dynamically adapt the surface sampling to the texture detail. Contrary to traditional, mesh-based approaches that rely on dynamically updated texture atlases, projection and discontinuity artifacts at texture patch boundaries can be avoided entirely. The surface representation is hidden from the user, and the system is designed to be intuitive to use by closely mimicking the real-world painting process. This is achieved by consequently following the painting metaphor, including a painter's palette, a physically-animated brush model, and haptic feedback. To guarantee interactive frame rates, a specialized renderer is used that is capable of modifying point-sampled surfaces without completely re-rendering them.

Much work has been done on point-sampled surfaces, and the number of applications supporting or even requiring point-sampled surfaces as a data representa-

tion has increased dramatically in recent years. However, meshes remain the dominant surface representation in computer graphics. Since each representation has its own advantages and disadvantages, it is important to ensure interoperability. Chapter 6 presents a method to convert point-sampled models to textured triangle meshes. An error metric controls the resolution of the textures used to capture the appearance of the surface. This way, objects produced in tools relying on point-sampled surfaces, for example the models produced using the painting system described in Chapter 5, can be converted and reused in mesh-based applications.

The second part of the thesis treated problems in the field of computer animation. Current research in computer animation is focused on physically-based animation. Simulation algorithms for physical phenomena usually have strict requirements regarding the discretization. Conversely, the nature of the discretization determines the type of algorithm applicable to a specific problem. In Chapter 7, a method for animation of point-sampled surfaces as thin shells was presented. Since the surface representation does not partition the mesh into disjoint elements, finite element methods or other methods that require a more structured representation cannot be applied to point-sampled surfaces. Instead, a network of fibers encoding the local neighborhood structure is used to provide area and curvature measurements. This information is then used to geometrically approximate the thin shell energy functional and animate the surfaces.

Chapter 8 explored the possibility of implementing elasticity without storing connectivity. This way, elastic forces can be integrated into a particle-based fluid simulation without storing a rest state. Since both elastic and fluid behavior can now be simulated using the same framework, phase transitions between solids and fluids are particularly easy to model. However, the connectivity information computed in each time step cannot sustain elastic forces under large deformations, and plasticity is inherent to the approach. Although less pronounced, this behavior is similar to other approach relying only on implicit connectivity. For true elasticity simulation, some stored connectivity is necessary.

The most popular method for the simulation of elastically deforming objects is the finite element method. Here, the material is usually discretized into a tetrahedral, or hexahedral mesh. In chapter Chapter 9, a more general finite element method was presented that does not require a certain element shape and is applicable to to irregular volumetric meshes consisting of arbitrary convex elements. Mean value coordinates are used to construct basis functions for the more general elements. The basis functions are nonlinear, requiring per-element integration of the strain. Since quadrature rules for approximate evaluation of the per-element stiffness matrices are not available for general convex polyhedra, a heuristic quadrature is used to approximate the integrals. Abandoning the requirement that all elements have to be of one specific shape drastically simplifies remeshing after topological changes within the domain. This is particularly useful in simulations involving cutting, where remeshing can be avoided after planar element splits, and slivers can be removed locally.

10.1 Discussion

Naturally, each form of data representation or discretization has its own strength and weaknesses. This thesis has presented several alternative methods with less stringent requirements on the connectivity present in the underlying data representation. These new algorithms present us with a trade-off. In the case of modeling with point-sampled surfaces, we gain easier resampling compared to triangle meshes, but we lose guarantees on topology and the ability to natively represent sharp features and geometric discontinuities. Hence, sharp features have to be represented explicitly, for instance as proposed in Chapter 4. The ability to easily resample surfaces where needed is used extensively in the painting system described in Chapter 5.

For algorithms in physically-based animation, the discretization largely determines the type of algorithms that can be used, and thus which convergence properties can be achieved and which physical phenomena can be modeled. In Chapter 7, we have traded the convergence guarantees of finite element methods for the ability to animate point-sampled thin shells without first computing a consistent triangulation.

The enhanced SPH simulation method proposed in Chapter 8 can simulate elasticity up to a certain point, but without persistent connectivity, it is impossible to avoid plastic effects and fracture under large deformations. However, since no connectivity is stored, the method can be integrated into an SPH fluid simulation, and the lack of rest state information makes phase transitions easy to implement.

Finally, the finite element method described in Chapter 9 maintains all convergence properties of standard tetrahedral finite element methods for linear elasticity, even though it relaxes the discretization requirements to irregular meshes consisting of convex elements. However, due to the definition of the basis functions, the approach is still limited to linear elasticity. Relaxing the discretization has other advantages, again related to resampling: In simulations involving cutting or fracture, the remeshing process necessary after topological changes in the domain is simplified significantly, and the ability to locally remove sliver elements yields a more stable simulation.

There is no easy answer to the question how much connectivity is optimal. As can be seen in the examples presented herein, some tasks require a certain minimum structure. However, this minimum is often lower than the requirements of the dominant method in the area. In some cases, additional structure presents more of a burden to the application than is gained by the added possibilities. A lighter data representation can perform better in such cases, examples for this are the finite element method from Chapter 9, or the painting system in Chapter 5.

In any case, interoperability with other methods has to be ensured. For modeling applications, this means we have to explicitly provide methods for conversion between representations, as done in Chapter 6. A basic level of compatibility for physics-based simulations can be achieved by allowing external forces to interact

with the simulation. Even this basic interaction is not always straightforward to implement if the discretization differs. Ideally, a new method can be used side-by-side with traditional approaches, such as the finite element method from Chapter 9, which encompasses traditional tetrahedral finite elements as a special case.

10.2 Outlook

With the availability of more flexible graphics processors, a large part of the computational burden of the methods for point-sampled modeling discussed here can be off-loaded to the graphics card. Today, it should be possible to implement the explicit handling of discontinuities for point-sampled surfaces as proposed in Chapter 4 entirely in hardware. Also the painting system described in Chapter 5 will benefit greatly from a hardware implementation of paint transfer.

We have seen that loosening the requirements on data representations makes it easier for them to be modified during use. Point-sampled modeling systems use this fact to resample the surface whenever necessary. Level of detail rendering systems relying on point samples were among the first applications of splat rendering.

Implementing adaptive physical simulation is more involved. Methods that adaptively solve for dynamic behavior rely on hierarchies of discretizations. Of course, such hierarchies are even harder to maintain than a flat structured representation. Representations with less structure can be used for adaptive multilevel simulation and might drastically reduce the maintenance overhead of such algorithms.

Appendix A

Correctness of Surfel Clipping

Algorithm 4.2 determines whether or not a fragment at position **x** created by a surfel belonging to an object $\mathcal{C}$ needs to rendered. The algorithm traverses the CSG tree bottom up, starting at the leaf node representing $\mathcal{C}$. The algorithm always terminates, either if at some point $on = false$, or when the root node is reached.

To show the correctness of the algorithm, we use a slightly modified but equivalent version of the algorithm, shown below, which simplifies the argument.

```
   Input: Point x, Patch C
 1 set T_A = T_C, on = true
 2 while ¬isRoot(T_A) do
 3     set T_B = getSibling(T_A)
 4     set in = insideTree(x, T_B)
 5     if in ≠ unknown then
 6         switch T_A.operator do
 7             case ∩ : set on = on ∧ in
 8             case ∪ : set on = on ∧ ¬in
 9     set T_A = getFather(T_A)
10 return on
```

Algorithm A.1: Determining whether a fragment at **x** of patch $\mathcal{C}$ should be rendered (without early termination)

We artificially extend the loop to traverse the tree until we reach the root node, also in case *on* becomes false. Hence, the condition of the while loop is changed to $\neg\texttt{isRoot}(T_\mathcal{A})$. The equations

$$[\mathbf{x} \in \mathcal{S}_{\mathcal{A}\cap\mathcal{B}}] \quad \Leftrightarrow \quad [\mathbf{x} \in \mathcal{S}_\mathcal{A} \wedge \mathbf{x} \text{ inside } \mathcal{B}], \tag{A.1}$$

$$[\mathbf{x} \in \mathcal{S}_{\mathcal{A}\cup\mathcal{B}}] \quad \Leftrightarrow \quad [\mathbf{x} \in \mathcal{S}_\mathcal{A} \wedge \neg[\mathbf{x} \text{ inside } \mathcal{B}]]\,. \tag{A.2}$$

replace (4.5) and (4.6) in lines 7 and 8. Note that as $\mathbf{x} \in \mathcal{S}_\mathcal{C}$, (4.5) and (4.6) are equivalent to (A.1) and (A.2).

It is easily verified that the above algorithm is equivalent to Algorithm 4.2. If Algorithm 4.2 returns *true*, so does Algorithm A.1: In this case, the loop terminates when reaching the root node, thus the changed termination condition in line 2 does not change the outcome of the algorithm. When computing a new value for *on*, *on* is never false. Therefore, it can be ignored in the conjunctions in lines 7 and 8, leading to the same equations as used in Algorithm 4.2.

If Algorithm 4.2 returns $false$, so does Algorithm A.1: In this case, $on = true$ only holds up to some iteration. Before that point, the two algorithms behave identical, as demonstrated above. Algorithm 4.2 terminates as soon as $on = false$. It is sufficient to show that in Algorithm A.1, once $on = false$, *on* is not changed any more. As *on* itself is one operand of the conjunctions in lines 7 and 8, these always yield $false$. Thus, $on = false$ until the root node is reached, and $false$ is returned as required.

The algorithm returns *on*, therefore it is correct if we can show that

$$[on = true] \Leftrightarrow [\mathbf{x} \in \mathcal{S}_{T_\mathcal{A}}] \tag{A.3}$$

is an invariant of the while loop. Written out, $\mathbf{x} \in \mathcal{S}_{T_\mathcal{A}}$ means "$\mathbf{x}$ is on the surface of $\mathcal{A}$", i. e. $\mathbf{x}$ is on the surface of the object represented by the current node. Once the root node is reached, we obtain the desired result.

This can be easily shown by induction over the number of iterations. The condition holds at the beginning of the first iteration: since $\mathbf{x}$ is part of a surfel of object $\mathcal{C}$ and $T_\mathcal{A} = T_\mathcal{C}$ due to the initialization of $T_\mathcal{A}$, $\mathbf{x} \in \mathcal{S}_\mathcal{A}$. *on* is initialized to *true*, so (A.3) holds.

Given that (A.3) holds at the beginning of some iteration, we show that it also holds at the end. We denote the value of $T_\mathcal{A}$ at the beginning of the loop $T_{\mathcal{A}_\text{start}}$. Its value at the end of the loop is $T_{\mathcal{A}_\text{end}}$, the father node of $T_{\mathcal{A}_\text{start}}$ and its sibling node $T_\mathcal{B}$. Since (A.3) holds, $\mathbf{x} \in \mathcal{S}_{\mathcal{A}_\text{start}}$. There are two cases to consider, depending on the inside/outside classification for $T_\mathcal{B}$, i. e. depending on the value of *in*.

1. If $in = unknown$, *on* remains in the same state it was. There are no clipping partners from the object represented by $T_\mathcal{B}$, hence $\mathbf{x} \in \mathcal{S}_{\mathcal{A}_\text{end}} \Leftrightarrow \mathbf{x} \in \mathcal{S}_{\mathcal{A}_\text{start}}$. Therefore, $\mathbf{x} \in \mathcal{S}_{\mathcal{A}_\text{end}}$, and (A.3) holds.

2. If $in = true$ or $in = false$, the value of *on* depends on the operator stored in $T_{\mathcal{A}_\text{end}}$. The logic directly follows equations (A.1) and (A.2), which implement (4.1) and (4.2). Thus, after evaluation, $[on = true] \Leftrightarrow [x \in \mathcal{S}_{\mathcal{A}_\text{end}}]$, which is the invariant.

Appendix B

Fiber Properties

A fiber in our implementation is a natural cubic spline through three simulation nodes on the object surface. Its parametric representation is given as a vector-valued, piecewise cubic polynomial:

$$\mathbf{f}(t) = \begin{cases} \mathbf{f}_1(2t) & 0 <= t < 0.5 \\ \mathbf{f}_2(2(t-0.5)) & 0.5 <= t < 1 \end{cases} \tag{B.1}$$

The polynomials $\mathbf{f}_1$ and $\mathbf{f}_2$ are defined as

$$\mathbf{f}_i(t) = \mathbf{a}_i + \mathbf{b}_i t + \mathbf{c}_i t^2 + \mathbf{d}_i t^3, \tag{B.2}$$

their coefficients depend linearly on the input points. For a fiber through the central node at position $\mathbf{x}$ and two of its neighbors $\mathbf{x}_1$ and $\mathbf{x}_2$, the coefficients are given by

$$\begin{aligned} \mathbf{a}_1 &= \mathbf{x}_1 \\ \mathbf{a}_2 &= \mathbf{x} \\ \mathbf{b}_1 &= -\tfrac{5}{4}\mathbf{x}_1 + \tfrac{3}{2}\mathbf{x} - \tfrac{1}{4}\mathbf{x}_2 \\ \mathbf{b}_2 &= -\tfrac{1}{2}\mathbf{x}_1 + \tfrac{1}{2}\mathbf{x}_2 \\ \mathbf{c}_1 &= 0 \\ \mathbf{c}_2 &= \tfrac{3}{4}\mathbf{x}_1 - \tfrac{3}{2}\mathbf{x} + \tfrac{3}{4}\mathbf{x}_2 \\ \mathbf{d}_1 &= \tfrac{1}{4}\mathbf{x}_1 - \tfrac{1}{2}\mathbf{x} + \tfrac{1}{4}\mathbf{x}_2 \\ \mathbf{d}_2 &= -\tfrac{1}{4}\mathbf{x}_1 + \tfrac{1}{2}\mathbf{x} - \tfrac{1}{4}\mathbf{x}_2 \end{aligned} \tag{B.3}$$

B.1 Arc Length Approximation $\tilde{l}$

The arc length of a parametric curve given by a spline $\mathbf{f}(t)$ as defined above can be numerically approximated with n samples:

$$\tilde{l}(\mathbf{f}) = \frac{1}{n}\sum_{i=0}^{n-1} \left\| \frac{\partial \mathbf{f}}{\partial t}\left(\frac{i}{n}\right) \right\| \tag{B.4}$$

Taking the first derivative of (B.2) and coefficients defined in (B.3), (B.4) can be computed easily. Since the splines in our framework are well-behaved, the approximation is sufficiently accurate even for a small number of sample points.

B.2 Gradient of $\tilde{l}$

The gradient of (B.4) is essentially a sum of gradients.

$$\nabla\tilde{l} = \frac{1}{n}\sum_{i=0}^{n-1}\nabla\|\frac{\partial\mathbf{f}_1}{\partial t}(\frac{i}{n})\| + \frac{1}{n}\sum_{i=0}^{n-1}\nabla\|\frac{\partial\mathbf{f}_2}{\partial t}(\frac{i}{n})\| \tag{B.5}$$

Thus, the gradients of the arc length with respect to the center point $\mathbf{x}$ and the endpoints $\mathbf{x}_1$ and $\mathbf{x}_2$ can be written as

$$\begin{aligned}\nabla_{\mathbf{x}_1}\tilde{l} &= \sum_{i=0}^{n-1}\frac{1}{n\|\frac{\partial\mathbf{f}_1}{\partial t}(i/n)\|}(-\frac{5}{4}+\frac{3}{4}t^2)\frac{\partial\mathbf{f}_1}{\partial t}(i/n) \qquad \text{(B.6)}\\ &\quad+\sum_{i=0}^{n-1}\frac{1}{n\|\frac{\partial\mathbf{f}_2}{\partial t}(i/n)\|}(-\frac{1}{2}+\frac{3}{2}t-\frac{3}{4}t^2)\frac{\partial\mathbf{f}_2}{\partial t}(i/n)\\ \nabla_{\mathbf{x}}\tilde{l} &= \sum_{i=0}^{n-1}\frac{1}{n\|\frac{\partial\mathbf{f}_1}{\partial t}(i/n)\|}(\frac{3}{2}-\frac{3}{2}t^2)\frac{\partial\mathbf{f}_1}{\partial t}(i/n) \qquad \text{(B.7)}\\ &\quad+\sum_{i=0}^{n-1}\frac{1}{n\|\frac{\partial\mathbf{f}_2}{\partial t}(i/n)\|}(-3t+\frac{3}{2}t^2)\frac{\partial\mathbf{f}_2}{\partial t}(i/n)\\ \nabla_{\mathbf{x}_2}\tilde{l} &= \sum_{i=0}^{n-1}\frac{1}{n\|\frac{\partial\mathbf{f}_1}{\partial t}(i/n)\|}(-\frac{1}{4}+\frac{3}{4}t^2)\frac{\partial\mathbf{f}_1}{\partial t}(i/n) \qquad \text{(B.8)}\\ &\quad+\sum_{i=0}^{n-1}\frac{1}{n\|\frac{\partial\mathbf{f}_2}{\partial t}(i/n)\|}(\frac{1}{2}+\frac{3}{2}t-\frac{3}{4}t^2)\frac{\partial\mathbf{f}_2}{\partial t}(i/n)\end{aligned}$$

B.3 Tangential Angle θ

As a measure for the curvature of a fiber, we use the angle θ between the fiber's tangents at $t=0$ and $t=1$: $\mathbf{t}_0$ and $\mathbf{t}_1$ respectively. This angle is oriented according to the surface normal in $\mathbf{x}$. Therefore, we multiply the angle with the sign of $-\mathbf{N}(\mathbf{x})*\frac{\partial\mathbf{f}}{\partial t^2}(0.5)$.

$$\kappa\approx\theta=\text{sign}(-\mathbf{N}(\mathbf{x})*\frac{\partial\mathbf{f}}{\partial t^2}(0.5))\arccos\frac{\mathbf{t}_0*\mathbf{t}_1}{\|\mathbf{t}_0\|\|\mathbf{t}_1\|} \tag{B.9}$$

B.4 Gradient of θ

Since the derivative of the sign function is zero, the gradient of (B.9) with respect to any point $\mathbf{x}_i$ is

$$\nabla_{\mathbf{x}_i}\theta=\text{sign}(-\mathbf{N}(\mathbf{x})*\frac{\partial\mathbf{f}}{\partial t^2}(0.5))\nabla_{\mathbf{x}_i}\left(\arccos\frac{\mathbf{t}_0*\mathbf{t}_1}{\|\mathbf{t}_0\|\|\mathbf{t}_1\|}\right) \tag{B.10}$$

The tangents are given by $\mathbf{t}_0 = \mathbf{b}_1$ and $\mathbf{t}_1 = \mathbf{b}_2 + 2\mathbf{c}_2 + 3\mathbf{d}_2$. Noting that the tangent to $\mathbf{f}$ at $t = 0.5$ is given by $\mathbf{t}_c = \mathbf{b}_2$, we find for the center point $\mathbf{x}$ and endpoints $\mathbf{x}_1$ and $\mathbf{x}_2$,

$$\begin{aligned}
\nabla_{\mathbf{x}_1}\theta &= c\left(\frac{-\frac{9}{2}d_1 - \mathbf{t}_c}{\|\mathbf{t}_0\|\|\mathbf{t}_1\|} + \frac{5}{4}\frac{\mathbf{t}_0 * \mathbf{t}_1}{\|\mathbf{t}_0\|^3\|\mathbf{t}_1|}\mathbf{t}_0 - \frac{1}{4}\frac{\mathbf{t}_0 * \mathbf{t}_1}{\|\mathbf{t}_0\|\|\mathbf{t}_1|^3}\mathbf{t}_1\right), \\
\nabla_{\mathbf{x}}\theta &= c\left(\frac{9d_1}{\|\mathbf{t}_0\|\|\mathbf{t}_1\|} - \frac{3}{2}\frac{\mathbf{t}_0 * \mathbf{t}_1}{\|\mathbf{t}_0\|^3\|\mathbf{t}_1|}\mathbf{t}_0 + \frac{3}{2}\frac{\mathbf{t}_0 * \mathbf{t}_1}{\|\mathbf{t}_0\|\|\mathbf{t}_1|^3}\mathbf{t}_1\right), \\
\nabla_{\mathbf{x}_2}\theta &= c\left(\frac{-\frac{9}{2}d_1 + \mathbf{t}_c}{\|\mathbf{t}_0\|\|\mathbf{t}_1\|} + \frac{1}{4}\frac{\mathbf{t}_0 * \mathbf{t}_1}{\|\mathbf{t}_0\|^3\|\mathbf{t}_1|}\mathbf{t}_0 - \frac{5}{4}\frac{\mathbf{t}_0 * \mathbf{t}_1}{\|\mathbf{t}_0\|\|\mathbf{t}_1|^3}\mathbf{t}_1\right),
\end{aligned} \tag{B.11}$$

where

$$c = \frac{\text{sign}(-\mathbf{N}(\mathbf{x}) * \frac{\partial \mathbf{f}}{\partial t^2}(0.5))}{\sqrt{1 - \cos^2(\theta)}}. \tag{B.12}$$

Appendix C

SPH Kernel Functions

A good polynomial kernel function for 3D SPH simulation is [146]:

$$w_h(d) = \begin{cases} \frac{315}{64\pi h^3}\left(1-\frac{d^2}{h^2}\right)^3 & d < h, \\ 0 & \text{otherwise.} \end{cases} \tag{C.1}$$

This since d is only used squared, (C.1) can be evaluated without using a square root. Their first and second derivatives are then:

$$w_h'(d) = \begin{cases} -\frac{945}{32\pi h^3}\frac{d}{h^2}\left(1-\frac{d^2}{h^2}\right)^2 & 0 < d < h, \\ 0 & \text{otherwise,} \end{cases} \tag{C.2}$$

$$w_h''(d) = \begin{cases} \frac{945}{32\pi h^3}\frac{1}{h^2}\left(1-\frac{d^2}{h^2}\right)\left(5\frac{d^2}{h^2}-1\right) & d < h, \\ 0 & \text{otherwise.} \end{cases} \tag{C.3}$$

For 2D simulations, the normalization has to be adjusted, yielding:

$$w_h(d) = \begin{cases} \frac{4}{\pi h^2}\left(1-\frac{d^2}{h^2}\right)^3 & d < h, \\ 0 & \text{otherwise,} \end{cases} \tag{C.4}$$

with derivatives

$$w_h'(d) = \begin{cases} -\frac{24}{\pi h^2}\frac{d}{h^2}\left(1-\frac{d^2}{h^2}\right)^2 & 0 < d < h, \\ 0 & \text{otherwise,} \end{cases} \tag{C.5}$$

$$w_h''(d) = \begin{cases} \frac{24}{\pi h^2}\frac{1}{h^2}\left(1-\frac{d^2}{h^2}\right)\left(5\frac{d^2}{h^2}-1\right) & d < h, \\ 0 & \text{otherwise.} \end{cases} \tag{C.6}$$

To avoid clumping in low resolution simulations, the spiky kernel $\hat{w}_h(d)$ can be used to compute the pressure gradient. Its gradient does not go smoothly to zero at

$d = 0$, thus preventing a force-free state where particles clump at a single position. For 3D, it is given by

$$\hat{w}_h(d) = \begin{cases} \frac{15}{\pi h^3}\left(1-\frac{d}{h}\right)^3 & d < h, \\ 0 & \text{otherwise}, \end{cases} \tag{C.7}$$

with first and second derivatives

$$\hat{w}_h'(d) = \begin{cases} -\frac{45}{\pi h^3}\frac{1}{h}\left(1-\frac{d}{h}\right)^2 & 0 < d < h, \\ 0 & \text{otherwise}, \end{cases} \tag{C.8}$$

$$\hat{w}_h''(d) = \begin{cases} \frac{90}{\pi h^3}\frac{1}{h^2}\left(1-\frac{d}{h}\right) & d < h, \\ 0 & \text{otherwise}. \end{cases} \tag{C.9}$$

In 2D, the changed normalization yields

$$\hat{w}_h(d) = \begin{cases} \frac{15}{\pi h^2}\left(1-\frac{d}{h}\right)^3 & d < h, \\ 0 & \text{otherwise}, \end{cases} \tag{C.10}$$

with first and second derivatives

$$\hat{w}_h'(d) = \begin{cases} -\frac{30}{\pi h^2}\frac{1}{h}\left(1-\frac{d}{h}\right)^2 & 0 < d < h, \\ 0 & \text{otherwise}, \end{cases} \tag{C.11}$$

$$\hat{w}_h''(d) = \begin{cases} \frac{60}{\pi h^2}\frac{1}{h^2}\left(1-\frac{d}{h}\right), & \\ 0 & \text{otherwise}. \end{cases} \tag{C.12}$$

Appendix D

Derivatives of Shape Functions

Consider a convex polyhedron with vertices at positions $\mathbf{x}_i$. Recalling (9.10) from page 126, we can write the gradient of the basis function ϕ_i as

$$\nabla\phi_i = \frac{\partial\phi_i}{\partial\mathbf{x}} = \frac{\nabla w_i \sum_k w_k - w_i \sum_k \nabla w_k}{\left(\sum_k w_k\right)^2}. \tag{D.1}$$

With the one-ring vertices of vertex i enumerated as $\mathbf{x}_j$, the gradient of the weight w_i is

$$\begin{aligned}\nabla w_i = \sum_j \Big[&\frac{\nabla c_{j,j+1} V_{i,j,j+1} - c_{j,j+1}\nabla V_{i,j,j+1}}{V_{i,j,j+1}^2} + \\ &\frac{\left(\nabla c_{i,j} V_{j-1,j+1,j} + c_{i,j}\nabla V_{j-1,j+1,j}\right) V_{i,j-1,j} V_{i,j,j+1}}{V_{i,j-1,j}^2 V_{i,j,j+1}^2} - \\ &\frac{c_{i,j} V_{j-1,j+1,j}\left(V_{i,j-1,j}\nabla V_{i,j,j+1} + \nabla V_{i,j-1,j} V_{i,j,j+1}\right)}{V_{i,j-1,j}^2 V_{i,j,j+1}^2}\Big].\end{aligned} \tag{D.2}$$

The gradient of a tetrahedron volume $V_{a,b,c}$ has the magnitude of one third the triangle area $A(a,b,c)$ and points in the direction of the triangle normal:

$$\nabla V_{a,b,c} = \frac{\left\|(\mathbf{x}_c - \mathbf{x}_a) \times (\mathbf{x}_b - \mathbf{x}_a)\right\|}{6}. \tag{D.3}$$

We define $\mathbf{d}_i = \mathbf{x} - \mathbf{x}_i$ and $\hat{\mathbf{d}}_i = \mathbf{d}_i / \|\mathbf{d}_i\|$. The gradient of the term $c_{a,b}$ is given by

$$\begin{aligned}\nabla c_{a,b} = \tfrac{1}{6}\Big[&\left(\hat{\mathbf{d}}_a * \hat{\mathbf{d}}_b\right)\left(\hat{\mathbf{d}}_a\|\mathbf{d}_b\| + \hat{\mathbf{d}}_b\|\mathbf{d}_a\|\right) - \mathbf{d}_a - \mathbf{d}_b + \\ &\frac{\arccos\left(\hat{\mathbf{d}}_a * \hat{\mathbf{d}}_b\right)}{\|\hat{\mathbf{d}}_a \times \hat{\mathbf{d}}_b\|}\left[\left(\hat{\mathbf{d}}_a \times \hat{\mathbf{d}}_b\right) \times (\mathbf{x}_b - \mathbf{x}_a)\right]\Big].\end{aligned} \tag{D.4}$$

Note that since $\frac{\arccos\left(\hat{\mathbf{d}}_a * \hat{\mathbf{d}}_b\right)}{\|\hat{\mathbf{d}}_a \times \hat{\mathbf{d}}_b\|} = \frac{\alpha}{\sin\alpha}$ and $\lim_{x\to 0}\frac{x}{\sin x} = 1$, (D.4) can be robustly evaluated in all cases.

Appendix E

Notation

Throughout this thesis, all scalar variables or functions are set in italics, while vector-valued expressions and functions, including matrices, are set in bold face. Symbols are also defined where they first appear.

Operators

$*$	dot product
$\times$	cross product
∇	gradient
$\nabla_{\mathbf{x}}$	gradient w. r. t. the components of $\mathbf{x}$
$\dot{\mathbf{x}}, \ddot{\mathbf{x}}$	first and second time derivative: $\dot{\mathbf{x}} = \frac{\partial \mathbf{x}}{\partial t}$, $\ddot{\mathbf{x}} = \frac{\partial \mathbf{x}}{\partial t^2}$
$\nabla *$	divergence
∇^2	Laplacian
$\|\cdot\|$	Euclidean norm
$\|\cdot\|_p$	pseudo-norm
$\|\cdot\|$	absolute value
det	determinant
$.^T$	matrix transpose

Geometry

$\mathcal{A}, \mathcal{B}, \mathcal{C}$	objects/patches
$\mathcal{S}$	surface
A	area
$\mathbf{c}$	centroid, or surfel center

$\mathbf{d}$	direction
$\mathbf{e}$	basis vectors
f	function
H	mean curvature
n	a number of points, vertices, or nodes
$\mathbf{N}$	normal
N	set of neighbor indices
P	set of points
r	radius
$\mathbf{t}$	tangent vector
V	volume
$\mathbf{x} = [x,y,z]$, $\mathbf{p}$, $\mathbf{q}$, $\mathbf{y}$	points in $\mathbb{R}^3$

Point-Sampled Surfaces

η	friction coefficient
ρ	optical density
$\Psi(\cdot)$	projection operator
$\mathbf{C}$	color
$\mathbf{D}$	diffusion coefficient matrix
D	isotropic diffusion coefficient
e	local error function
E	total error
$\mathbf{M}$	object to screen space mapping
$\mathbf{T}(\cdot,\cdot)$	texture function
$w(\cdot)$	weight function

Physics

ε	strain
ρ	density
τ	torque
$\mathbf{D}$	damping coefficient matrix
D	scalar damping coefficient
E	energy
$\mathbf{F}$	force

$\mathbf{K}$	stiffness matrix
K	scalar stiffness
$\mathbf{M}$	mass matrix
m	mass
T	temperature
$t, \Delta t$	time, time step
U	energy density
$\mathbf{u} = [u_x, u_y, u_z]$	displacement
$\mathbf{U}$	vector of displacements $\mathbf{U} = [u_{x1}, u_{y1}, u_{z1} \dots u_{xn}, u_{yn}, u_{zn}]^T$
$\mathbf{v}$	velocity
$\mathbf{X}$	vector of positions $\mathbf{X} = [x_1, y_1, z_1 \dots x_n, y_n, z_n]^T$

Point-Sampled Thin Shells

α	plastic creep
β	plastic yield
γ	fracture threshold
χ	plastic weakening coefficient
$\mathbf{R}$	first fundamental tensor
$\mathbf{S}$	shape operator

Visco-Elastic Fluid Simulation

$\frac{D\cdot}{D\cdot}$	material derivative
$\langle \cdot \rangle$	SPH approximation
Γ	exponent in density-pressure relationship
ζ	negative pressure scaling
λ	Poisson process parameter
μ	viscosity coefficient
ξ	XSPH parameter
ω	probability
$\mathbf{l}$	lattice point
$\mathbf{g}$	rest position
h	smoothing length
p	pressure
s	volume preserving deformation parameter

Finite Elements on Irregular Meshes

μ, κ	volume fractions
ν	Poisson ratio
ϕ	basis function
σ	stress
$\mathbf{C}$	constitutive matrix
$\mathbf{H}$	interpolation matrix
Y	Young's modulus

Appendix F

Simulation Parameters

This appendix summarizes simulation parameters used to create the example animations shown in this thesis.

F.1 Point Sampled Thin Shells

	Elasticity		Plasticity		Fracture			
Figure	K_s	K_b	α	β	γ_s	γ_b	D	Δt
7.4	25	25					0.4	0.01
7.5	1000	1000	2	0.2			0.5	0.01
7.8	1000	10000			0.02	5	0.3	0.001
7.10	100	100			0.04	5	0.4	0.001

F.2 Visco-Elastic Fluid Simulation

In all simulations from Chapter 8, $\Gamma = 1$, $\rho = 1$, and the smoothing length h is twice the particle radius.

SPH Parameters

Figure	h	K	ξ	ζ	K_e	Δt
8.4	0.04	200	0.1	0.01	39528	0.005
8.5	0.11	200	0.3	0.5	58320	0.004
8.8	0.07	20	0.1	0.01	$2.4 \cdot 10^7$	0.0004

F.3 Finite Elements on Irregular Meshes

All simulations in Chapter 9 are computed without damping (i. e. $\mathbf{D} = D = 0$), since the implicit integration scheme used provides sufficient damping by itself. Furthermore, the simulations assume a homogeneous, isotropic material, which can be described by Young's modulus Y and Poisson ratio ν alone. The density of the material is assumed to be $\rho = 1$.

Figure	Y	ν	Δt
9.3	50	0.3	0.001
9.4	50	0.3	0.005
9.9	50	0.3	0.01
9.10	$3 \cdot 10^3$	0.3	0.04

Bibliography

[1] Adams, B. *Point-Based Modeling, Animation and Rendering of Dynamic Objects*. PhD thesis, Katholieke Universiteit Leuven, 2006.

[2] Adams, B. and Dutré, P. Interactive Boolean Operations on Surfel-Bounded Solids. In *Proceedings of SIGGRAPH'03*, pages 651–656, 2003.

[3] Adams, B., Keiser, R., Pauly, M., Guibas, L., Gross, M., and Dutré, P. Efficient Raytracing of Deforming Point-Sampled Surfaces. In *Proceedings of Eurographics'05*, pages 677–684, 2005.

[4] Adams, B., Wicke, M., Dutré, P., Gross, M., and Teschner, M. Interactive 3D Painting on Point-Sampled Surfaces. In *Proceedings of the Symposium on Point-Based Graphics'04*, pages 57–66, 2004.

[5] Adamson, A. and Alexa, M. Approximating and Intersecting Surfaces from Points. In *Proceedings of the Symposium on Geometry Processing'03*, pages 230–239, 2003.

[6] Adamson, A. and Alexa, M. Ray-Tracing Point Set Surfaces. In *Proceedings of Shape Modeling International'03*, pages 272–279, 2003.

[7] Adamson, A. and Alexa, M. Approximating Bounded, Non-Orientable Surfaces from Points. In *Proceedings of Shape Modeling International'04*, pages 243–252, 2004.

[8] Adamson, A. and Alexa, M. Point-Sampled Cell Complexes. In *Proceedings of SIGGRAPH'06*, pages 671–680, 2006.

[9] Agrawala, M., Beers, A. C., and Levoy, M. 3D Painting on Scanned Surfaces. In *Proceedings of the Symposium on Interactive 3D Graphics'95*, pages 145–150, 1995.

[10] Alexa, M. and Adamson, A. On Normals and Projection Operators for Surfaces Defined by Point Sets. In *Proceedings of the Symposium on Point-Based Graphics'04*, pages 150–155, 2004.

[11] Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., and Silva, C. T. Point Set Surfaces. In *Proceedings of IEEE Visualization'01*, pages 21–28, 2001.

[12] Amenta, N. and Bern, M. Surface reconstruction by Voronoi filtering. In *Proceedings of the Symposium on Computational Geometry'98*, pages 39–48, New York, NY, USA, 1998. ACM Press.

[13] Amenta, N., Bern, M., and Kamvysselis, M. A new Voronoi-based surface reconstruction algorithm. In *Proceedings of SIGGRAPH'98*, pages 415–421, 1998.

[14] Amenta, N., Choi, S., Dey, T., and Leekha, N. A Simple Algorithm for Homeomorphic Surface Reconstruction. In *Proceedings of the Symposium on Computational Geometry'00*, pages 213–222, 2000.

[15] Amenta, N., Choi, S., Dey, T., and Leekha, N. A Simple Algorithm for Homeomorphic Surface Reconstruction. *International Journal on Computational Geometry and Applications*, 12(1–2):125–141, 2002.

[16] Amenta, N., Choi, S., and Kolluri, R. The Power Crust. In *Proceedings of the Symposium on Solid Modeling and Applications'01*, pages 249–260, 2001.

[17] Amenta, N. and Kil, Y. J. The Domain of a Point Set Surface. In *Proceedings of the Symposium on Point-Based Graphics'04*, pages 139–147, 2004.

[18] Amenta, N. and Kil, Y. J. Defining Point-Set Surfaces. In *Proceedings of SIGGRAPH'04*, pages 264–270, 2004.

[19] Arya, S. and Mount, D. M. Algorithms for Fast Vector Quantization. In *Proceedings of the IEEE Data Compression Conference '93*, pages 381–390, 1993.

[20] Baker, B. S., E. G. Coffman, j., and Rivest, R. L. Orthogonal Packings in Two Dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[21] Bancsik, Z. and Juhász, I. On the Arc Length of Parametric Cubic Curves. *Journal for Geometry and Graphics*, 3(1):1–15, 1999.

[22] Baraff, D. and Witkin, A. Large Steps in Cloth Simulation. In *Proceedings of SIGGRAPH'98*, pages 43–54, 1998.

[23] Bathe, K.-J. *Finite Element Procedures*. Prentice Hall, 1995.

[24] Baxter, W., Liu, Y., and Lin, M. C. A Viscous Paint Model for Interactive Applications. Technical Report TR04-006, University of North Carolina at Chapel Hill, 2003.

[25] Baxter, W., Scheib, V., Lin, M. C., and Manocha, D. DAB: Interactive Haptic Painting with 3D Virtual Brushes. In *Proceedings of SIGGRAPH'01*, pages 461–468, 2001.

[26] Baxter, W., Wendt, J., and Lin, M. C. IMPaSTo, A Realistic, Interactive Model for Paint. In *Proceedings of NPAR '00*, 2004.

[27] Benson, D. and Davis, J. Octree Textures. In *Proceedings of SIGGRAPH'02*, pages 785–790, 2002.

[28] Bentley, J. L. Multidimensional Binary Search Tree Used for Associative Searching. *Communications of the ACM*, 18:509–517, 1975.

[29] Bentley, J. L. and Friedman, J. H. Data Structures for Range Searching. *ACM Computing Surveys*, 11:397–409, 1979.

[30] Berman, D. F., Bartell, J. T., and Salesin, D. H. Multiresolution Painting and Compositing. In *Proceedings of SIGGRAPH'94*, pages 85–90, 1994.

[31] Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C., and Taubin, G. The Ball-Pivoting Algorithm for Surface Reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, 1999.

[32] Bielser, D., Glardon, P., Teschner, M., and Gross, M. A State Machine for Real-Time Cutting of Tetrahedral Meshes. In *Proceedings of Pacific Graphics'03*, pages 377–386, 2003.

[33] Bittar, E., Tsingos, N., and Gascuel, M.-P. Automatic Reconstruction of Unstructured 3D Data: Combining Medial Axis and Implicit Surfaces. In *Proceedings of Eurographics'95*, pages 457–468, 1995.

[34] Boissonnat, J.-D. and Oudot, S. An Effective Condition for Sampling Surfaces With Guarantees. In *Proceedings of the Symposium on Solid Modeling and Applications'04*, pages 101–112, 2004.

[35] Boissonnat, J.-D. Geometric Structures for Three-Dimensional Shape Representation. In *Proceedings of SIGGRAPH'04*, pages 266–286, 1984.

[36] Boissonnat, J.-D. Geometric structures for three-dimensional shape representation. *ACM Trans. Graph.*, 3(4):266–286, 1984.

[37] Botsch, M. and Kobbelt, L. High-Quality Point-Based Rendering on Modern GPUs. In *Proceedings of Pacific Graphics'03*, pages 335–343, 2003.

[38] Botsch, M., Hornung, A., Zwicker, M., and Kobbelt, L. High Quality Surface Splatting on Today's GPUs. In *Proceedings of the Symposium on Point-Based Graphics'05*, pages 17–24, 2005.

[39] Botsch, M., Spernat, M., and Kobbelt, L. Phong Splatting. In *Proceedings of the Symposium on Point-Based Graphics'04*, pages 25–32, 2004.

[40] Breen, D. E., House, D. H., and Wozny, M. J. Predicting the Drape of Woven Cloth Using Interacting Particles. In *Proceedings of SIGGRAPH'94*, pages 365–372, 1994.

[41] Bridson, R., Marino, S., and Fedkiw, R. Simulation of Clothing with Folds and Wrinkles. In *Proceedings of the Symposium on Computer Animation'03*, 2003.

[42] Bridson, R., Fedkiw, R., and Anderson, J. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. In *Proceedings of SIGGRAPH'02*, 2002.

[43] Carlson, M., Mucha, P. J., and Turk, G. Rigid Fluid: Animating the Interplay Between Rigid Bodies and Fluid. In *Proceedings of SIGGRAPH'04*, pages 377–384, 2004.

[44] Carlson, M., Mucha, P. J., van Horn, R. B., and Turk, G. Melting and Flowing. In *Proceedings of the Symposium on Computer Animation'02*, pages 167–174, 2002.

[45] Carr, J. C., Beatson, R. K., Cherrie, J. B., Mitchell, T. J., Fright, W. R., McCallum, B. C., and Evans, T. R. Reconstruction and Representation of 3D Objects with Radial Basis Functions. In *Proceedings of SIGGRAPH'01*, pages 67–76, 2001.

[46] Carr, N. A. and Hart, J. C. Painting detail. In *Proceedings of SIGGRAPH'04*, pages 845–852, 2004.

[47] Celniker, G. and Gossard, D. Deformable Curve and Surface Finite-Elements for Free-Form Shape Design. In *Proceedings of SIGGRAPH'91*, pages 257–266, 1991.

[48] Chadwick, J. E., Haumann, D. R., and Parent, R. E. Layered Construction for Deformable Animated Characters. In *Proceedings of SIGGRAPH'89*, pages 243–252, 1989.

[49] Chanzy, P., Devroye, L., and Zamora-Cura, C. Analysis of Range Search for Random k-d Trees. *Acta Informatica*, 37(4/5):355–383, 2001.

[50] Chen, C.-B. and He, D.-H. A Heuristic Method for Solving Triangle Packing Problem. *Journal of Zhejiang University SCIENCE*, 6A(6):565–570, 2005.

[51] Chen, H. and Fang, S. A Volumetic Approach to Interactive CSG Modeling and Rendering. In *Proceedings of the Symposium on Solid Modeling and Applications'99*, pages 318–319, 1999.

[52] Chen, W., Ren, L., Zwicker, M., and Pfister, H.-P. Hardware-Accelerated Adaptive EWA Volume Splatting. In *Proceedings of IEEE Visualization'04*, pages 67–74, 2004.

[53] Chu, N. S.-H. and Tai, C.-L. An Efficient Brush Model for Physically-Based 3D Painting. In *Proceedings of Pacific Graphics'02*, pages 413–422, 2002.

[54] Cignoni, P., Montani, C., Scopigno, R., and Rocchini, C. A General Method for Preserving Attribute Values on Simplified Meshes. In *Proceedings of IEEE Visualization'98*, pages 59–66, 1998.

[55] Cirak, F., Ortiz, M., and Schröder, P. Subdivision Surfaces: A New Paradigm for Thin-Shell Finite-Element Analysis. *International Journal for Numerical Methods in Engineering*, 47:2039–2072, 2000.

[56] Clavet, S., Beaudoin, P., and Poulin, P. Particle-Based Viscoelastic Fluid Simulation. In *Proceedings of the Symposium on Computer Animation'05*, pages 219–228, 2005.

[57] Cover, S., Ezquerra, N., O'Brien, J., Rowe, R., Gadacz, T., and Palm, E. Interactively Deformable Models for Surgery Simulation. *Computer Graphics and Applications*, 13(6):68–75, 1993.

[58] Curtis, C., amd J. Seims, S. A., Fleischer, K., and Salesin, D. Computer-Generated Watercolor. In *Proceedings of SIGGRAPH'97*, pages 421–430, 1997.

[59] Dachsbacher, C., Vogelsang, C., and Stamminger, M. Sequential Point Trees. In *Proceedings of SIGGRAPH'04*, pages 657–662, 2003.

[60] DeBry, D., Gibbs, J., Petty, D. D., and Robins, N. Painting and Rendering Textures on Unparameterized Models. In *Proceedings of SIGGRAPH'02*, pages 763–768, 2002.

[61] Debunne, G., Desbrun, M., Barr, A., and Cani, M.-P. Interactive Multiresolution Animation of Deformable Models. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation'99*, pages 133–144, 1999.

[62] Debunne, G., Desbrun, M., Cani, M.-P., and Barr, A. H. Dynamic Real-Time Deformations using Space and Time Adaptive Sampling. In *Proceedings of SIGGRAPH'01*, pages 31–36, 2001.

[63] DeRose, T. and Meyer, M. Harmonic Coordinates. Technical Memo #06-02, Pixar Animation Studios, 2006.

[64] Desbrun, M. and Gascuel, M.-P. Smoothed Particles: A New Paradigm for Animating Highly Deformable Bodies. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation'96*, pages 61–76, 1996.

[65] Dey, T. and Giesen, J. Detecting Undersampling in Surface Reconstruction. In *Proceedings of the Symposium on Geometry Processing'01*, pages 257–263, 2001.

[66] Dey, T. and Goswami, S. Tight Cocone: A Water-Tight Surface Reconstructor. *Journal of Computing, Information Science and Engineering*, 30:302–307, 2003.

[67] Dey, T. and Goswami, S. Provable Surface Reconstruction from Noisy Samples. In *Proceedings of the Symposium on Computational Geometry'04*, pages 330–339, 2004.

[68] Dey, T. and Sun, J. An Adaptive MLS Surface for Reconstruction with Guarantees. In *Proceedings of the Symposium on Geometry Processing'05*, pages 43–52, 2005.

[69] Dowsland, K. A., Vaid, S., and Dowsland, W. B. An Algorithm for Polygon Placement Using a Bottom-Left Strategy. *European Journal of Operational Research*, 141(2):371–381, 2002.

[70] Feldman, B. E., O'Brien, J. F., and Klingner, B. M. Animating Gases with Hybrid Meshes. In *Proceedings of SIGGRAPH'05*, pages 904–909, 2005.

[71] Flajolet, P. and Puech, C. Partial Match Retrieval of Multidimensional Data. *Journal of the ACM*, 33:371–407, 1986.

[72] Floater, M. S. Mean Value Coordinates. *Computer Aided Geometric Design*, 20(1):19–27, 2003.

[73] Floater, M. S., Kos, G., and Reimers, M. Mean Value Coordinates in 3D. *Computer Aided Geometric Design*, 22:623–631, 2005.

[74] Foster, N. and Metaxas, D. Realistic Animation of Liquids. *Graphics Modeling and Image Processing*, 58(5):471–483, 1996.

[75] Foster, N. and Metaxas, D. Controlling Fluid Animation. In *Proceedings of Computer Graphics International'97*, pages 178–188, 1997.

[76] Foster, N. and Metaxas, D. Modeling the Motion of Hot, Turbulent Gas. In *Proceedings of SIGGRAPH'97*, pages 181–188, 1997.

[77] Fournier, A. and Reeves, W. T. A Simple Model of Ocean Waves. In *Proceedings of SIGGRAPH'86*, pages 75–84, 1986.

[78] Friedman, J. H., Bentley, J. L., and Finkel, R. A. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions in Mathematical Software*, 3:209–226, 1977.

[79] Garg, K. and Nayar, S. K. Photorealistic rendering of rain streaks. In *Proceedings of SIGGRAPH'06*, pages 996–1002, 2006.

[80] Garland, M. and Heckbert, P. Surface Simplification Using Quadric Error Metrics. In *Proceedings of SIGGRAPH'97*, pages 209–216, 1997.

[81] Gingold, R. A. and Monaghan, J. J. Smoothed Particle Hydrodynamics: Theory and Application to Non-Spherical Stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, 1977.

[82] Goktekin, T. G., Bargteil, A. W., and O'Brien, J. F. A method for animating viscoelastic fluids. In *Proceedings of SIGGRAPH'04*, pages 463–468, 2004.

[83] Goldfeather, J., Hultquist, J. P. M., and Fuchs, H. Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System. In *Proceedings of SIGGRAPH'86*, pages 107–116, 1986.

[84] Gourret, J.-P., Thalmann, N. M., and Thalmann, D. Simulation of Object and Human Skin Formations in a Grasping Task. In *Proceedings of SIGGRAPH'89*, pages 21–30, 1989.

[85] Green, A. E. and Zerna, W. *Theoretical Elasticity*. Oxford University Press, 1968.

[86] Green, M. and Sun, H. Interactive Animation: A Language and System for Procedural Modeling and Motion. *Computer Graphics and Applications*, 8(6):52–64, 1988.

[87] Gregory, A. D., Ehmann, S. A., and Lin, M. C. inTouch: Interactive Multiresolution Modeling and 3D Painting with a Haptic Interface. In *Proceedings of the IEEE Conference in Virtual Reality'00*, pages 45–52, 2000.

[88] Grimm, S., Bruckner, S., Kanitsar, A., and Gröller, E. VOTS: VOlume doTS as a Point-Based Representation of Volumetric Data. In *Proceedings of Eurographics'04*, pages 668–661, 2004.

[89] Grinspun, E., Hirani, A. N., Desbrun, M., and Schröder, P. Discrete Shells. In *Proceedings of the Symposium on Computer Animation'03*, pages 62–67, 2003.

[90] Grinspun, E., Krysl, P., and Schröder, P. CHARMS: A Simple Framework for Adaptive Simulation. In *Proceedings of SIGGRAPH'02*, pages 281–290, 2002.

[91] Gross, D. B. M. Interactive Simulation of Surgical Cuts. In *Proceedings of Pacific Graphics'00*, pages 116–125, 2000.

[92] Gross, M. Getting to the Point. *Computer Graphics and Applications*, 26(5):96–99, 2006.

[93] Gross, M. and Pfister, H.-P., editors. *Point-Based Graphics*. in press, 2007.

[94] Grossmann, J. P. and Dally, W. J. Point Sample Rendering. In *Proceedings of the Eurographics Rendering Workshop'98*, pages 181–192, 1998.

[95] Guennebaud, G. and Gross, M. Algebraic Point Set Surfaces. In *Proceedings of SIGGRAPH'07*, 2007. to appear.

[96] Guo, X., Li, X., Bao, Y., Gu, X., and Qin, H. Meshless Thin-Shell Simulation Based on Global Conformal Parameterization. *IEEE Transactions on Visualization and Computer Graphics*, 12(3):375–385, 2006.

[97] Hale, J. G. Texture Re-Mapping for Decimated Polygonal Meshes. Master's thesis, Edinburgh University, 1998.

[98] Hall-Holt, O. and Rusinkiewicz, S. Stripe Boundary Codes for Real-Time Structured-Light Range Scanning of Moving Objects. In *Proceedings of the IEEE International Conference on Computer Vision'01*, pages 359–366, 2001.

[99] Hanrahan, P. and Haeberli, P. Direct WYSIWYG Painting and Texturing on 3D Shapes. In *Proceedings of SIGGRAPH'90*, pages 215–223, 1990.

[100] Hin, A. J. S. and Post, F. H. Visualization of turbulent flow with particles. In *Proceedings of IEEE Visualization'93*, pages 46–53, 1993.

[101] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. Surface reconstruction from unorganized points. In *Proceedings of SIGGRAPH'92*, pages 71–78, 1992.

[102] Hormann, K. and Floater, M. S. Mean value coordinates for arbitrary planar polygons. *Transactions on Graphics*, 25(4):1424–1441, 2006.

[103] Horn, B. K. P. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America*, 4:629–642, 1987.

[104] Igarashi, T. and Cosgrove, D. Adaptive Unwrapping for Interactive Texture Painting. In *Proceedings of the Symposium on Interactive 3D Graphics'01*, pages 209–216, 2001.

[105] Irving, G., Teran, J., and Fedkiw, R. Tetrahedral and Hexahedral Invertible Finite Elements. *Graphical Models*, 68(2):66–89, 2006.

[106] Johnson, D., Thompson, T. V., Kaplan, M., Nelson, D. D., and Cohen, E. Painting Textures with a Haptic Interface. In *Proceedings of the IEEE Conference in Virtual Reality'99*, pages 282–285, 1999.

[107] Ju, T., Schaefer, S., Warren, J., and M.Desbrun. Geometric Construction of Coordinates for Convex Polyhedra using Polar Duals. In *Proceedings of the Symposium on Geometry Processing'05*, pages 181–186, 2005.

[108] Ju, T., Liepa, P., and Warren, J. A general geometric construction of coordinates in a convex simplicial polytope. *Computer Aided Geometric Design*, 2007. to appear.

[109] Ju, T., Schaefer, S., and Warren, J. Mean value coordinates for closed triangular meshes. In *Proceedings of SIGGRAPH'05*, pages 561–566, 2005.

[110] Kanade, T., Yoshida, A., Oda, K., Kano, H., and Tanaka, M. A Stereo Machine for Video-Rate Dense Depth Mapping and Its New Applications. In *Computer Vision and Pattern Recognition (CVPR)'96*, pages 196–202, 1996.

[111] Keiser, R. *Meshless Lagrangian Methods for Physics-Based Animations of Solids and Fluids*. PhD thesis, ETH Zurich, 2006.

[112] Keiser, R., Adams, B., Gasser, D., Bazzi, P., Dutré, P., and Gross, M. A Unified Lagrangian Approach to Solid-Fluid Animation. In *Proceedings of the Symposium on Point-Based Graphics'05*, pages 125–148, 2005.

[113] Keiser, R., Müller, M., Heidelberger, B., Teschner, M., and Gross, M. Contact Handling for Deformable Point-Based Objects. In *Proceedings of Vision, Modeling, and Visualization (VMV)'04*, pages 339–347, 2004.

[114] Kharevych, L., Weiwei, Tong, Y., Kanso, E., Marsden, J. E., Schröder, P., and Desbrun, M. Geometric, Variational Integrators for Computer Animation. In *Proceedings of SCA '06*, pages 43–51, 2006.

[115] Kim, L., Sukhatme, G. S., and Desbrun, M. Haptic Editing of Decoration and Material Properties. In *Proceedings of the Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems '03*, pages 213–220, 2003.

[116] Kim, T., Henson, M., and Lin, M. C. A Hybrid Algorithm for Modeling Ice Formation. In *Proceedings of the Symposium on Computer Animation '04*, pages 305–314, 2004.

[117] Kim, T. and Lin, M. C. Visual Simulation of Ice Crystal Growth. In *Proceedings of the Symposium on Computer Animation '03*, pages 86–97, 2003.

[118] Klingner, B. M., Feldman, B. A., Chentanez, N., and O'Brien, J. F. Fluid Animation with Dynamic Meshes. In *Proceedings of SIGGRAPH'06*, pages 820–825, 2006.

[119] Koch, R. M., Gross, M. H., Carls, F. R., von Büren, D. F., Frankhauer, G., and Parish, Y. I. H. Simulating Facial Surgery Using Finite Element Models. In *Proceedings of SIGGRAPH'96*, pages 421–428, 1996.

[120] Kolluri, R. Provably Good Moving Least Squares. In *Prodeedings of the Symposium on Discrete Algorithms'05*, pages 1008–1017, 2005.

[121] Kubelka, P. and Munk, F. Ein Beitrag zur Optik der Farbanstriche. *Zeitschrift für technische Physik*, 12(11a):593–601, 1931.

[122] Lamousin, H. and Warren N. Waggenspack, j. Nesting of Two-Dimensional Irregular Parts Using a Shape Reasoning Heuristic. *Computer Aided Design*, 29(3):221–238, 1997.

[123] Lee, D. T. and Wong, C. K. Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees. *Acta Informatica*, 9(1):23–29, 1977.

[124] Lee, H., Kim, L., Meyer, M., and Desbrun, M. Meshes on fire. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation*, pages 75–84, 2001.

[125] Lee, J. Simulating Oriental Black-Ink Painting. *Computer Graphics and Applications*, 19(3):74–81, 1999.

[126] Lefebvre, S. and Hoppe, H. Perfect Spatial Hashing. In *Proceedings of SIGGRAPH'06*, pages 579–588, 2006.

[127] Levin, D. The Approximation Power of Moving Least Squares. *Mathematics of Computation*, 67:1517–1531, 1998.

[128] Levin, D. Mesh-Independent Surface Interpolation. In Brunett, Hamann, and Mueller, editors, *Geometric Modeling for Scientific Visualization*, pages 37–49. Springer, 2003.

[129] Levoy, M. and Whitted, T. The Use of Points as a Display Primitive. Technical Report TR85-022, University of North Carolina at Chapel Hill, January 1985.

[130] Lewis, J. P., Cordner, M., and Fong, N. Pose-Space Deformation: A unified approach to Shape Interpolation and Skeleton-Driven Deformation. In *Proceedings of SIGGRAPH'00*, pages 165–172, 2000.

[131] Lischinski, D. and Rappoport, A. Image-Based Rendering for Non-Diffuse Synthetic Scenes. In *Proceedings of the Eurographics Rendering Workshop'98*, pages 301–314, 1998.

[132] Liu, G.-R. and Liu, M. *Smoothed Particle Hydrodynamics*. World Scientific, 2003.

[133] Lorensen, W. and Kline, H. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of SIGGRAPH'87*, pages 163–170, 1987.

[134] Losasso, F., Gibou, F., and Fedkiw, R. Simulating Water and Smoke with an Octree Data Structure. In *Proceedings of SIGGRAPH'04*, pages 457–462, 2004.

[135] Losasso, F., Irving, G., and Guendelman, E. Melting and Burning Solids into Liquids and Gases. *IEEE Transactions on Visualization and Computer Graphics*, 12(3):343–352, 2006.

[136] Losasso, F., Irving, G., Guendelman, E., and Fedkiw, R. Melting and Burning Solids into Liquids and Gases. *IEEE Transactions on Visualization and Computer Graphics*, 12(3):343–352, 2006.

[137] Lucy, L. B. A Numerical Approach to the Testing of the Fission Hypothesis. *The Astronomical Journal*, 82(12):1013–1024, 1977.

[138] Maruya, M. Texture Map Generation from Object-Surface Data. In *Proceedings of Eurographics'95*, pages 397–405, 1995.

[139] McGuire, M. and Fein, A. Real-time Rendering of Cartoon Smoke and Clouds. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering'06*, pages 21–26, 2006.

[140] Meyer, M., Desbrun, M., Schröder, P., and Barr, A. Discrete Differential Geometry Operators for Triangulated 2-Manifolds. VisMath, 2002.

[141] Molino, N., Bao, Z., and Fedkiw, R. A Virtual Node Algorithm for Changing Mesh Topology During Simulationode algorithm for changing mesh topology during simulation. In *Proceedings of SIGGRAPH'04*, pages 385–392, 2004.

[142] Monaghan, J. J. On the Problem of Penetration in Particle Methods. *Journal of Computational Physics*, 82:1–15, 1989.

[143] Monaghan, J. J. Smoothed Particle Hydrodynamics. *Annu. Rev. Astron. Physics*, 30:543, 1992.

[144] Monaghan, J. J. Simulating free surface flows with SPH. *J. Comput. Phys.*, 110(2):399–406, 1994.

[145] Monaghan, J. J. Smoothed Particle Hydrodynamics. *Reports on Progress in Physics*, 68:1703–1759, 2005.

[146] Müller, M., Charypar, D., and Gross, M. Particle-Based Fluid Simulation for Interactive Applications. In *Proceedings of the Symposium on Computer Animation'03*, pages 154–159, 2003.

[147] Müller, M. and Gross, M. Interactive Virtual Materials. In *Proceedings of Graphics Interface'04*, pages 239–246, 2004.

[148] Müller, M., Heidelberger, B., Teschner, M., and Gross, M. Meshless Deformations Based on Shape Matching. In *Proceedings of SIGGRAPH'05*, pages 471–478, 2005.

[149] Müller, M., Keiser, R., Nealen, A., Pauly, M., Gross, M., and Alexa, M. Point-Based Animation of Elastic, Plastic, and Melting Objects. In *Proceedings of the Symposium on Computer Animation'04*, pages 141–151, 2004.

[150] Müller, M., Solenthaler, B., Keiser, R., and Gross, M. Particle-Based Fluid-Fluid Interaction. In *Proceedings of the Symposium on Computer Animation'05*, pages 237–244, 2005.

[151] Müller, M., Teschner, M., and Gross, M. Physically-Based Simulation of Objects Represented by Surface Meshes. In *Proceedings of Computer Graphics International'04*, pages 26–33, 2004.

[152] Müller, M., Teschner, M., Heidelberger, B., and Gross, M. Interaction of Fluids with Deformable Objects. In *Proceedings of the Conference on Computer Animation and Social Agents'04*, pages 159–171, 2004.

[153] Nealen, A., Müller, M., Keiser, R., Boxermann, E., and Carlson, M. Physically Based Deformable Models in Computer Graphics. In *Proceedings of Eurographics'05*, pages 71–94, 2005.

[154] O'Brien, J. F., Bargteil, A. W., and Hodgins, J. K. Graphical Modeling and Animation of Ductile Fracture. In *Proceedings of SIGGRAPH'02*, pages 291–294, 2002.

[155] O'Brien, J. F. and Hodgins, J. K. Graphical Modeling and Animation of Brittle Fracture. In *Proceedings of SIGGRAPH'99*, pages 137–146, 1999.

[156] Ohtake, Y., Belyaev, A., Alexa, M., Turk, G., and Seidel, H.-P. Multi-Level Partition of Unity Implicits. In *Proceedings of SIGGRAPH'03*, pages 463–470, 2003.

[157] Pauly, M. and Gross, M. Spectral Processing of Point-Sampled Geometry. In *Proceedings of SIGGRAPH'01*, pages 379–386, 2001.

[158] Pauly, M., Gross, M., and Kobbelt, L. Efficient Simplification of Point-Sampled Surfaces. In *Proceedings of IEEE Visualization'02*, pages 162–170, 2002.

[159] Pauly, M., Keiser, R., Adams, B., Dutré, P., and Gross, M. Meshless Animation of Fracturing Solids. In *Proceedings of SIGGRAPH'05*, pages 957–964, 2005.

[160] Pauly, M., Keiser, R., and Gross, M. Multi-Scale Feature Extraction on Point-Sampled Surfaces. In *Proceedings of Eurographics'03*, pages 281–290, 2003.

[161] Pauly, M., Keiser, R., Kobbelt, L., and Gross, M. Shape Modeling with Point-Sampled Geometry. In *Proceedings of SIGGRAPH'03*, pages 281–290, 2003.

[162] Pauly, M., Pai, D. K., and Guibas, L. J. Quasi-Rigid Objects in Contact. In *Proceedings of the Symposium on Computer Animation'04*, pages 109–119, 2004.

[163] Pentland, A., Darrell, T., Turk, M., and Huang, W. A Simple, Real-Time Range Camera. In *Computer Vision and Pattern Recognition (CVPR)'98*, pages 256–261, 1989.

[164] Perry, C. H. and Picard, R. W. Synthesizing Flames and Their Spreading. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation'94*, pages 105–117, 1994.

[165] Pfister, H., Zwicker, M., van Baar, J., and Gross, M. Surfels: Surface Elements as Rendering Primitives. In *Proceedings of SIGGRAPH'00*, pages 335–342, 2000.

[166] Platt, S. M. and Badler, N. I. Animating Facial Expressions. In *Proceedings of SIGGRAPH'81*, pages 245–252, 1981.

[167] Proesmans, M., van Gool, L., and Defoort, F. Reading Between the Lines — A Method for Extracting Dynamic 3D with Texture. In *Proceedings of the IEEE International Conference on Computer Vision'98*, pages 1081–1086, 1998.

[168] Quinlan, J. R. Learning Efficient Classification Procedures and their Applications to Chess Endgames. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning*, 1983.

[169] Rappoport, A. and Spitz, S. Interactive Boolean Operations for Conceptual Design of 3D Solids. In *Proceedings of SIGGRAPH'97*, pages 269–278, 1997.

[170] Reeves, W. T. Particle Systems: A Technique for Modeling a Class of Fuzzy Objects. In *Proceedings of SIGGRAPH'83*, pages 91–108, 1983.

[171] Reuter, P., Schmitt, B., Pasko, A., and Schlick, C. Interactive Solid Texturing using Point-based Multiresolution Representations. In *Proceedings of the Winter School on Computer Graphics'04*, pages 363–370, 2004.

[172] Reynolds, C. W. Flocks, Herds and Schools: A Distributed Behavioral Model. In *Proceedings of SIGGRAPH'87*, pages 25–34, 1987.

[173] Rossignac, J., Megahed, A., and Schneider, B. O. Interactive Inspection of Solids: Cross-sections and Interferences. In *Proceedings of SIGGRAPH'92*, pages 353–360, 1992.

[174] Roth, M., Gross, M., Turello, S., and Carls, F. R. A Bernstein-Bézier Based Approach to Soft Tissue Modeling. In *Proceedings of Eurographics'98*, pages 285–294, 1998.

[175] Rusinkiewicz, S. and Levoy, M. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of SIGGRAPH'00*, pages 343–352, 2000.

[176] Ruspini, D. C., Kolarov, K., and Khatib, O. The Haptic Display of Complex Graphical Environments. In *Proceedings of SIGGRAPH'97*, pages 345–352, 1997.

[177] Sagar, M. A., Bullivant, D., Mallinson, G. D., and Hunter, P. J. A Virtual Environment and Model of the Eye for Surgical Simulation. In *Proceedings of SIGGRAPH'94*, pages 205–212, 1994.

[178] Samet, H. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2005.

[179] Sato, Y., Wheeler, M. D., and Ikeuchi, K. Object Shape and Reflectance Modeling. In *Proceedings of SIGGRAPH'97*, pages 379–387, 1997.

[180] Schall, O., Belyaev, A., and Seidel, H.-P. Robust Filtering of Noisy Scattered Point Data. In *Proceedings of the Symposium on Point-Based Graphics'05*, pages 71–77, 2005.

[181] Shewchuck, J. What Is a Good Linear Finite Element? Interpolation, Conditioning, and Quality Measures. In *Proceedings of the International Meshing Roundtable '02*, pages 115–126, 2002.

[182] Shewchuck, J. What Is a Good Linear Finite Element? Interpolation, Conditioning, Anisotropy, and Quality Measures. unpublished extended version, 2002.

[183] Shewchuck, J. Updating and Constructing Constrained Delaunay and Constrained Regular Triangulations by Flips. In *Proceedings of the Symposium on Computational Geometry'03*, pages 181–190, 2003.

[184] Shi, L. and Yu, Y. Visual Smoke Simulation with Adaptive Octree Refinement. Technical Report UIUCDCS-R-2002-2311, University of Illinois at Urbana-Champaign, 2002.

[185] Sigg, C. and Hadwiger, M. Fast Third-Order Texture Filtering. In Pharr, M., editor, *GPU Gems 2*, pages 313–329. Addison-Wesley, 2005.

[186] Sims, K. Particle Animation and Rendering Using Data Parellel communication. In *Proceedings of SIGGRAPH'90*, pages 405–413, 1990.

[187] Singh, K. and Kokkevis, E. Skinning Characters using Surface-Oriented Free-Form Deformations. In *Proceedings of Graphics Interface*, pages 35–42, 2000.

[188] Small, D. Simulating Watercolor by Modeling Diffusion, Pigment, and Paper Fibers. In *Proceedings of the SPIE*, volume 1460, pages 140–146, 1991.

[189] Soucy, M., Godin, G., and Rioux, M. A Texture-Mapping Approach for the Compression of Colored 3D Triangulations. *The Visual Computer*, 12(10):503–514, 1996.

[190] Stam, J. Stable Fluids. In *Proceedings of SIGGRAPH'99*, pages 121–128, 1999.

[191] Stam, J. and Fiume, E. Depicting Fire and Other Gaseous Phenomena. In *Proceedings of SIGGRAPH'95*, pages 129–136, 1995.

[192] Steinemann, D., Harders, M., Gross, M., and Szekely, G. Hybrid Cutting of Deformable Solids. In *Proceedings of the IEEE Conference in Virtual Reality'06*, 2006.

[193] Steinemann, D., Otaduy, M. A., and Gross, M. Fast Arbitrary Splitting of Deforming Objects. In *Proceedings of the Symposium on Computer Animation'06*, pages 63–72, 2006.

[194] Stewart, N., Leach, G., and John, S. An Improved Z-Buffer CSG Rendering Algorithm. In *Proceedings of the Graphics Hardware Workshop'98*, pages 25–30, 1998.

[195] Strassmann, S. Hairy Brushes. In *Proceedings of SIGGRAPH'86*, pages 225–232, 1986.

[196] Sukumar, N. and Malsch, E. A. Recent Advances in the Construction of Polygonal Finite Element Interpolants. *Archives of Computational Methods in Engineering*, 13(1):129–163, 2006.

[197] Szeliski, R. and Tonnesen, D. Surface Modeling with Oriented Particle Systems. In *Proceedings of SIGGRAPH'92*, pages 185–194, 1992.

[198] Terzopoulos, D., Platt, J., and Fleischer, K. Heating and Melting Deformable Models (From Goop to Glop). In *Proceedings of Graphics Interface'89*, pages 219–226, 1989.

[199] Terzopoulos, D., Platt, J., Barr, A., and Fleischer, K. Elastically Deformable Models. In *Proceedings of SIGGRAPH'87*, pages 205–214, 1987.

[200] Teschner, M., Heidelberger, B., Müller, M., and Gross, M. A Versatile and Robust Model for Geometrically Complex Deformable Solids. In *Proceedings of Computer Graphics International'04*, pages 312–319, 2004.

[201] Teschner, M., Heidelberger, B., Müller, M., Pomeranerts, D., and Gross, M. Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *Proceedings of Vision, Modeling, and Visualization (VMV)'03*, pages 47–54, 2003.

[202] Tonnesen, D. Modeling Liquids and Solids Using Thermal Particles. In *Proceedings of Graphics Interface'91*, pages 255–262, 1991.

[203] Tonnesen, D. Spatially Coupled Particle Systems. Chapter in SIGGRAPH '92 Course Notes, Course #16: Particle System Modeling, Animation, Physically Based Techniques, 1992.

[204] Tonnesen, D. *Dynamically Coupled Particle Systems for Geometric Modeling, Reconstruction, and Animation*. PhD thesis, University of Toronto, 1998.

[205] Tu, X. and Terzopoulos, D. Artificial Fishes: Physics, Locomotion, Perception, Behavior. In *Proceedings of SIGGRAPH'94*, pages 43–50, 1994.

[206] van Laerhoven, T., Loesenborgs, J., and van Reeth, F. Real-Time Watercolor Painting on a Distributed Paper Model. In *Proceedings of Computer Graphics International'04*, pages 640–643, 2004.

[207] Verlet, L. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159(1):98–103, 1967.

[208] Wachspress, E. L. A Rational Basis for Function Approximation. In *Lecture Notes in Mathematics*, volume 228, pages 223–252, New York, 1971. Springer.

[209] Wachspress, E. L. *A Rational Finite Element Basis*. Academic Press, New York, 1975.

[210] Wald, I. and Seidel, H.-P. Interactive Ray Tracing of Point Based Models. In *Proceedings of the Symposium on Point-Based Graphics'05*, 2005.

[211] Warren, J. Barycentric Coordinates for Convex Polytopes. *Advances in Computational Mathematics*, 6:97–108, 1996.

[212] Wendland, H. Piecewise Polynomial, Positive Definite and Compactly Supported Radial Functions of Minimal Degree. *Advances in Computational Mathematics*, 4:389–396, 1995.

[213] Wess, S., Althoff, K.-D., and Derwand, G. Using k-d Trees to Improve the Retrieval Step in Case-Based Reasoning. In *Selected papers from the European Workshop on Case Based Reasoning '93*, pages 167–181, 1994.

[214] Weyrich, T., Heinzle, S., Aila, T., Fasnacht, D., Oetiker, S., Botsch, M., Flaig, C., Mall, S., Rohrer, K., Felber, M., Haeslin, H., and Gross, M. A Hardware Architecture for Surface Splatting. In *Proceedings of SIGGRAPH'07*, 2007. to appear.

[215] Weyrich, T., Pauly, M., Keiser, R., Heinzle, S., Scandella, S., and Gross, M. Post-Processing of Scanned 3D Surface Data. In *Proceedings of the Symposium on Point-Based Graphics'04*, pages 85–94, 2004.

[216] Wicke, M., Botsch, M., and Gross, M. A Finite Element Method on Convex Polyhedra. In *Proceedings of Eurographics'07*, 2007. to appear.

[217] Wicke, M., Hatt, P., Pauly, M., Müller, M., and Gross, M. Versatile Virtual Materials Using Implicit Connectivity. In *Proceedings of the Symposium on Point-Based Graphics'06*, pages 137–144, 2006.

[218] Wicke, M., Olibet, S., and Gross, M. Conversion of Point-Sampled Models to Textured Meshes. In *Proceedings of the Symposium on Point-Based Graphics'05*, pages 119–124, 2005.

[219] Wicke, M., Steinemann, D., and Gross, M. Efficient Animation of Point-Based Thin Shells. In *Proceedings of Eurographics'05*, pages 667–676, 2005.

[220] Wicke, M., Teschner, M., and Gross, M. CSG Tree Rendering of Point-Sampled Objects. In *Proceedings of Pacific Graphics'04*, pages 160–168, 2004.

[221] Wiegand, T. F. Interactive Rendering of CSG Models. *Computer Graphics Forum*, 15(4):249–261, 1996.

[222] Wong, H. T. F. and Ip, H. H. S. Virtual Brush: A Model-Based Synthesis of Chinese Calligraphy. *Computers and Graphics*, 24(1):99–113, 2000.

[223] Wu, J. and Kobbelt, L. Optimized Sub-Sampling of Point Sets for Surface Splatting. In *Proceedings of Eurographics'04*, pages 643–652, 2004.

[224] Wyckoff, R. W. G. *Crystal Structures*. Wiley & Sons, 1963.

[225] Xu, S., Tang, M., Lau, F., and Pan, Y. A Solid Model Based Virtual Hairy Brush. In *Proceedings of Eurographics'02*, pages 299–308, 2002.

[226] Xu, S., Lau, F. C., Tang, F., and Pan, Y. Advanced Design for a Realistic Virtual Brush. In *Proceedings of Eurographics'03*, pages 533–542, 2003.

[227] Yeh, J., Lien, T., and Ouhyoung, M. On the Effects of Haptic Display in Brush and Ink Simulation for C hinese Painting and Calligraphy. In *Proceedings of Pacific Graphics'02*, pages 439–441, 2002.

[228] Zhu, Y. and Bridson, R. Animating Sand as a Fluid. In *Proceedings of SIGGRAPH'05*, pages 965–972, 2005.

[229] Zwicker, M., Pauly, M., Knoll, O., and Gross, M. Pointshop 3D: An Interactive System for Point-Based Surface Editing. In *Proceedings of SIGGRAPH'02*, pages 322–329, 2002.

[230] Zwicker, M., Pfister, H.-P., van Baar, J., and Gross, M. EWA Volume Splatting. In *Proceedings of IEEE Visualization'01*, pages 29–36, 2001.

[231] Zwicker, M., Pfister, H., van Baar, J., and Gross, M. Surface Splatting. In *Proceedings of SIGGRAPH'01*, pages 371–378, 2001.

[232] Zwicker, M., Pfister, H., van Baar, J., and Gross, M. EWA Splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, 2002.

[233] Zwicker, M., Räsänen, J., Botsch, M., Dachsbacher, C., and Pauly, M. Perspective Accurate Splatting. In *Proceedings of Graphics Interface'04*, pages 247–254, 2004.

Copyrights

Some of the models used to create animations and pictures shown in this thesis are copyrighted material. They are used courtesy of their respective owners.

Item	used in Fig.	Copyright by
Max Plank model	3.3 a, 7.1, 7.5	Max Planck Institut für Informatik Saarbrücken
Igea model (geometry only)	3.3 b, 6.6	Cyberware, Inc.
Femur model	4.8	Cyberware, Inc.
Bunny model (geometry only)	5.8, 5.9, 5.10 a, 5.12, 6.3, 6.4, 8.4, 9.10	Stanford Computer Graphics Laboratory
Dragon model (geometry only)	5.1, 5.5 a, 5.7, 5.10 b+c, 6.5	Stanford Computer Graphics Laboratory
Mask model	7.4	Cyberware, Inc.
Lenna texture	7.8	Playboy
Balloon photo	7.10	Chris DiBona

Curriculum Vitae

Martin Wicke

born 3. May 1979 in Frankenthal (Pfalz), Germany
citizen of Germany

Education

Jun. 2003 - Jul. 2007	Ph.D. student at the Computer Graphics Laboratory, ETH Zurich, Switzerland
Aug. 2005 - Sep. 2005	Research Visit at the Geometric Computing Group, Stanford University, USA
May 2003	*Diplom* in Computer Science (Dipl. Inform.)
Oct. 1997 - May 2003	Studies in Computer Science, Universität Kaiserslautern, Germany
Apr. 2000 - Oct. 2000	Studies in Computer Science, University of Western Australia, Perth, Australia
Aug. 1997	*Abitur*, Wilhelm von Humboldt Gymnasium, Ludwigshafen, Germany

Publications

Martin Wicke, Mario Botsch, Markus Gross. A Finite Element Method on Convex Polyhedra. In *Proceedings of Eurographics 2007*.

Mario Botsch, Mark Pauly, Martin Wicke, Markus Gross. Adaptive Space Deformations Based on Rigid Cells. In *Proceedings of Eurographics 2007*.

Martin Wicke, Richard Keiser, Markus Gross. Fluid Simulation. Chapter in *Point-Based Graphics*, edited by Markus Gross and Hanspeter Pfister. Elsevier/Morgan Kaufmann.

Matthias Zwicker, Martin Wicke. 3D Editing and Painting. Chapter in *Point-Based Graphics*, edited by Markus Gross and Hanspeter Pfister. Elsevier/Morgan Kaufmann.

Bernd Bickel, Martin Wicke, Markus Gross. Adaptive Simulation of Electrical Discharges. In *Proceedings of Vision, Modeling, and Visualization 2006*.

Martin Wicke, Hermes Lanker, Markus Gross. Untangling Cloth with Boundaries. In *Proceedings of Vision, Modeling, and Visualization 2006*.

Martin Wicke, Philipp Hatt, Mark Pauly, Matthias Müller, Markus Gross. Versatile Virtual Materials Using Implicit Connectivity. In *Proceedings of the Symposium on Point-Based Graphics 2006*.

Martin Wicke, Denis Steinemann, Markus Gross. Efficient Animation of Point-Sampled Thin Shells. In *Proceedings of Eurographics 2005*.

Martin Wicke, Sandro Olibet, Markus Gross. Conversion of Point-Sampled Models to Textured Meshes. In *Proceedings of the Symposium on Point-Based Graphics 2005*.

Martin Wicke, Matthias Teschner, Markus Gross. CSG Tree Rendering of Point-Sampled Objects. In *Proceedings of Pacific Graphics 2004*.

Bart Adams, Martin Wicke, Phil Dutré, Markus Gross, Mark Pauly, Matthias Teschner. Interactive 3D Painting on Point-Sampled Objects. In *Proceedings of the Symposium on Point-Based Graphics 2004*.

Martin Wicke. Reconstruction of Scenes using High Dynamic Range Panoramic Scans. *Diplomarbeit*, Universität Kaiserslautern, 2003.