Point-Based Computer Graphics SIGGRAPH 2004 Course Notes

Marc Alexa, Darmstadt University of Technology
Markus Gross, ETH Zurich
Mark Pauly, Stanford University
Hanspeter Pfister, Mitsubishi Electric Research Laboratories
Marc Stamminger, University of Erlangen-Nuremberg
Matthias Zwicker, Massachusetts Institute of Technology

Abstract

This course introduces points as a powerful and versatile graphics primitive. Speakers present their latest concepts for the acquisition, representation, modeling, processing, and rendering of point sampled geometry along with applications and research directions. We describe algorithms and discuss current problems and limitations, covering important aspects of point based graphics.

CONTENTS

SECTION 1

INTRODUCTION

Point primitives have experienced a major "renaissance" in recent years, and considerable research has been devoted to efficient representation, modeling, processing, and rendering of point-sampled geometry. There are two main reasons for this new interest in points: First, we have witnessed a dramatic increase in the polygonal complexity of computer graphics models. The overhead of managing, processing, and manipulating very large polygonal-mesh connectivity information has led many leading researchers to question the future utility of polygons as the fundamental graphics primitive. Second, modern 3D digital photography and 3D scanning systems acquire both the geometry and the appearance of complex, real-world objects. These techniques generate huge volumes of point samples, which constitute discrete building blocks of 3D object geometry and appearance, much as pixels are the digital elements for images.

This course presents the latest research results in point-based computer graphics. After an overview of the key research issues, affordable 3D scanning devices are discussed, and novel concepts for mathematical representation of point-sampled shapes are presented. The course describes methods for high-performance and high-quality rendering of point models, including advanced shading, antialiasing, and transparency. It also presents efficient data structures for hierarchical rendering on modern graphics processors and summarizes methods for geometric processing, filtering, and resampling of point models. Other topics include: a framework for shape modeling of point-sampled geometry, including Boolean operations and free-form deformations, and Pointshop3D, an open-source framework that facilitates design of new algorithms for point-based graphics.

1.1 PREREQUISITES

The course assumes familiarity with the standard computer graphics techniques for surface representation, modeling, and rendering. No previous knowledge about point-based methods is required.

1.2 INTENDED AUDIENCE

The course is intended for computer graphics researchers who would like to learn more about point based techniques. They will obtain a state-of-the-art overview of the use of points to solve fundamental computer graphics problems such as surface data acquisition, representation, processing, modeling, and rendering. With this course, we hope to stimulate research in this exciting field. To this aim we will also introduce Pointshop3D, an open source software package that is publicly available as a test-bed for point-based computer graphics research.

1.3 COURSE SYLLABUS

Course Introduction. Gross (15 minutes)

Acquisition of Point Sampled Geometry and Appearance. Pfister (60 minutes)
- System overview
- Silhouette extraction
- View-dependent hull calculation
- Point-sampled data structure
- View-dependent shading
- Results

Point Based Surface Representations. Alexa (60 minutes)
- Introduction & basics
- Delaunay subcomplex methods
- Approximating or interpolating implicits
- Projection-based methods

Algorithms for High Quality Point Rendering. Zwicker (60 minutes)
- Signal processing basics and antialiasing
- Resampling filters for point rendering
- Surface splatting: a splatting algorithm with antialiasing
- Hardware acceleration

Effiecient Data Structures. Stamminger (45 minutes)
- The QSplat bounding spheres hierarchy
- Optimizing the memory layout for hardware accelerated rendering
- Sequential point trees
- Implementation and results

Processing, Sampling and Filtering of Point Models. Gross (30 minutes)
Spectral processing of point sampled geometry
Decomposition and patching of point clouds
Local and global Fourier transforms
Advanced geometric filtering
Error and resampling

Efficient Simplification of Point Sampled-Geometry. Pauly (30 minutes)
Local surface analysis
Simplification methods
Hierarchical clustering, iterative simplification, particle simulation
Comparison and analysis

Pointshop3D: An Interactive System for Point-Based Surface Editing. Gross (30 minutes)
System overview
Point cloud parameterization
Surface reconstruction
Editing operators (painting, texturing, filtering, sculpting)
Interactive demo

Shape Modeling of Point-Sampled Geometry. Pauly (30 minutes)
Boolean operations
Free-form deformation

Pointshop3D Demonstration. Pauly (30 minutes)
Open system architecture
Demonstration of interactive surface editing and modeling

Panel on the Future of Point Based Computer Graphics. all (30 minutes)

1.4 COURSE WEBSITE

We have compiled a web site for the course including slides and presentation material, which can be found on http://graphics.ethz.ch/publications/tutorials/points/.

The Pointshop3D software package, including source code and documentation, is publicly available from the Pointshop3D web site http://graphics.ethz.ch/pointshop3d/. Using the software infrastructure provided by Pointshop3D course attendants will be able to quickly prototype their research ideas based on point primitives.

1.5 SPEAKERS CONTACT INFORMATION

Dr. Marc Alexa
Professor
Department of Computer Science
Darmstadt University of Technology

Fraunhoferstr. 5
64283 Darmstadt
Germany
alexa@igd.fhg.de
http://www.igd.fhg.de/~alexa

Dr. Markus Gross
Professor
Department of Computer Science
Swiss Federal Institute of Technology (ETH)
CH 8092 Zürich
Switzerland
grossm@inf.ethz.ch
http://graphics.ethz.ch

Dr. Mark Pauly
Postdoctoral Associate
Computer Science Department
Gates Building 375
Stanford University
Stanford, CA 94305
USA
mapauly@stanford.edu
http://graphics.stanford.edu/~mapauly/

Dr. Hanspeter Pfister
Associate Director
MERL - Mitsubishi Electric Research Laboratories
201 Broadway
Cambridge, MA 02139
USA
pfister@merl.com
http://www.merl.com/people/pfister/

Dr. Marc Stamminger
Professor
Graphische Datenverarbeitung Erlangen
University of Erlangen-Nurnberg
Am Weichselgarten 9
91058 Erlangen
Germany
Marc.Stamminger@informatik.uni-erlangen.de
http://www9.informatik.uni-erlangen.de/Persons/Stamminger

Dr. Matthias Zwicker
Postdoctoral Associate
The Stata Center, 32 Vassar Street
Massachusetts Institute of Technology

Cambridge, MA 02139
USA
matthias@graphics.csail.mit.edu
http://graphics.csail.mit.edu/~matthias

1.6 SPEAKERS BIOGRAPHIES

Marc Alexa. is an assistant professor at Darmstadt University of Technology and head of the Discrete Geometric Modeling group. He received his PhD and MS degrees in Computer Science with honors from Darmstadt University of Technology. His research interests include shape representations, modeling, transformation and animation as well as conversational user interfaces and information visualization.

Markus Gross . Markus Gross is a professor of computer science and the director of the computer graphics laboratory of the Swiss Federal Institute of Technology (ETH) in Zürich since 1994. He received a degree in electrical and computer engineering and a Ph.D. on computer graphics and image analysis, both from the University of Saarbrucken, Germany. From 1990 to 1994 Dr. Gross worked for the Computer Graphics Center in Darmstadt, where he established and directed the Visual Computing Group. His research interests include point based graphics, physics-based modeling, multiresolution analysis and virtual reality. He has widely published and lectured on computer graphics and scientific visualization and he authored the book "Visual Computing", Springer, 1994. Dr. Gross has taught courses at major graphics conferences including SIGGRAPH, IEEE Visualization, and Eurographics. He is associate editor of the IEEE Computer Graphics and Applications and has served as a member of international program committees of many graphics conferences. Dr. Gross was a papers co-chair of the IEEE Visualization '99, the Eurographics 2000, and the IEEE Visualization 2002 conferences. Dr. Gross is a member of the ETH research and planning commissions.

Mark Pauly. Mark Pauly obtained his PhD from the Computer Graphics Lab at ETH Zurich, Switzerland. Before, he had received his MS degree in computer science (with honors) from the University of Kaiserslautern, Germany. He has been working on point-based surface representations for 3D digital geometry processing, focusing on spectral methods for surface filtering and resampling. Further research activities are directed towards multiresolution modeling, geometry compression and texture synthesis of point-sampled objects. Currently, Dr. Pauly is a postdoctoral scholar with Stanford University.

Hanspeter Pfister . Hanspeter Pfister is Associate Director and Senior Research Scientist at MERL - Mitsubishi Electric Research Laboratories - in Cambridge, MA. He is the chief architect of VolumePro, Mitsubishi Electric's real-time volume rendering hardware for PCs. His research interests include computer graphics, scientific visualization, and computer architecture. His work spans a range of topics, including point-based graphics, 3D photography, volume graphics, and computer graphics hardware. Hanspeter Pfister received his Ph.D. in Computer Science in 1996 from the State University of New York at Stony Brook. He received his M.S. in Electrical Engineering from the Swiss Federal Institute of Technology (ETH) Zurich, Switzerland, in 1991. Dr. Pfister has taught courses at

major graphics conferences including SIGGRAPH, IEEE Visualization, and Eurographics. He is Associate Editor of the IEEE Transactions on Visualization and Computer Graphics (TVCG), member of the Executive Committee of the IEEE Technical Committee on Graphics and Visualization (TCVG), and has served as a member of international program committees of major graphics conferences. Dr. Pfister was the general chair of the IEEE Visualization 2002 conference in Boston. He is member of the ACM, ACM SIGGRAPH, IEEE, the IEEE Computer Society, and the Eurographics Association.

Marc Stamminger. Marc Stamminger received his PhD in computer graphics in 1999 from the University of Erlangen, Germany, for his work about finite element methods for global illumination computations. After that he worked as a PostDoc at the Max-Planck-Institut for Computer Science (MPII) in Saarbrücken, Germany and in Sophia -Antipolis in France on interactive rendering and modeling of natural environments. Since 2002 he is a professor for computer graphics and visualization at the University of Erlangen. His current research interests are pointbased methods for complex, dynamic scenes, and interactive global illumination methods.

Matthias Zwicker. Matthias Zwicker obtained his PhD from the Computer Graphics Lab at ETH Zurich, Switzerland. He has developed rendering algorithms and data structures for point-based surface representations, which he presented in the papers sessions of SIGGRAPH 2000 and 2001. He has also extended this work towards high quality volume rendering. Other research interests concern compression of point-based data structures, acquisition of real world objects, and texturing and painting of point sampled surfaces. Currently, Dr. Zwicker is a postdoctoral associate with the Computer Graphics Group at MIT.

SECTION 2

COURSE SCHEDULE

TABLE 2.1 Morning Schedule

Time	Topic	Lectureer
8.30-8.45	Welcome and Introduction	Gross
8.45-9.45	Acquisition	Pfister
9.45-10.15	Representations – I	Alexa
10.15-10.30	Coffee Break	
10.30-11.00	Representations – II	Alexa
11.00-12.15	High Quality Rendering	Zwicker
12.15-1.45	Lunch Break	

TABLE 2.2 Afternoon Schedule

Time	Topic	Speaker
1.45-2.30	Data Structures	Stamminger
2.30-3.00	Processing and Filtering	Gross
3.00-3.30	Simplification	Pauly
3.30-3.45	Coffee Break	
3.45-4.15	Pointshop3D	Gross
4.15-4.45	Shape Modeling (Pointshop 2)	Pauly
4.45-5.15	Pointshop3D Demo	Pauly
5.15-5.30	Panel	all

SECTION 3

TALK SLIDES

Course Introduction. Gross

Acquisition of Point Sampled Geometry and Appearance. H. Pfister

Point Based Surface Representations. M. Alexa

Algorithms for High Quality Point Rendering. M. Zwicker

Effiecient Data Structures. M. Stamminger

Processing, Sampling and Filtering of Point Models. M. Gross

Efficient Simplification of Point Sampled-Geometry. M. Pauly

Pointshop3D: An Interactive System for Point-Based Surface Editing. M. Gross

Shape Modeling of Point-Sampled Geometry. M. Pauly

SIGGRAPH2004

Point-Based Computer Graphics

Marc Alexa, Markus Gross, Mark Pauly,
Hanspeter Pfister, Marc Stamminger,
Matthias Zwicker

So…Points

SIGGRAPH2004

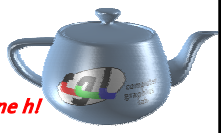
- Point primitives have experienced a major „renaissance“ in Graphics
- Two reasons for that:
  - Dramatic increase in polygonal complexity
  - Upcoming 3D digital photography
- Researchers start to question the utility of polygons as „the one and only“ fundamental graphics primitive
- Points *complement* triangles !

Polynomials....

SIGGRAPH2004

✓ Rigorous mathematical concept
✓ Robust evaluation of geometric entities
✓ Shape control for smooth shapes

× Require proper parameterization
× Discontinuity modeling
× Topological flexibility



Reduce p, refine h!

Polynomials -> Triangles

SIGGRAPH2004

- Piecewise linear approximations
- Irregular sampling of the surface
- No parameterization needed (for now)

Triangle meshes

- Multiresolution modeling
- Compression
- Geometric signal processing

Triangles...

SIGGRAPH2004

✓ Simple and efficient representation
✓ Hardware pipelines support Δ
✓ Advanced geometric processing
✓ The widely accepted queen of graphics primitives

× Sophisticated modeling is difficult
× (Local) parameterizations still needed
× Complex LOD management
× Compression and streaming is highly non-trivial

Remove connectivity !

Triangles -> Points

SIGGRAPH2004

- Piecewise linear functions to Delta distributions
- Discrete samples of geometry
- No connectivity or topology – most simple
- Store all attributes per surface sample

Point clouds

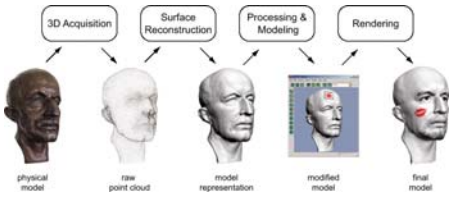
- Points are natural representations for 3D acquisition systems
- Meshes constitute an „enhancement“ of point samples

(Incomplete) History of Points

- Particle systems [Reeves 1983]
- *Points as a display primitive [Whitted, Levoy 1985]*
- Oriented particles [Szeliski, Tonnesen 1992]
- Particles and implicit surfaces [Witkin, Heckbert 1994]
- Rendering Architectures [Grossmann, Dally 1998]
- Digital Michelangelo [Levoy et al. 2000]
- Image based visual hulls [Matusik 2000]
- Surfels [Pfister et al. 2000]
- QSplat [Rusinkiewicz, Levoy 2000]
- Point Clouds [Linsen, Prautzsch 2001]
- Point set surfaces [Alexa et al. 2001]
- Radial basis functions [Carr et al. 2001]
- Surface splatting [Zwicker et al. 2001]
- Randomized z-buffer [Wand et al. 2001]
- Sampling [Stamminger, Drettakis 2001]
- Opacity hulls [Matusik et al. 2002]
- Pointshop3D [Zwicker, Pauly, Knoll, Gross 2002]
- Raytracing [Alexa et al. 2003]
- Sequential Point Trees [Dachsbacher, Stamminger 2003]
- Boolean Operations [Adams et al. 2003]
- Modeling [Pauly et al. 2003]
- blue-c [Gross et al. 2003]

A Motivation…

- 3D content creation pipeline



Points generalize Pixels !

Taxonomy

Rendering (Zwicker)

Acquisition (Pfister)

Point-Based Graphics

Representation (Alexa)

Processing & Editing (Gross, Pauly)

Schedule - Morning

Time	Topic	Lecturer
8.30-8.45	Welcome and Introduction	Gross
8.45-9.45	Acquisition	Pfister
9.45-10.15	Representations – I	Alexa
10.15-10.30	Coffee Break	
10.30-11.00	Representations – II	Alexa
11.00-12.15	High Quality Rendering	Zwicker
12.15-1.45	Lunch Break	

Schedule - Afternoon

Time	Topic	Lecturer
1.45-2.30	Data Structures	Stamminger
2.30-3.00	Processing and Filtering	Gross
3.00-3.30	Simplification	Pauly
3.30-3.45	Coffee Break	
3.45-4.15	Pointshop3D	Gross
4.15-4.45	Shape Modeling (Pointshop 2)	Pauly
4.45-5.15	Pointshop3D Demo	Pauly
5.15-5.30	Panel on the future of PBG	All

The Purpose of our Course is …

1. *…to introduce points as a versatile and powerful graphics primitive*
2. *…to present state of the art concepts for acquisition, representation, processing and rendering of point sampled geometry*
3. *…to stimulate YOU to help us to further develop Point Based Graphics*

SIGGRAPH2004

Acquisition of Point-Sampled Geometry and Appearance

Hanspeter Pfister, MERL [pfister@merl.com]

In this part of the course we will discuss acquisition and rendering of point-sampled models from real-world objects.

In Collaboration With

- Wojciech Matusik, MIT / MERL
- Addy Ngan, MIT
- Matt Loper, MERL
- Paul Beardsley, MERL
- Remo Ziegler, MERL
- Leonard McMillan, UNC

This work would not have been possible without the people mentioned on this slide. In particular, the hard work and great talent of Wojciech Matusik was instrumental for this research.

The goal of our work is – quite simply – to capture and display complex three-dimensional objects, such as an angle with feathers, a bonsai tree, or a teddy bear. A secondary goal is to solve one of the longstanding open problems in computer graphics: capturing and rendering a glass of beer.

Goals

- Fully-automated 3D model creation
- Faithful representation of appearance
- Placement into new virtual environments

More specific goals are to build a system with fully-automated 3D model acquisition. We focus on faithful acquisition of appearance, not geometry, although good geometry will help our rendering results. Our objects can be placed in arbitrary environments with arbitrary lighting and can be rendered from any viewpoint.

The pictures show some of the acquired models (hat, ship, glass) placed into new environments.

Previous Work - I

- Contact digitizers – intensive manual labor
- Passive methods – require texture, Lambertian BRDF
- Active light imaging systems – restrict types of materials
- Fuzzy, transparent, and refractive objects are difficult

4 million pts.
[Levoy et al. 2000]

There has been a lot work on acquiring high quality 3D shape of real-world objects. This includes contact digitizers, passive methods, and active light imaging systems.

Contact digitizers require a lot of manual labor.

Nearly all passive techniques require that the object has texture and that the BRDF of the object is Lambertian.

Active Methods are very popular. For example, on the right you see a laser range-scan of David, one of the great point-based models that resulted from the Digital Michelangelo project at Stanford [Levoy et al. 2000].

However, active light systems place severe restrictions on types of materials that can be scanned. For example, specular and fuzzy materials are difficult to scan. Furthermore, the obtained models require complicated alignment and hole filling procedures.

Previous Work - II

- BRDF estimation, inverse rendering
- Image based modeling
- Point-based rendering

There has been also a considerable amount of work on estimation of appearance of objects from a set of images.

These methods usually try to fit parametric BRDF model to the observed measurements. However parametric BRDF models are not capable to represent complex materials (e.g., feathers, hair, cloth, etc.) They also do not capture inter-reflections, self-shadowing, subsurface scattering.

To acquire objects with arbitrary materials we use an image-based modeling and a point-based rendering approach.

Image-based representations have the advantage of capturing and representing an object regardless of the complexity of its geometry and appearance. Of course, this advantage comes at the cost of requiring more data.

Outline

- Acquisition
- Model
- Point-Based Rendering
- Relighting
- Reflectance Field Estimation

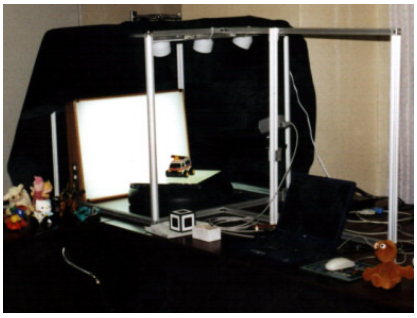
In this talk I will talk about an image-based method to acquire models of complex real-world objects. First I will present the acquisition system, followed by a discussion of the opacity lightfield, which we use to model our objects. Then I will briefly discuss our point-based rendering pipeline with special emphasis on view-dependent shading. After that I will show how opaque objects can be relit using reflectance fields, which allows us to place them into arbitrary new environments. Finally I will discuss a novel method for reflectance field illumination for arbitrary scenes, including reflective, refractive, and transparent objects.

Outline

- Acquisition
- Model
- Point-Based Rendering
- Relighting
- Reflectance Field Estimation

Let’s start by looking at our image-based acquisition system.

Acquisition Systems – I & II



V1.0 (July-Oct 2000)

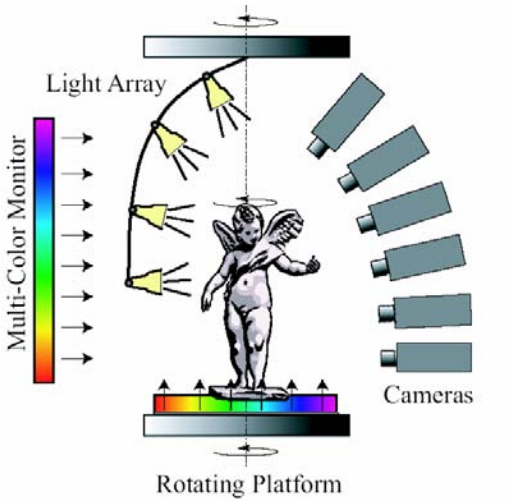
V2.0 (Oct-Dec 2000)

We have been building image-based acquisition systems for some time. The general idea is to take multiple pictures of the object from different viewpoints.

Our first system – shown on the left – consisted of a turntable, a camera, and some static overhead lights. Two light boxes – one opposite the camera and one on the turntable – provided background illumination to make segmentation of the foreground object easier.

The second system – shown on the right – added two more cameras to improve the quality of the image-based model. We also mounted the overhead lights on a second turntable so that they can rotate with the object. This ensures that specular highlights remain stationary on the object throughout acquisition.

Acquisition System - III



This is our third system, which one may call the mother ship of all image-based acquisition systems [Matusik et al. 2002].

Objects are placed onto a plasma display that is mounted on the rotating platform. An array of lights is mounted on an overhead rotating platform. Six video cameras are on the stationary arc and are pointed towards the object.

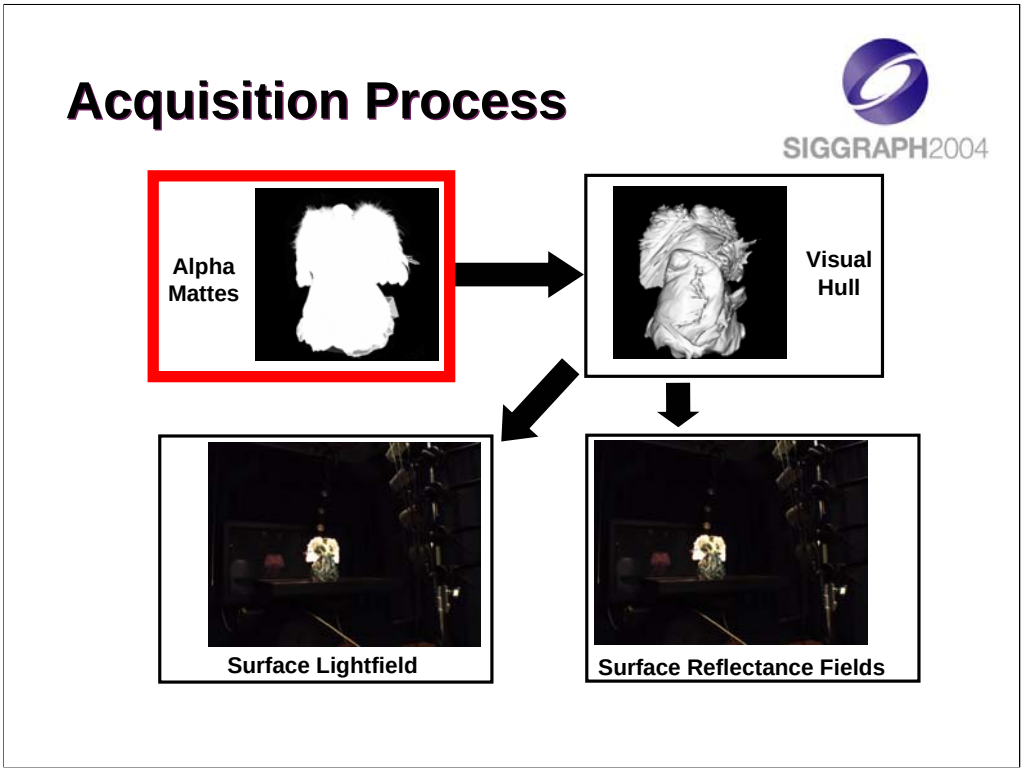
We use the plasma displays for multi-background matting techniques [Smith & Blinn 96] to acquire alpha mattes of the object from multiple viewpoints. We also use them to estimate the light transport through transparent and refractive objects, but more on that later.

A lot of headwork…

SIGGRAPH2004

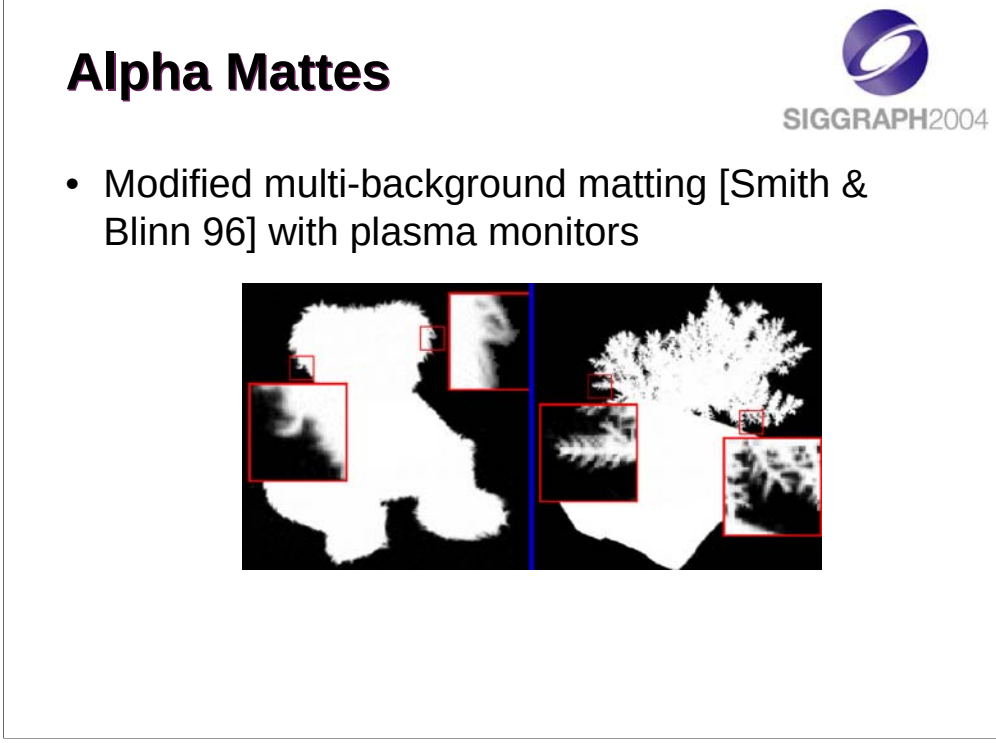
- Building systems is hard, but worthwhile

Building these systems is a lot of headwork… But I highly encourage you to try it, because the experience of building systems will focus your thinking. And playing with turntable and cameras is fun… mostly…



The acquisition starts by first acquiring high-quality alpha mattes of the object from different viewpoints

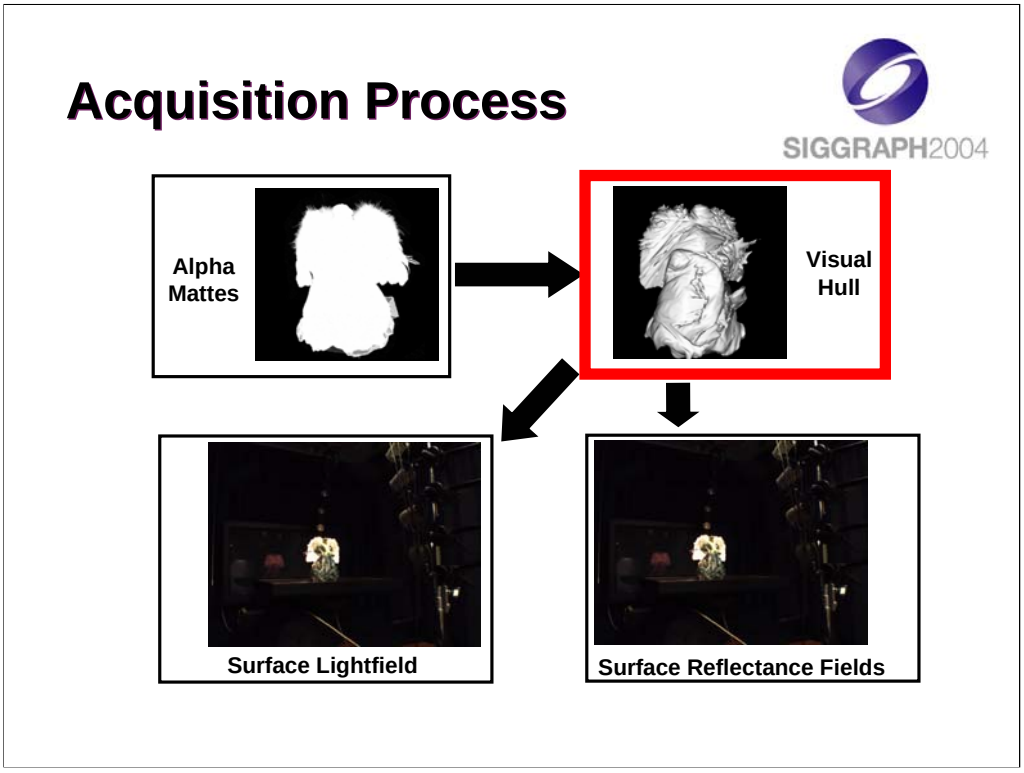
Alpha Mattes



- Modified multi-background matting [Smith & Blinn 96] with plasma monitors

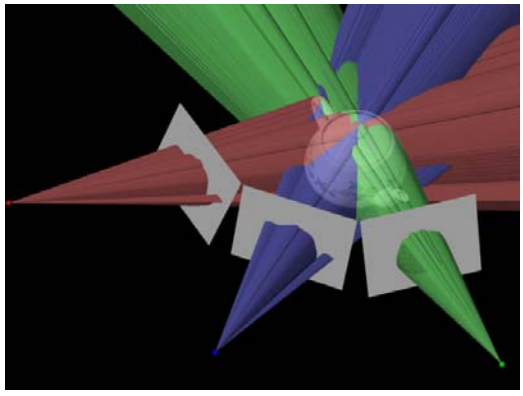
As mentioned before, we use the plasma monitors to display know, random patterns in the background. By comparing the acquired images with images of the known background we can compute the partial coverage of the foreground object with the background matte [Smith & Blinn 96]. The result is a sub-pixel accurate alpha matter, as shown in these examples.

After alpha matte acquisition we switch off the plasma displays.



Next we use a threshold to convert the alpha mattes into binary silhouette images. These images are then used to compute an approximate geometry of the object know as the *visual hull*.

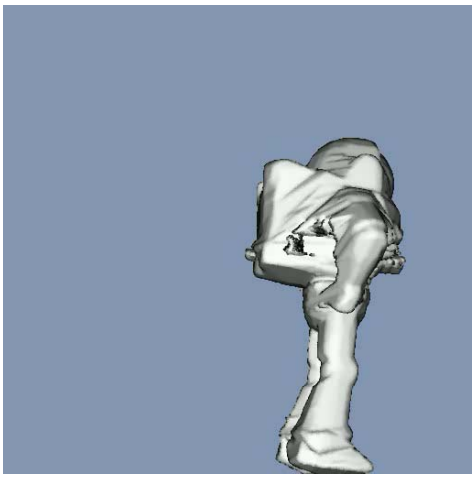
Visual Hull



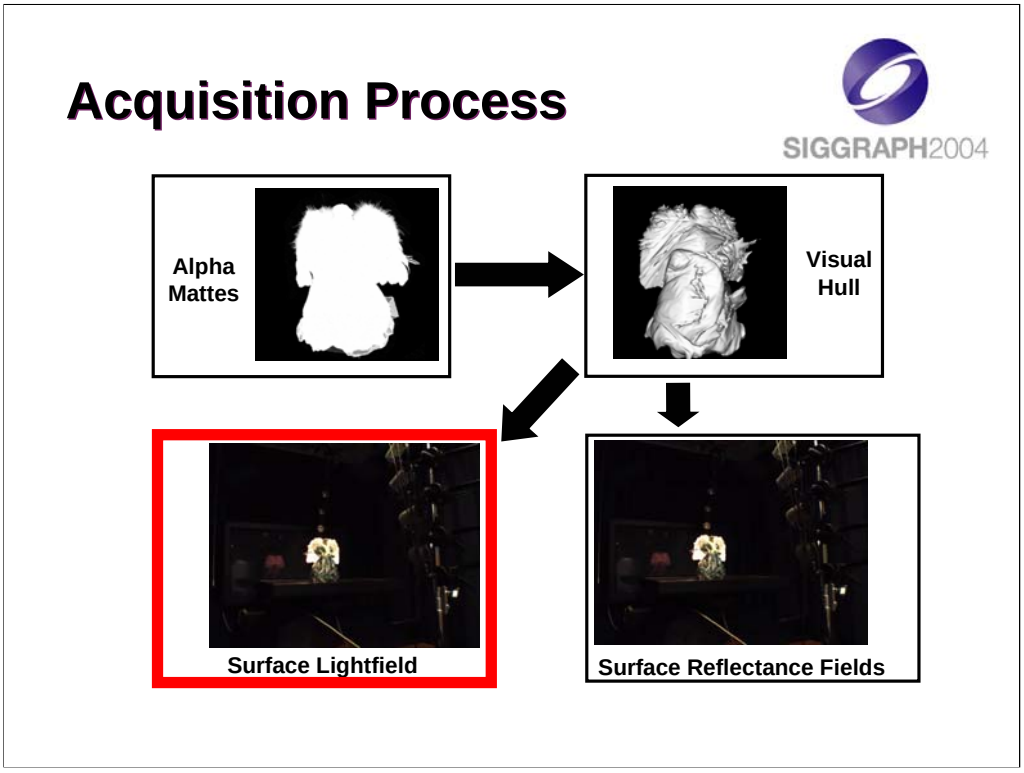
- The maximal object consistent with a given set of silhouettes

A Visual hull is an approximating geometric shape which is obtained using the silhouettes of an object as seen from a number of views. Each silhouette is extruded creating one cone-like volume that bounds the extent of the object. The intersection of these bounding volumes gives us the visual hull. Note that this method is unable to capture all concavities.

Visual Hull Example

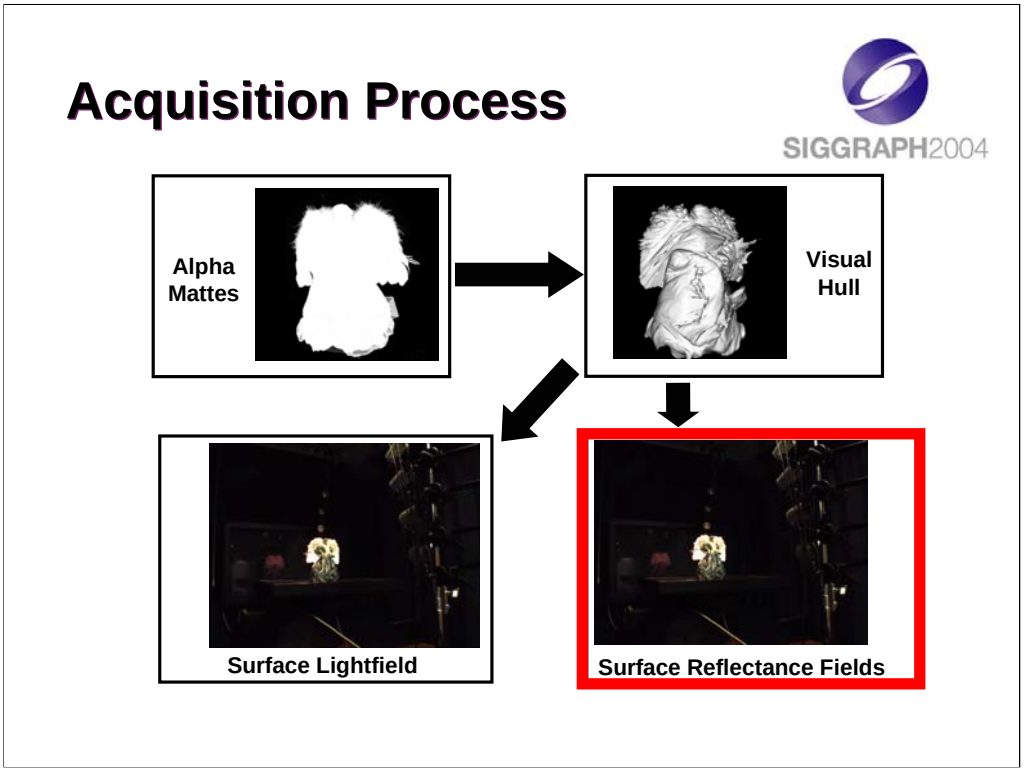


Here is an animation of the visual hull of a Buzz Lightyear model that was scanned in one of our early systems. Note that the quality of the visual hull geometry is a function of the number of viewpoints / silhouettes.



Now we have two options:

We can acquire the surface lightfield of the object – the object under a fixed illumination. That means we rotate the object while the lighting is fixed with respect to the object.



Or we can acquire the surface reflectance field of the object, which is necessary for relightable models.

The array of lights is rotated around the object. For each rotation position, each light in the light array is sequentially turned on and an image is captured with each camera.

Outline

- Acquisition
- ➢ Model
- Point-Based Rendering
- Relighting
- Reflectance Field Estimation

From the acquired data we build a model of our objects based on the opacity lightfield, a surface lightfield that includes view-dependent opacity.

View-Dependent Textures

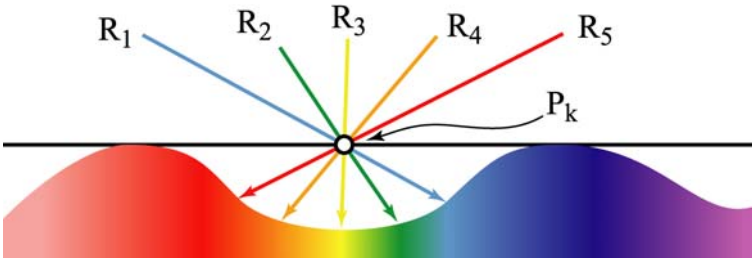
- Capture concavities, reflections, and transparency with view-dependent textures [Pulli 97, Debevec 98]

To add appearance to our model we can use view-dependent (or projective) texture mapping [Debevec et al. 98][Pulli et al. 97]. Here we see our Buzz Lightyear model from different viewpoints.

Note that the view-dependent textures capture reflections and transparency and give the illusion of improved concavities. For example, the visual hull does not represent Buzz's head, although it clearly appears in the texture-mapped model.

Surface Lightfield

- Parameterize view-dependent textures onto the object surface [Wood *et al*., 2000]



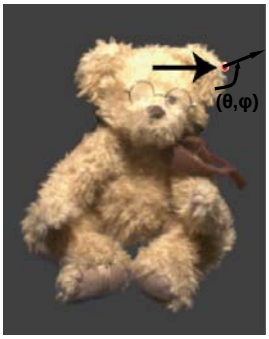
To improve the efficiency for compression and to improve the image quality during view-interpolation we parameterize the view-dependent radiance samples on the visual hull surface of the object. This is called a *surface lightfield* [Wood et al. 2000].

Note that the visual hull geometry (the black line in the figure) does not correspond to the true geometry of the surface. Nevertheless, the view-dependent surface lightfield is able to capture the actual surface *appearance*.

Opacity Lightfield

SIGGRAPH2004

- Also parameterize view-dependent opacity onto the object surface
- 4D function f(u,v,ϕ,θ) → RGBA



Opacity

Red = invisible
Black = transparent

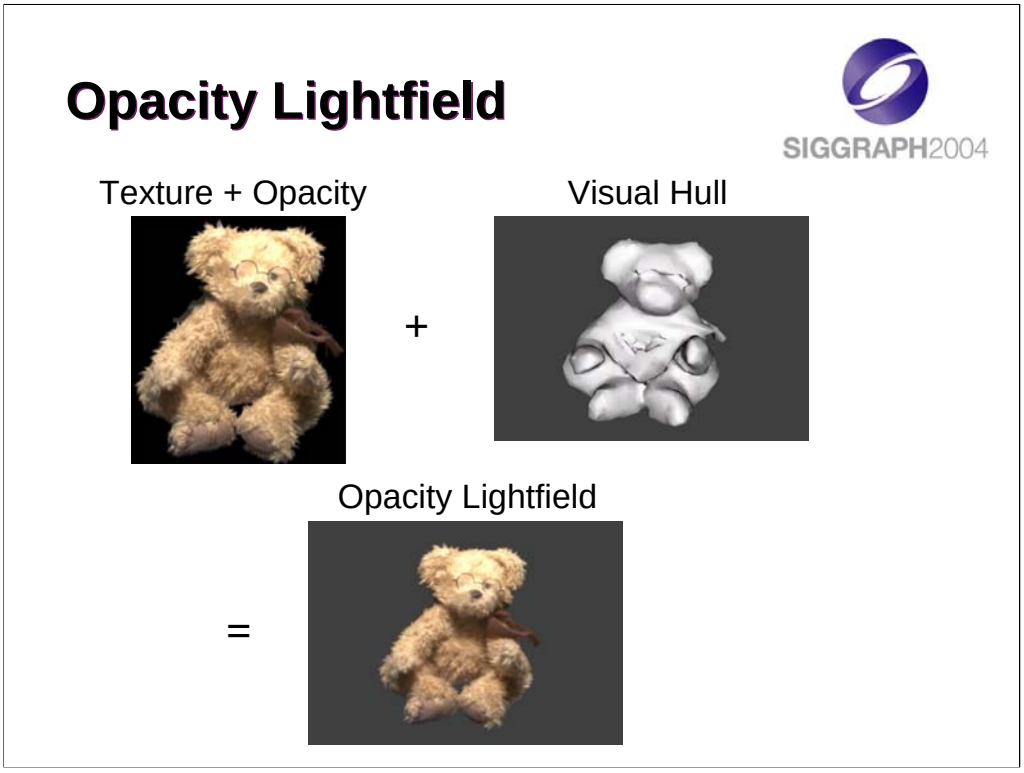
Texture + Opacity

To further improve the appearance, especially for fuzzy or furry objects, we enhance the surface lightfield with view-dependent opacity. We call this new representation the *opacity lightfield* [Matusik et al. 2002, Vlasic et al. 2003].

The opacity lightfield is a four dimensional function. For each point (u, v) on the object surface it stores radiance and opacity as a 2D function of the viewing direction (phi, theta).

In these plots we show the opacity function and the opacity lightfield for one particular surface point. White represents the opaque values, black represents the transparent values, and red represents invisible directions. In general, it is quite difficult to fit a parametric function to the opacity lightfield, although that would certainly be possible for certain materials.

It is worth to note that the parameterization surface does not have to be a surface of the visual hull. It can be any surface that contains the actual object.



In summary: Our appearance representation for static light sources is the opacity lightfield, which simply consists of texture + opacity for each acquisition viewpoint. Those textures with alpha are mapped onto the visual hull geometry in a view-dependent fashion. We will discuss the details of rendering of our models later.

Example

Photo

Let’s look at an example. Here we is a photograph of a small bonsai tree.

Example

Photo

Visual Hull

This animation shows the visual hull of the tree. Note that the visual hull does not capture any of the fine features, such as leaves and branches.

Example

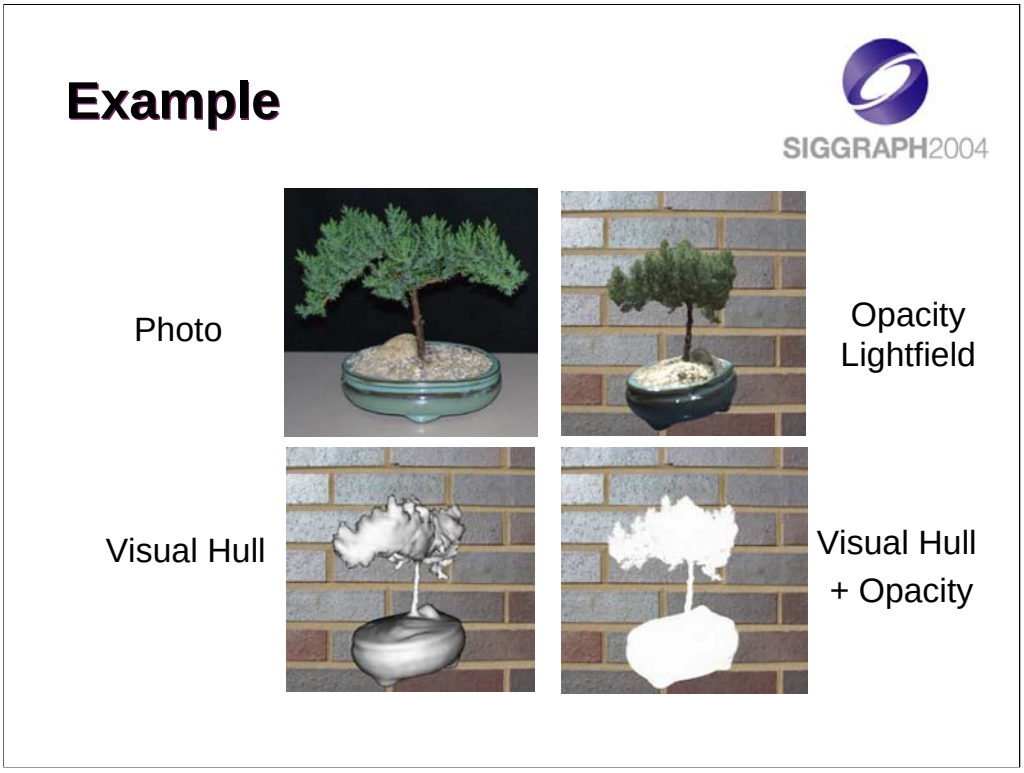
SIGGRAPH2004

Photo

Visual Hull

Visual Hull
+ Opacity

Here is the visual hull plus view-dependent opacity. It greatly enhances the quality of the object's silhouette. The background is also visible through the leaves and branches, even though the geometry is exactly the same as on the left.



Finally we map the view-dependent textures onto the surface, too, for the final opacity lightfield model. Note the specular highlights on the base of the tree.

Results Video

Here is a video of the angel. Look at the feathers, the anisotropic cloth, and the semi-transparent cloth on top of that. At some places you will see that the approximate visual hull geometry leads to artifacts.

Results Video

Here is another video of the bonsai tree, this time with a close-up view. Look at the leaves in the tree and the specular highlights on the vase.

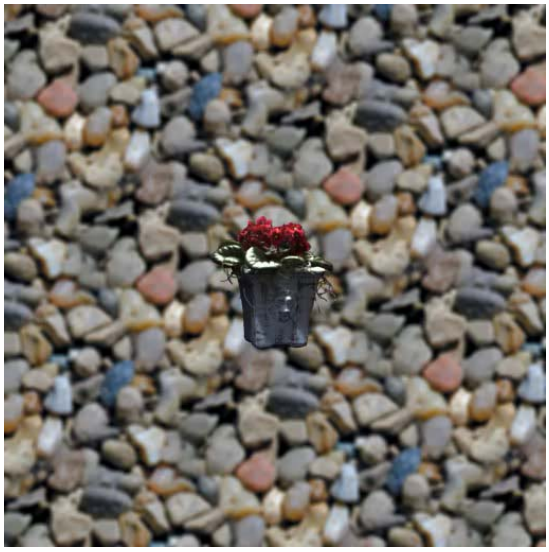
Results Video

Here is a very specular teapot with intricate texture. You will see some jumping of the highlights in the close-up view. This is due to the limited number of pictures – even though we took 72 x 6 = 432 of them. Very specular objects require a tremendous amount of image data.

Results Video

Here is a video of a teddy bear. The organic complexity of the silhouette is captured nicely by the opacity lightfield, as is the transparency of the glasses. You will see artifacts in-between the arms of the teddy because of the bad visual hull geometry.

Results Video



Our acquisition system can also capture organic models, such as living plants, which would be very hard to do with active light or passive systems.

Lessons

- View-dependent opacity vs. detailed geometry trade-off
- Sometimes acquiring the geometry is not possible
- Sometimes representing true geometry would be very inefficient
- Opacity lightfield stores the “macro” effect

There is a logical trade-off between having accurate geometry and the opacity lightfield.

If we have good geometry the sampling of the opacity and textures does not need to be very dense. However, if we do not have good geometry we need the view-dependent opacity and textures at a very good sampling rate.

There are at least two good reasons why to use the opacity lightfield:

Sometimes, geometry is impossible to get. The resolution of the acquisition device – such as range scanner or a camera – may be limited, which means we cannot pick up the details below a certain resolution.

And sometimes representing the true geometry would be very inefficient (e.g. for hair). We would simply need too many points or polygons to represent it.

The opacity lightfield stores the macro effect, the overall effect of the geometry.

Outline

- Acquisition
- Model
- ➢ Point-Based Rendering
- Relighting
- Reflectance Field Estimation

Let's now talk about the rendering of our point-based models. In particular we will discuss the view-dependent shading using the opacity lightfield (RGBA) data.

Why Points?

- Generating a consistent triangle mesh or texture parameterization is time consuming and difficult
- Points represent organic models (feathers, tree) much more readily than polygon models

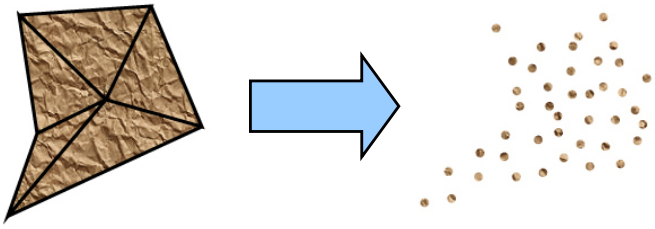
What is the connection between our system and point-based graphics? Quite simply, we use a point-based model of the visual hull to store the opacity lightfield at each surface point.

The main advantage is that we do not need to generate a consistent triangle mesh or texture parameterization for our models. We avoid difficult issues such as mesh-stitching, finding a global parameterization of the mesh, texture resampling, texture anti-aliasing, etc.

In addition, points are a more natural choice to represent organic models such as feathers and trees, which may have very small features and very complex silhouette lines.

Points as Rendering Primitives

- Point clouds instead of triangle meshes [Levoy and Whitted 1985]
- 2D vector versus pixel graphics



triangle mesh (with textures)

point cloud

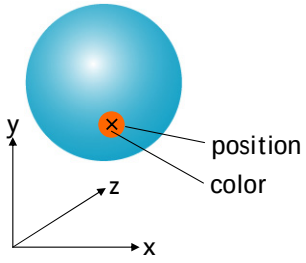
The idea of using points instead of triangle meshes and textures has first been proposed by Levoy and Whitted in a pioneering report [Levoy & Whitted 85].

Think of the difference between points and triangles in 3D similar as of the difference between pixels and vector graphics in 2D. Points in 3D are analogous to pixels in 2D, replacing textured triangles or higher order surfaces by zero-dimensional elements.

Surface Elements - Surfels

- Each point corresponds to a surface element, or ***surfel,*** describing the surface in a small neighborhood
- Basic surfels:

```
BasicSurfel {
  position;
  color;
}
```



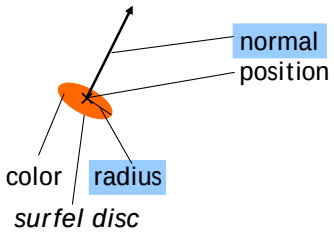
We call a point sample on the object surface a *surface element*, or *surfel*. It is similar to a pixel in 2D or a voxel in 3D. As a matter of fact, there is a tight connection between volume rendering and point-based surface rendering (see [Zwicker et al. 2002]).

A surfel describes the surface in a (possibly infinitely) small neighborhood. The basic surfel is just a point with a position and a constant color.

Surfels

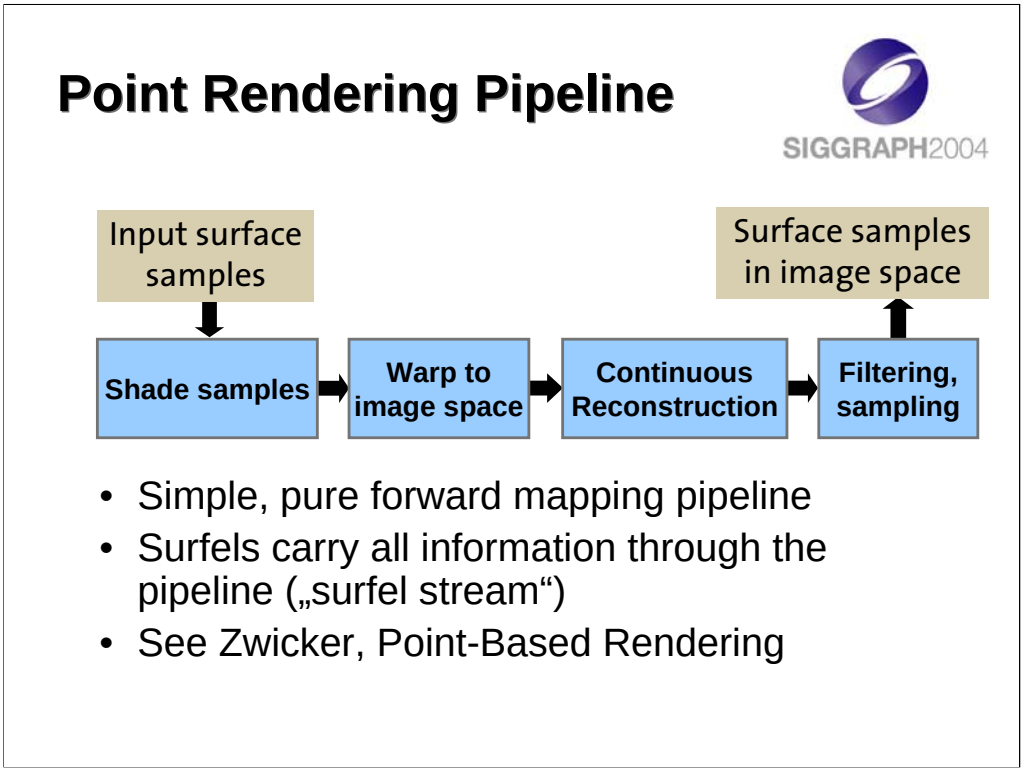
- Surfels can be extended by storing additional attributes

```
ExtendedSurfel {
  position;
  color;
  normal;
  radius;
  etc...
}
```



Surfels can of course be extended by storing additional attributes. We also store a normal and a radius with each surfel. This corresponds to having a surfel disc with radius r and normal n.

Intuitively, the union of surfel discs covers the object such that there are no holes on the object surface. This will come in handy for reconstructing a continuous image during rendering, as you will see in the talk by Matthias Zwicker.



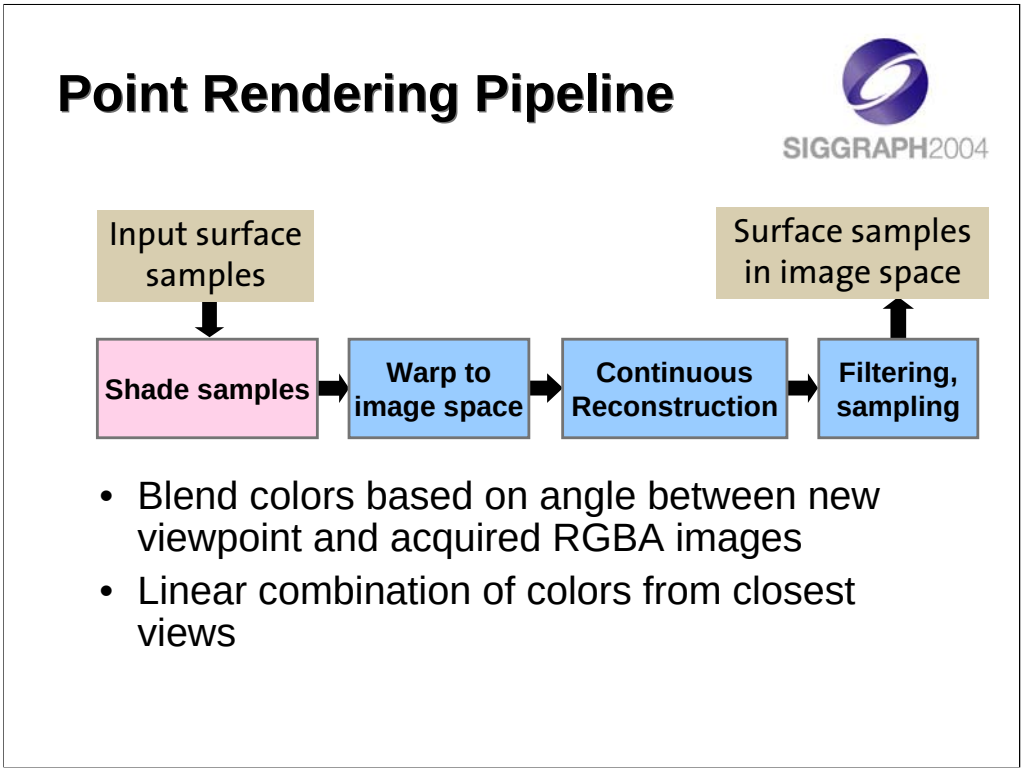
The point rendering pipeline processes point data as follows: Input surface samples are first shaded according to the opacity lightfield data. We will discuss this step in more detail next.

In the next stage, the points are projected, or warped, to image space. This is analogous to projecting triangle vertices to image space.

Next, a continuous surface is reconstructed in image space, which is similar to triangle rasterization.

The final stage in the pipeline is filtering and sampling the continuous representation at the output pixel positions, which is is similar to texture filtering and sampling.

Matthias Zwicker will talk in much more detail about the reconstruction, filtering, and resampling steps [Zwicker et al. 2001]. Point-based rendering can also be accelerated with commodity graphics hardware, e.g., [Ren et al. 2002].

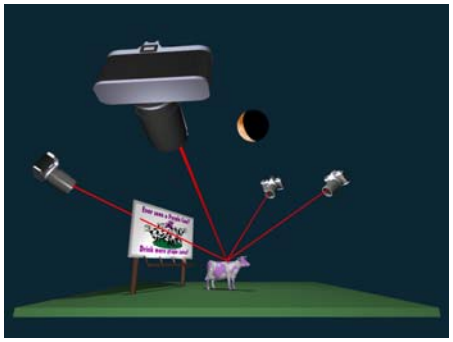


We will now focus on the pre-surfel shading.

The basic shading algorithm blends colors and alphas from the closest images that were acquired from different camera views. The final color per surfel is a linear combination of the colors from the closest views.

Color Blending

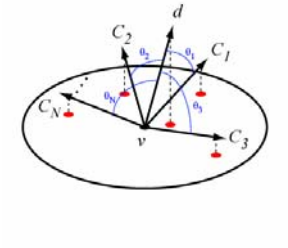
- Unstructured Lumigraph blending [Buehler 2001]
- Weights are based on angles between camera vectors and the new viewpoint



The formula for this linear combination was developed for Unstructured Lumigraph Rendering by [Buehler et al. 2001]. The blending weights are computed based on the angles between the camera vectors and the new viewpoint.

ULR Blending

- For each surfel and some view direction d:
  - Find the k closest camera views
  - Compute the blending weights r_i for each
  - Normalize to get the final weights w_i



$$r_i = \frac{\cos\theta_i - \cos\theta_k}{1 - \cos\theta_i}$$

$$w_i = \frac{r_i}{\sum_{i=1}^{k-1} r_i},\ 1 \le i \le k$$

ULR uses a ranking function based on the angles between original camera views and the desired view.

For each surfel *v* and some viewing direction *d*, the algorithm finds the *k* nearest visible views by comparing the angles they subtend with *d.*

Blending weights are computed according to the following formula:

$$r_i = \frac{\cos\theta_i - \cos\theta_k}{1 - \cos\theta_i}$$

Here, r_i represent relative weights of the closest *k*-1 views. Those are normalized to sum up to one, resulting in actual weights w_i:

$$w_i = \frac{r_i}{\sum_{i=1}^{k-1} r_i},\ 1 \le i \le k$$

For more details about this blending formula, including implementation and optimization suggestions, see [Vlasic et al. 2003].

Shading Algorithm

- Pre-compute and store visibility vector per surfel
- During rendering, project surfel into k closest visible images to get RGBA colors
- Blend the RGBA colors using ULR weights

```
ExtendedSurfel {
  position;
  visibility[NUM_VIEWS];
  normal;
  radius;
  etc...
}
```

We can further optimize this calculation by pre-computing the visibility of each camera for each surfel by simply projecting it into each camera view. We store the visibility as a binary vector per surfel. For each acquired camera view, the vector stores a 0 (not visible) or 1 (visible).

During rendering, we determine the k closest visible camera views for each surfel based on the angles they subtend with the new virtual view. We project the surfel into the k closest images to get RGBA colors for ULR blending.

Then we compute the blending weights based on the formula presented on the previous slide and compute the final surfel color.

Demo

- Real-time for SW-only implementation
- No alpha blending
- Graphics HW implementation [Vlasic 2003]

This demo runs on my laptop in real-time using a software-only implementation. However, we are not doing any alpha blending and the resolution of the textures is limited. Nevertheless, the achievable image quality is very good.

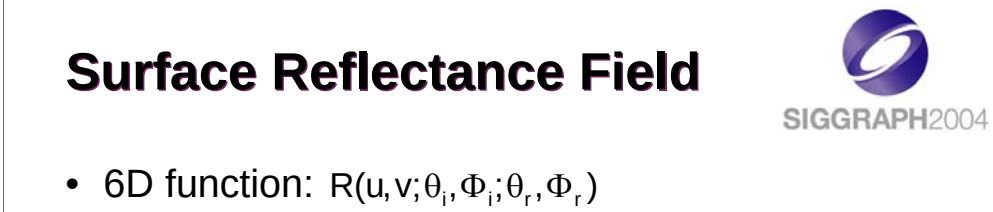
We also implemented opacity lightfield rendering on modern graphics hardware. See [Vlasic et al. 2003] for details.

Outline

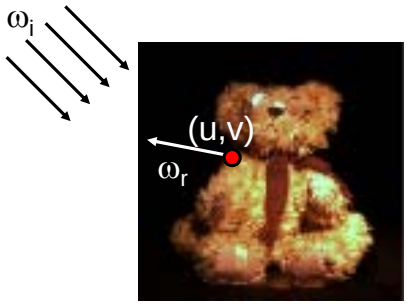
- Acquisition
- Model
- Point-Based Rendering
- ➢ Relighting
- Reflectance Field Estimation

So far, our models just display the lighting that was used during acquisition. In order to place the models into arbitrary new environments we need to be able to relight them.

Surface Reflectance Field



- 6D function: $R(u,v;\theta_i,\Phi_i;\theta_r,\Phi_r)$
- Assumes directional illumination [Debevec 2000]



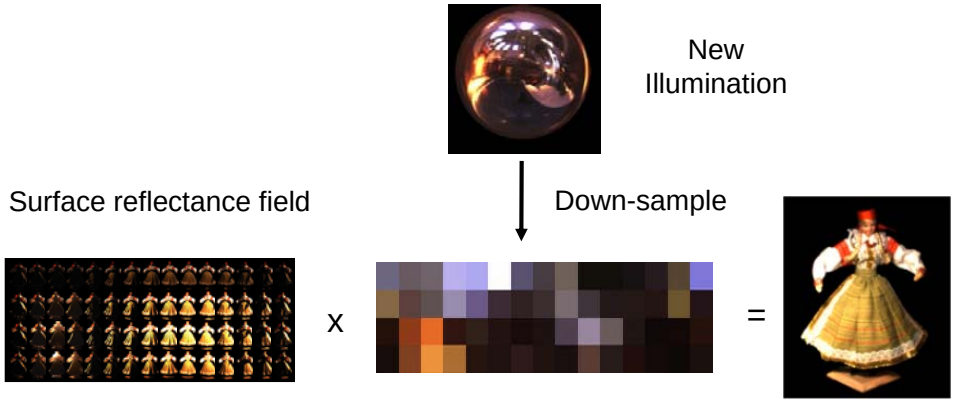
To relight our models we acquire the *surface reflectance field*. The SRF is a 6D function. For each surfel (u,v) and for each incoming light direction at infinity (ω_i) and for each outgoing direction (ω_o) it stores the ratio of the incoming radiance to the outgoing radiance [Debevec et al. 2000]. It is a simplified form of the BSSRDF for the case of a fixed viewpoint.

For opaque objects we capture images of the object and vary the incoming illumination for each turntable position by rotating the overhead light array. We use 4 lights and 15 positions for a total of 60 light configurations per camera view.

Relighting – 1st Step

SIGGRAPH2004

- Generate new images for camera views



Our modified rendering algorithm has two steps.

In the first step, we generate the new radiance images for all original camera views given the new illumination environment.

Since we acquire the surface reflectance field only for 60 light position we have to severely downsample the new illumination map.

Relighting – 2nd step

- Interpolate images with ULR blending as before

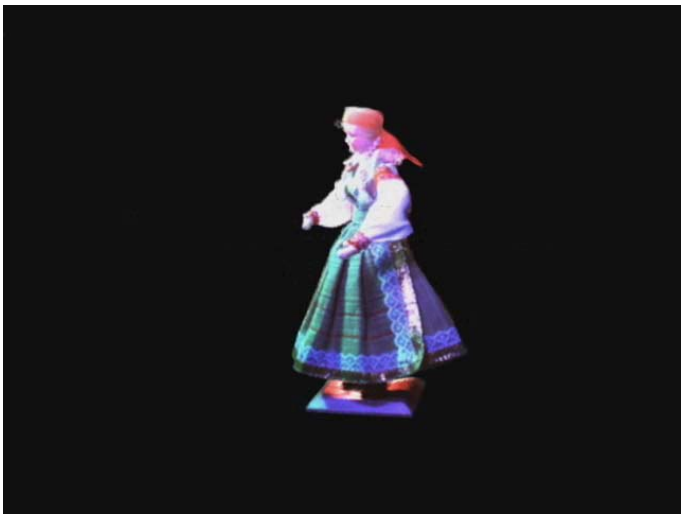
Camera View 1

New View

Camera View 2

In the second step we just use standard ULR blending as presented before to compute the color values of each surfel for the new viewpoint. Note that we interpolate both the radiance (color) and the opacity.

Results Video



In this short video we show some results for opaque 3D models.

First, we use directional light sources with random colors to illustrate the relighting. Then we place the 3D model into a synthetic environment where an environment map provides the incoming illumination.

We also acquired a short video of a real scene in a local library. The camera was tracked, and we acquired the illumination at the scene using a mirror ball and a camera. Using the camera tracking information and an environment map of the illumination we can then render new views of our objects and composite them into the video.

Outline

- Acquisition
- Model
- Point-Based Rendering
- Relighting
- ➢ Reflectance Field Estimation

We now look at a new method to acquire a surface reflectance field of arbitrary objects, including transparent and refractive ones. Our method uses natural illumination for the estimation of the reflectance field. It works for small objects or large, outdoor scenes.

Previous Work

- Forward Approaches
  - Reflectance Fields [Debevec 2000]
- Inverse Approaches
  - Environment Matting [Zongker 99]
- Pre-computed Light Transport

- We use an inverse approach [Matusik 2004]

We can classify the methods for estimating reflectance functions into *forward* and *inverse* methods.

Most forward methods sample the reflectance functions exhaustively and tabulate the results. For each incident illumination direction they store the reflectance function weights for a fixed observation direction. In practice, only low-resolution incident illumination can be used, since one reflectance table has to be stored per surfel. This is what we used so far.

Inverse methods observe an output and compute the probability that it came from a particular region in the incident illumination domain. The incident illumination is typically represented by a bounded region, such as an environment map, which is then modeled as a sum of basis functions. The inverse problem can then be stated as follows: Given an observation (e.g., an image pixel), what are the weights and parameters of the basis functions that best explain the observation? This is what we will discuss now.

Note that recent real-time rendering methods use pre-computed light transport (or pre-computed radiance transfer) to increase the realism of their objects. The estimation method for the light transport is similar to what we will discuss now.

For a detailed list of references see [Matusik et al. 2004].

Light Transport Model

- Incident illumination is 2D parameterizable
- Linear “black box” system:

$$B = TL$$

B = relit image (stacked in vector)

L = incident light (stacked in vector)

T = light transport matrix (***reflectance field***)

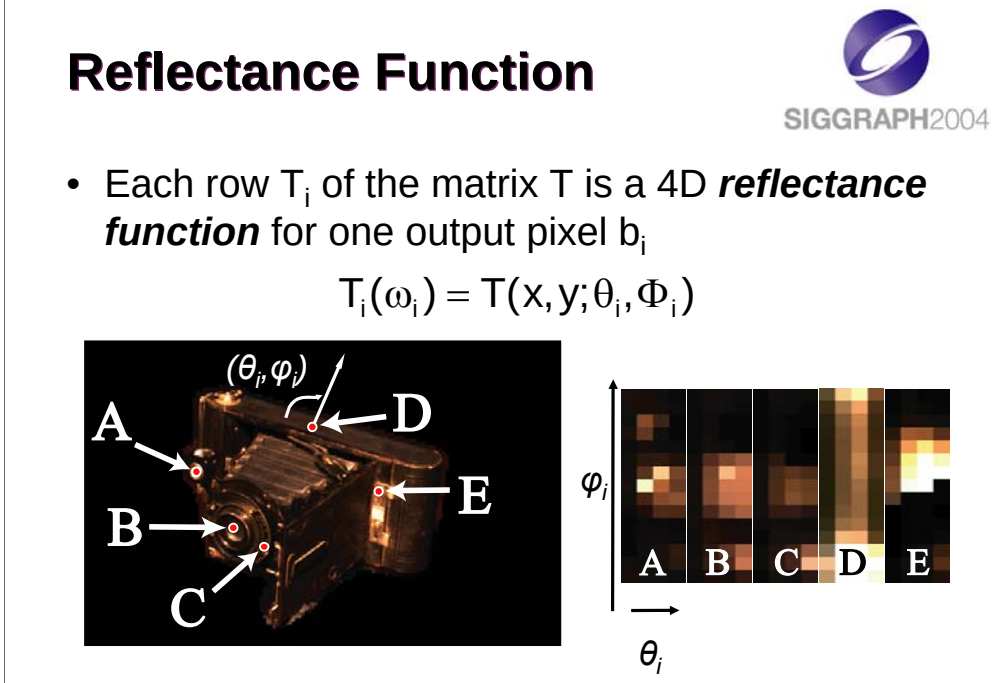
First we start with the light transport model that we adopt:

We assume that the incident illumination is 2D parameterizable (i.e., can be represented by an environment map). If we consider the problem as a black box linear system, we can write:

$$B = TL$$

where *L* is the incident illumination (stacked in a vector), *B* is the resulting relit image (stacked in a vector), and *T* is the light transport matrix (or reflectance field). Each row in *T* represents a reflection function T_i, thus $T = [T_0, \ldots, T_n]$.

Reflectance Function



- Each row T_i of the matrix T is a 4D ***reflectance function*** for one output pixel b_i

$$T_i(\omega_i) = T(x, y; \theta_i, \Phi_i)$$

Here is an example of a row of the reflectance field matrix T. Each row stores a 4D *reflectance function* T_i.

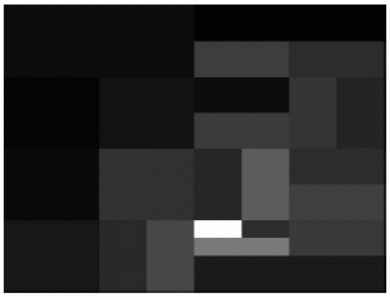
For each pixel (x,y) and each incoming light direction (ω_i) it stores the ratio of the incoming radiance to the outgoing radiance. Note that the outgoing direction is implicitly fixed to be the camera viewpoint.

Here we show five samples of this functions for different image points. We draw these samples as a function of the incoming radiance directions phi_i and $theta_i$.

Representation

- Approximate T_i as a sum of 2D rectangular kernels $R_{k,i}$, each with weight $w_{k,i}$.

$$T_i \approx \sum_k w_{k,i} R_{k,i}$$



We approximate T_i as a weighted sum of several 2D *rectangular kernels*, such that:

$$T_i \approx \sum_k w_{k,i} R_{k,i}$$

where $w_{k,i}$ are the weights of each 2D rectangular box $R_{k,i}$. The figure shows an example of the axis-aligned 2D kernels. The grayscale color corresponds to the weight magnitude, where white is largest and black is zero.

Inverse Estimation

- Display ***input*** images L_j and record ***observed*** images B_j:

$$B_j = TL_j$$

- Given L_j and $b_j(x,y)$, the goal is to estimate T
  - Weights $w_{k,i}$
  - Position and size of the 2D box filters $R_{k,i}$
- We typically use 200 input images

The problem of reflectance field acquisition is now equivalent to estimating the impulse response function (or point spread function) of a linear system with multiple inputs (the incident illumination *L*) and multiple outputs (the observation *B*).

The estimation process can be described as:

$$B_j = TL_j$$

where L_j is a set of different natural incident illumination (the *input images*) and B_j is the resulting set of observed photographs of the scene (the *observed images*).

We use over 200 random input images of indoor and outdoor scenes – such as kitchens, gardens, cities – that are displayed on the monitor.

We denote a pixel *b(i, j)* as pixel *i* in the *j*-th observed image. Given L_j and *b*(*i, j*), the algorithm estimates the weights $w_{k,i}$, positions, and sizes of the 2D rectangular kernels for each pixel b_i in the reflectance field. As we increase the number *n* of input and observed images, we refine the kernels to improve the estimate of the reflectance field.

Algorithm Input and Output

- Input:
  - Images (environment maps) of the input illumination (L_1,…,L_n)
  - Observed images of the scene under input illumination (O_1,…,O_n)
- Output:
  - For each pixel: Weights $w_{k,i}$, positions and sizes for $R_{k,i}$

Here again, more explicitly, the input and outputs of our algorithm.

Algorithm

- Minimize error for each pixel b_i:

$$\arg\min_w \left\| A_i W_i - B_i \right\|^2 \quad \text{subject to } w \geq 0$$

W_i = vector of weights $w_{k,i}$
B_i = vector of observed pixels
A_i = matrix of dot products $R_{k,i} \bullet L_j$

We observe that the problem can be solved independently for each pixel b_i. We solve the following optimization problem:

$$\arg\min_w \left\| A_i W_i - B_i \right\|^2 \quad \text{subject to } w \geq 0$$

W_i is the stacked vector of weights $w_{k,i}$ to be computed (for a single pixel b_i).

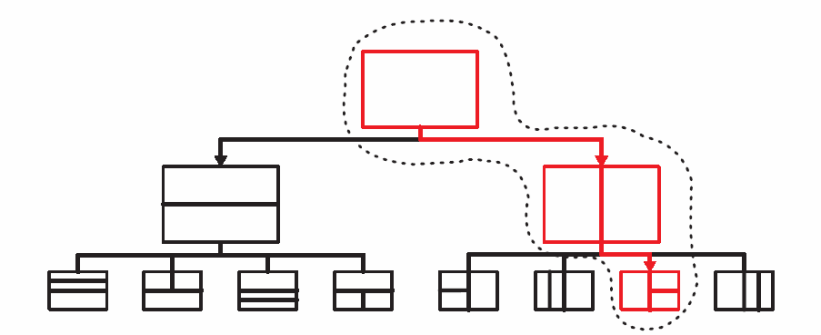
$A_i = [R_{k,i} \bullet L_j]$, where L_j is one of the input natural illumination images, and $B_i = b_{i,j}$

We found through experimentation that constraining the kernel weights to be positive is very important for the stability of the solution.

This over-constrained system can be efficiently solved using quadratic programming. It estimates the weights W_i that satisfy the system best in the least-squares sense.

Kernel Subdivision

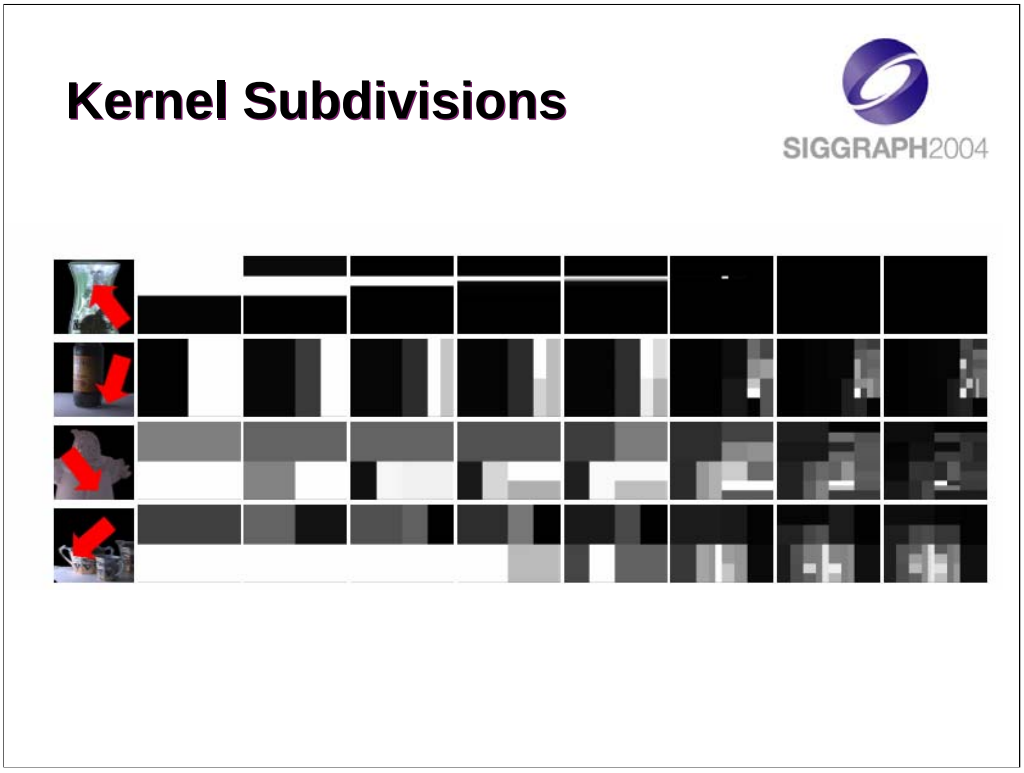
- Hierarchical quad-tree subdivision of the input image
- Typically k=25 kernels per output pixel b_i



To find the kernel positions and sizes for pixel *bi*, we use a hierarchical quad-tree subdivision of the input image to find them.

We first start with a kernel that occupies the whole image. We split it into two equal size rectangles. Having the option of splitting the kernel horizontally or vertically, we choose the subdivision that yields the lowest error. To compute this error we solve for the kernel weights using the equation in the previous slide. In the second iteration we have four possible splits of the two kernels from the first iteration, and so on.

The recursive subdivision stops if *K* kernels are computed (or *K*-1 subdivisions have been made).We found that *K* = 25 yields a good quality-vs.-time tradeoff



Here we show images of the progressive subdivision of the kernels for different scene points indicated by the red arrows. The kernels progress from left to right with 1, 2, 3, 4, 5, 10, 20, and 24 subdivisions.

The gray-scale colors of the kernels signify their weight (white is highest). Intuitively, these images show which light from the background is transported towards the pixel.

The first row shows how our algorithm quickly (after 10 subdivisions) identifies the small illumination kernel that is refracted by the glass. We identify one pixel (from the 800×600 input image) after 24 quad-tree subdivisions, which is quite remarkable.

The second row shows a primarily diffuse point in a hard shadow area. Note how the light transport comes mostly from the right as expected, and how a small reflective component is visible as well.

The third row shows a point with subsurface scattering. Light from a large area is scattered towards the viewer.

Finally, the last row shows a glossy point. The illumination enters the scene from the left (i.e., where the monitor was placed during acquisition). Note how the light is transported from almost concentric circular areas towards the incoming illumination.

Summary

- Quadratic programming to estimate weights W_i such that the error is minimized
- Quad-tree subdivision of the image to determine the optimal kernels $R_{k,i}$

$$W_i^{n+1} \longrightarrow R_{k,i}^{n+1} \longrightarrow W_i \text{ and } R_{k,i}$$

iterate

In summary, our algorithm iteratively proceeds as follows:

We first choose the whole image as a kernel and 1 as the initial weight. We then compute the error of this estimation compared with the actual observed images.

Assuming the error is too large, we subdivide the kernels and compute the minimum error according to the formula three slides back. This process yields a new set of weights and a set of new kernels.

The algorithm proceeds until the error of the estimation is below a certain threshold.

Relighting

- For each output pixel b_i:

$$b_i \approx \sum_k w_{k,i}\left(R_{k,i} \bullet L_i\right)$$

- Weights times the incident light from each kernel
- The incident illumination can be stored in a summed-are table to accelerate the computation

The pixel value of a single relit output pixel $B = [b_i]$ is:

$$b_i = T_i \bullet L$$

Note that the pixel value b_i is the inner product of two vectors T_i (reflectance function) and L (incident illumination).

Results

SIGGRAPH2004

- Single viewpoint, new illumination

Estimate

Actual

As a standard test of the quality of reflectance field estimation, we first compare the actual picture of the scenes under new illumination with the prediction obtained using our estimated reflectance field.

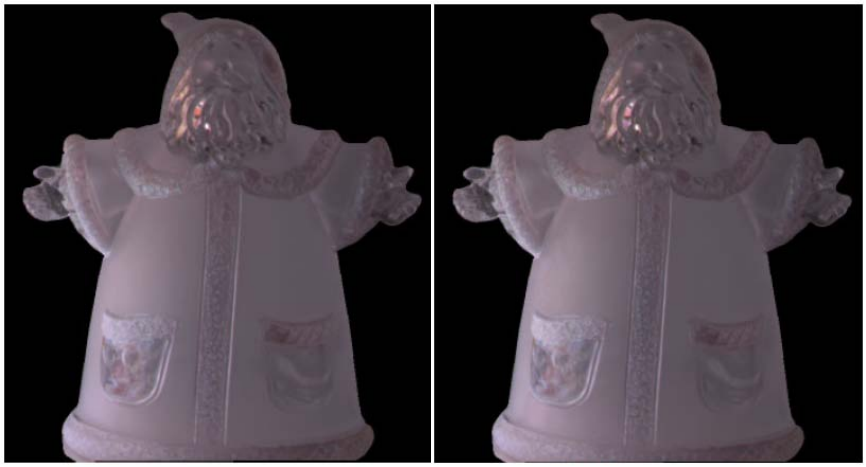
In general, our prediction works very well, for glossy, diffuse, and transparent objects.

Results

Estimate Actual

Diffuse elements (e.g., the label on the molasses jar) are reproduced correctly.

Results



Estimate Actual

The Santa model, which is hollow, shows both subsurface scattering and refractions.

Moving White Bar

- New synthetic illumination

Estimate Actual

Since our algorithm depends entirely on natural images, the most difficult test for our reflectance field prediction is to use synthetic input illumination with very high frequencies, such as images with white bars on a black background.

Here we show our predictions of the scenes with a vertical white bar, 100 pixels wide, sliding across the illumination domain. The glass clearly shows the greasy fingerprints, which corresponds well to reality. However, the movie contains a slight discontinuity due to the vertical subdivision used for the kernels.

3D Models

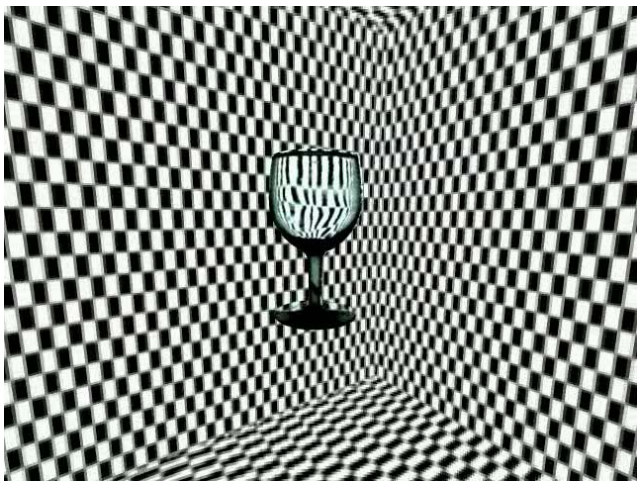
- Generate new images for camera views using this technique
- Interpolate images with ULR blending
- The following results were actually computed using an older technique [Matusik et al. 2002]

For 3D models we use the same procedure as before:

Generate new, relit images for each of the camera views and interpolate the results using ULR blending for each new viewpoint.

The following results were computed using an older technique that uses a combination of low-resolution reflectance fields and environment mattes (see [Matusik et al. 2002]).

Results



Note the refraction of the checkerboard texture in this scanned wine glass.

Transparent Objects

Here is our beer glass put into a new, synthetic environment. Note the refraction and the reflections.

Note: No beer was harmed to produce this model. We used apple juice, of course.

Transparent Objects

Note the reflection on the stem of the glass and the refraction.

Transparent Objects

A crystal sugar bowl. Of course, we do not know what this object really would look like in this environment.

Conclusions

- Image-based 3D photography is able to capture any type of object
- Point-based rendering offers high image-quality for display of complex models
- Opacity lightfields provide realistic 3D graphics models
- Robust reflectance field estimation allows us to integrated objects into new environments

We can draw the following conclusions from our work:

Image-based 3D photography is able to capture any type of object, as long as we are willing to acquire enough images.

Point-based rendering allows us to display these objects with maximum image quality without compromises due to texture resampling or geometry parameterization.

Opacity lightfields offer a simple alternative to having complex geometry, since they allow us to display objects with high complexity using an approximate shape model.

Finally, reflectance fields can be estimated robustly, which allows us to relight the objects and place them into arbitrary new environments.

Acknowledgements

- Colleagues:
  - MIT: Chris Buehler, Tom Buehler
  - MERL: Bill Yerazunis, Darren Leigh
- Thanks to:
  - David Tames, Jennifer Roderick Pfister
- Papers available at:
  http://www.merl.com/people/pfister/

We would like to thank these individuals who helped during this research.
Please visit my web page to download PDF files of our publications.

References

- [Debevec et al. 98] *Efficient View-Dependent Image-Based Rendering with Projective Texture Mapping.* In *Proceedings of the 9th Eurographics Workshop on Rendering*, 1998.
- [Levoy et al. 2000] *The digital Michelangelo project*, SIGGRAPH 2000.
- [Matusik et al. 2004] *Progressively-Refined Reflectance Functions from Natural Illumination,* Eurographics Symposium on Rendering 2004.
- [Matusik et al. 2002] *Image-based 3D photography using opacity hulls.* SIGGRAPH 2002, July 2002.
- [Matusik et al. 2002] *Acquisition and Rendering of Transparent and Refractive Objects*, Thirteenth Eurographics Workshop on Rendering, June 2002.
- [Pfister et al. 2000] *Surfels: Surface elements as rendering primitives*, SIGGRAPH 2000.
- [Pulli et al. 97] *View-based rendering: Visualizing real objects from scanned range and color data.* In *Eurographics Rendering Workshop 1997*.
- [Ren et al. 2002] *Object space EWA splatting: A hardware accelerated approach to high quality point rendering*, Eurographics 2002.
- [Smith & Blinn 96] *Blue screen matting.* Smith, A. R., and Blinn, J. F. 1996. In *Computer Graphics*, vol. 30 of *SIGGRAPH 96 Proceedings*.
- [Vlasic et al. 2003] *Opacity Light Fields: Interactive Rendering of Surface Light Fields with View-Dependent Opacity,* Proceedings of the Interactive 3D Graphics Symposium 2003, Monterey, April 2003.
- [Zongker et al. 99] *Environment matting and compositing*. In *Computer Graphics* (Aug. 1999), SIGGRAPH 99 Proceedings.
- [Zwicker et al. 2001] *Surface splatting*, SIGGRAPH 2001.
- [Zwicker et al. 2002] *EWA Splatting*, IEEE TVCG 2002.

[Debevec et al. 98] *Efficient View-Dependent Image-Based Rendering with Projective Texture Mapping.* In *Proceedings of the 9th Eurographics Workshop on Rendering*, 1998.

[Levoy et al. 2000] *The digital Michelangelo project*, SIGGRAPH 2000.

[Matusik et al. 2004] *Progressively-Refined Reflectance Functions from Natural Illumination,* Eurographics Symposium on Rendering 2004.

[Matusik et al. 2002] *Image-based 3D photography using opacity hulls.* SIGGRAPH 2002, July 2002.

[Matusik et al. 2002] *Acquisition and Rendering of Transparent and Refractive Objects*, Thirteenth Eurographics Workshop on Rendering, June 2002.

[Pfister et al. 2000] *Surfels: Surface elements as rendering primitives*, SIGGRAPH 2000.

[Pulli et al. 97] *View-based rendering: Visualizing real objects from scanned range and color data.* In *Eurographics Rendering Workshop 1997*.

[Ren et al. 2002] *Object space EWA splatting: A hardware accelerated approach to high quality point rendering*, Eurographics 2002.

[Smith & Blinn 96] *Blue screen matting.* Smith, A. R., and Blinn, J. F. 1996. In *Computer Graphics*, vol. 30 of *SIGGRAPH 96 Proceedings*.

[Vlasic et al. 2003] *Opacity Light Fields: Interactive Rendering of Surface Light Fields with View-Dependent Opacity,* Proceedings of the Interactive 3D Graphics Symposium 2003, Monterey, April 2003.

[Zongker et al. 99] *Environment matting and compositing*. In *Computer Graphics* (Aug. 1999), SIGGRAPH 99 Proceedings.

[Zwicker et al. 2001] *Surface splatting*, SIGGRAPH 2001.

[Zwicker et al. 2002] *EWA Splatting*, IEEE TVCG 2002.

SIGGRAPH2004

Point-based Computer Graphics

Point-based Surface Representations

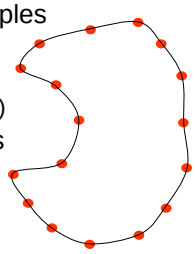
Point-based Surface Reps

Presenter

– Marc Alexa
– Discrete Geometric Modeling Group
– Darmstadt University of Technology
– alexa@informatik.tu-darmstadt.de

Motivation

- Many applications need a definition of surface based on point samples
  – Reduction
  – Up-sampling
  – Interrogation (e.g. ray tracing)
- Desirable surface properties
  – Manifold
  – Smooth
  – Local (efficient computation)



Overview

- Introduction & Basics
- Fitting Implicit Surfaces
- Surfaces from Local Frames

Overview

- Introduction & Basics
- Fitting Implicit Surfaces
- Surfaces from Local Frames

Introduction & Basics

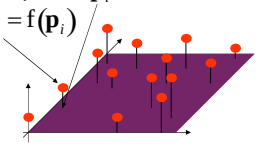
- Notation, Terms
  – Regular/Irregular, Approximation/Interpolation, Global/Local
- Standard interpolation/approximation techniques
  – Global: Triangulation, Voronoi-Interpolation, Least Squares (LS), Radial Basis Functions (RBF)
  – Local: Shepard/Partition of Unity Methods, Moving LS
- Problems
  – Sharp edges, feature size/noise
- Functional -> Manifold

Introduction & Basics

- Notation, Terms
  - Regular/Irregular, Approximation/Interpolation, Global/Local
- Standard interpolation/approximation techniques
  - Global: Triangulation, Voronoi-Interpolation, Least Squares (LS), Radial Basis Functions (RBF)
  - Local: Shepard/Partition of Unity Methods, Moving LS
- Problems
  - Sharp edges, feature size/noise
- Functional -> Manifold

Notation

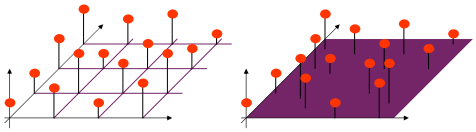
- Consider functional (height) data for now
- Data points are represented as
  - Location in parameter space $\mathbf{p}_i$
  - With certain height $f_i = \mathrm{f}(\mathbf{p}_i)$



- Goal is to approximate f from $f_i, \mathbf{p}_i$

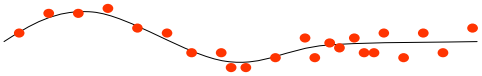
Terms: Regular/Irregular

- Regular (on a grid) or irregular (scattered)
- Neighborhood (topology) is unclear for irregular data



Terms: Approximation/Interpolation

- Noisy data ⇒ Approximation

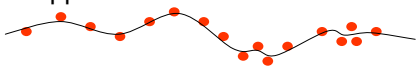


- Perfect data ⇒ Interpolation

Terms: Global/Local

- Global approximation

- Local approximation



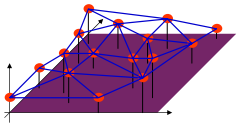
- Locality comes at the expense of fairness

Introduction & Basics

- Terms
  - Regular/Irregular, Approximation/Interpolation, Global/Local
- Standard interpolation/approximation techniques
  - Global: Triangulation, Voronoi-Interpolation, Least Squares (LS), Radial Basis Functions (RBF)
  - Local: Shepard/Partition of Unity Methods, Moving LS
- Problems
  - Sharp edges, feature size/noise
- Functional -> Manifold

Triangulation

- Exploit the topology in a triangulation (e.g. Delaunay) of the data
- Interpolate the data points on the triangles
  - Piecewise linear → C^0
  - Piecewise quadratic → C^1?
  - …



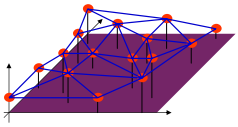
Triangulation: Piecewise linear

- Barycentric interpolation on simplices (triangles)
  - given point $\mathbf{x}$ inside a simplex defined by $\mathbf{p}_i$
  - Compute α_i from

$$\mathbf{x} = \sum_i \alpha_i \mathbf{p}_i \quad \text{and} \quad 1 = \sum_i \alpha_i$$

  - Then

$$f(\mathbf{x}) = \sum_i \alpha_i f_i$$



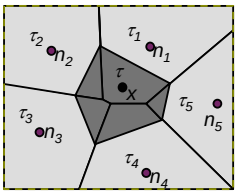
Voronoi Interpolation

- compute Voronoi diagram (dual of Delaunay triangulation)
- for any point $\mathbf{x}$ in space
  - add $\mathbf{x}$ to Voronoi diagram
  - Voronoi cell τ around $\mathbf{x}$ intersects original cells τ_i of natural neighbors n_i
  - interpolate $f(\mathbf{x}) = \sum_i \lambda_i(x) f_i \Big/ \sum_i \lambda_i(x)$

with $\lambda_i(\mathbf{x}) = \dfrac{|\tau \cap \tau_i|}{|\tau| \cdot \|\mathbf{x} - \mathbf{p}_i\|}$

Voronoi Interpolation

- Compute Voronoi diagram (dual of Delaunay triangulation)
- For any point $\mathbf{x}$ in space
  - Add $\mathbf{x}$ to Voronoi diagram
  - Compute weights from the areas of new cell relative to old cells
- Properties
  - Piecewise cubic
  - Differentiable, continous derivative



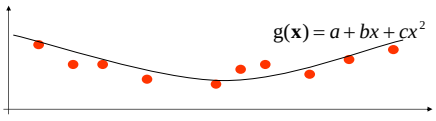
Voronoi Interpolation

Properties of Voronoi Interpolation:

- linear Precision
- local
- $f(\mathbf{x}) \in C^1$ on domain
- $f(\mathbf{x}, \mathbf{x}_1, \ldots, \mathbf{x}_n)$ is continuous in $\mathbf{x}_i$

Least Squares

- Fits a primitive to the data
- Minimizes squared distances between the $\mathbf{p}_i$'s and primitive g



$$\min_{g} \sum_i \left(f_i - g(\mathbf{p}_i)\right)^2$$

Least Squares - Example

- Primitive is a (univariate) polynomial

$$g(x) = (1, x, x^2, \ldots) \cdot \mathbf{c}^T$$

- $$\min \sum_i \left(f_i - (1, p_i, p_i^2, \ldots)\mathbf{c}^T \right)^2 \Rightarrow$$

$$0 = \sum_i 2 p_i^j \left(f_i - (1, p_i, p_i^2, \ldots)\mathbf{c}^T \right)$$

- Linear system of equations

Least Squares - Example

- Resulting system

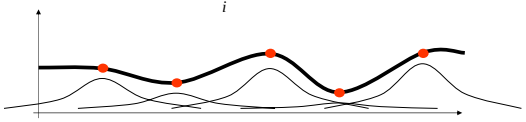
$$0 = \sum_i 2 p_i^j \left(f_i - (1, p_i, p_i^2, \ldots)\mathbf{c}^T \right) \Leftrightarrow$$

$$\sum_i \begin{pmatrix} 1 & p_i & p_i^2 & \cdots \\ p_i & p_i^2 & p_i^3 & \\ p_i^2 & p_i^3 & p_i^4 & \\ \vdots & & & \ddots \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \end{pmatrix} = 2 \sum_i f_i \begin{pmatrix} 1 \\ p_i \\ p_i^2 \\ \vdots \end{pmatrix}$$

Radial Basis Functions

- Represent approximating function as
  - Sum of radial functions r
  - Centered at the data points p_i

$$f(\mathbf{x}) = \sum_i w_i \, r(\|\mathbf{p}_i - \mathbf{x}\|)$$



Radial Basis Functions

- Solve $$f_j = \sum_i w_i r(\|\mathbf{p}_i - \mathbf{p}_j\|)$$

  to compute weights w_i
- Linear system of equations

$$\begin{pmatrix} r(0) & r(\|\mathbf{p}_0 - \mathbf{p}_1\|) & r(\|\mathbf{p}_0 - \mathbf{p}_2\|) & \cdots \\ r(\|\mathbf{p}_1 - \mathbf{p}_0\|) & r(0) & r(\|\mathbf{p}_1 - \mathbf{p}_2\|) & \\ r(\|\mathbf{p}_2 - \mathbf{p}_0\|) & r(\|\mathbf{p}_2 - \mathbf{p}_1\|) & r(0) & \\ \vdots & & & \ddots \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \end{pmatrix}$$

Radial Basis Functions

- Solvability depends on radial function
- Several choices assure solvability
  - $r(d) = d^2 \log d$ (thin plate spline)
  - $r(d) = e^{-d^2/h^2}$ (Gaussian)
    - h is a data parameter
    - h reflects the feature size or anticipated spacing among points

Function Spaces!

- Monomial, Lagrange, RBF share the same principle:
  - Choose basis of a function space
  - Find weight vector for base elements by solving linear system defined by data points
  - Compute values as linear combinations
- Properties
  - One costly preprocessing step
  - Simple evaluation of function in any point

Function Spaces?

- Problems
  - Many points lead to large linear systems
  - Evaluation requires global solutions
- Solutions
  - RBF with compact support
    - Matrix is sparse
    - Still: solution depends on every data point, though drop-off is exponential with distance
  - Local approximation approaches

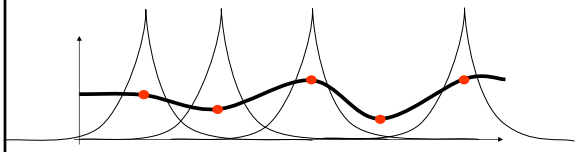
Introduction & Basics

- Terms
  - Regular/Irregular, Approximation/Interpolation, Global/Local
- Standard interpolation/approximation techniques
  - Global: Triangulation, Voronoi-Interpolation, Least Squares (LS), Radial Basis Functions (RBF)
  - Local: Shepard/Partition of Unity Methods, Moving LS
- Problems
  - Sharp edges, feature size/noise
- Functional -> Manifold

Shepard Interpolation

- Approach: $\mathrm{f}(\mathbf{x}) = \sum_i \phi_i(\mathbf{x}) f_i$

  with basis functions $\phi_i(\mathbf{x}) = \frac{\|\mathbf{x}-\mathbf{x}_i\|^{-p}}{\sum_j \|\mathbf{x}-\mathbf{x}_j\|^{-p}}$
- define $\mathrm{f}(\mathbf{p}_i) = f_i = \lim_{\mathbf{x}\to\mathbf{p}_i} \mathrm{f}(\mathbf{x})$



Shepard Interpolation

- $\mathrm{f}(\mathbf{x})$ is a convex combination of ϕ_i, because all $\phi_i \in [0,1]$ and $\sum \phi_i(\mathbf{x}) \equiv 1$
- $\mathrm{f}(\mathbf{x})$ is contained in the convex hull of data points
- $|\{\mathbf{p}_i\}| > 1 \Rightarrow \mathrm{f}(\mathbf{x}) \in \mathrm{C}^\infty$ and $\nabla \mathrm{f}(\mathbf{p}_i) = \mathbf{0}$
  - ➔Data points are saddles
- global interpolation
  - ➔every $\mathrm{f}(\mathbf{x})$ depends on all data points
- Only constant precision, i.e. only constant functions are reproduced exactly

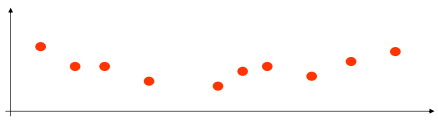
Shepard Interpolation

Localization:

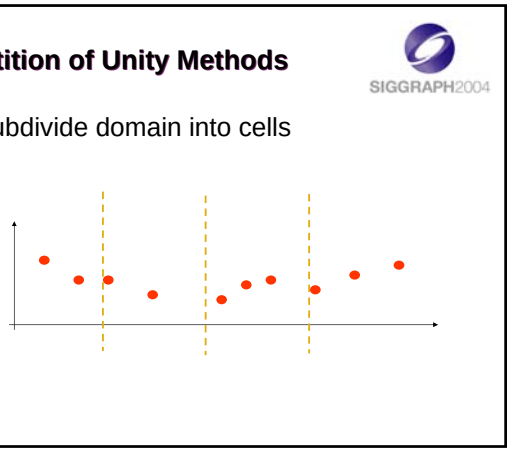
- Set $\mathrm{f}(\mathbf{x}) = \sum_i \mu_i(\mathbf{x}) \phi_i(\mathbf{x}) f_i$
- with $\mu_i(\mathbf{x}) = \begin{cases} \left(1 - \|\mathbf{x}-\mathbf{p}_i\|/R_i\right)^\nu & \text{if } \|\mathbf{x}-\mathbf{p}_i\| < R_i \\ 0 & \text{else} \end{cases}$

  for reasonable R_i and $\nu > 1$
  - ➔no constant precision because of possible holes in the data

Partition of Unity Methods

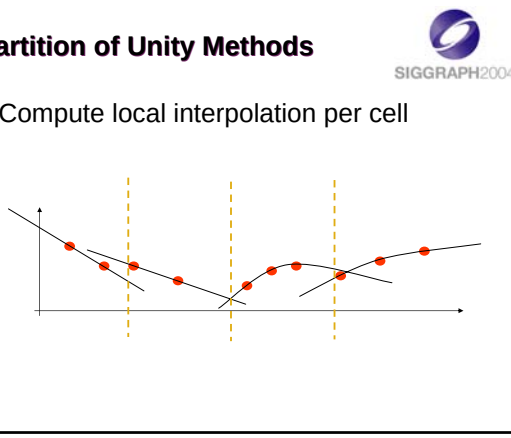


Partition of Unity Methods



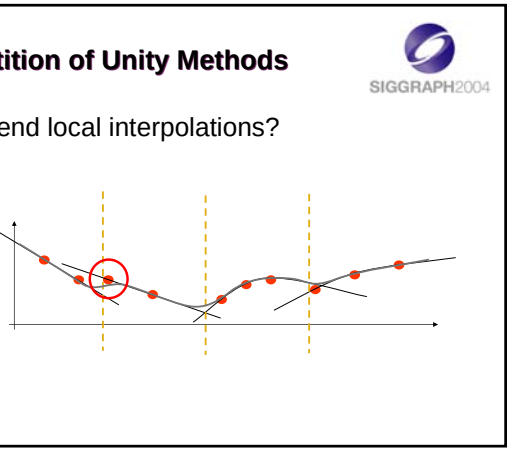
- Subdivide domain into cells

Partition of Unity Methods



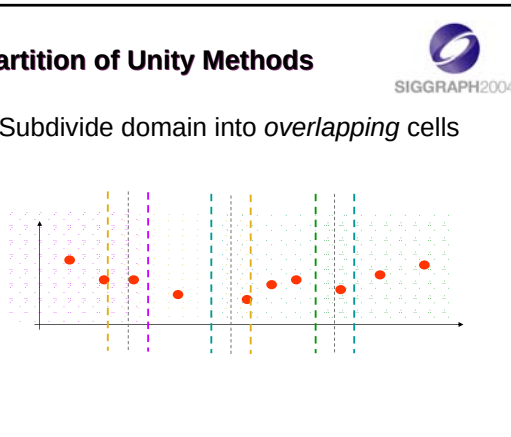
- Compute local interpolation per cell

Partition of Unity Methods



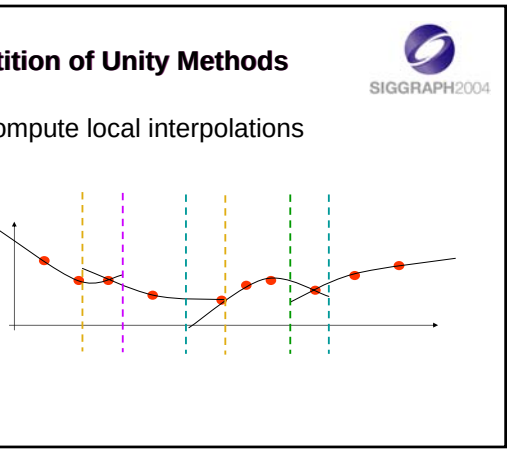
- Blend local interpolations?

Partition of Unity Methods



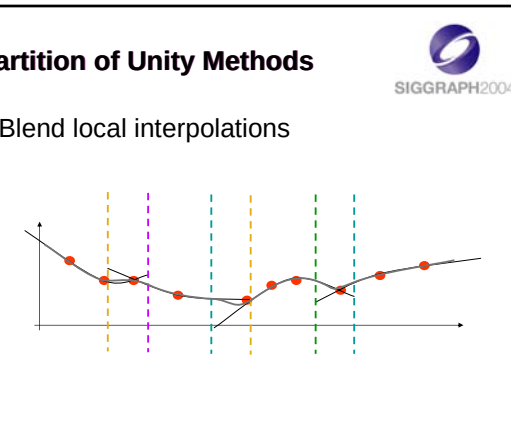
- Subdivide domain into *overlapping* cells

Partition of Unity Methods



- Compute local interpolations

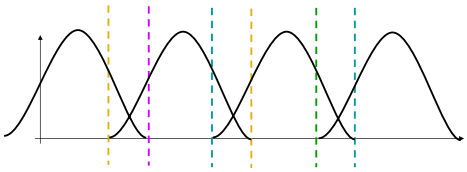
Partition of Unity Methods



- Blend local interpolations

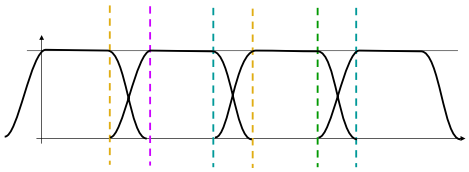
Partition of Unity Methods

- Weights should
  - have the (local) support of the cell



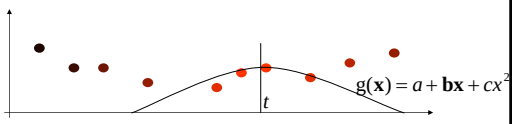
Partition of Unity Methods

- Weights should
  - sum up to one everywhere (Shepard weights)
  - have the (local) support of the cell



Moving Least Squares

- Compute a local LS approximation at $\mathbf{x}$
- Weight data points based on distance to $\mathbf{x}$



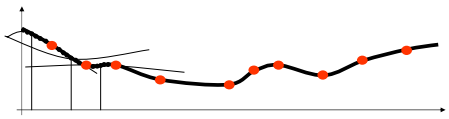
$$\min \sum_i \left(f_i - g(\mathbf{p}_i)\right)^2 \theta\left(\|\mathbf{x} - \mathbf{p}_i\|\right)$$

Moving Least Squares

- The set

$$f(\mathbf{x}) = g_{\mathbf{x}}(\mathbf{x}), g_{\mathbf{x}} : \min_g \sum_i \left(f_i - g(\mathbf{p}_i)\right)^2 \theta\left(\|\mathbf{x} - \mathbf{p}_i\|\right)$$

  is a smooth curve, iff θ is smooth



Moving Least Squares

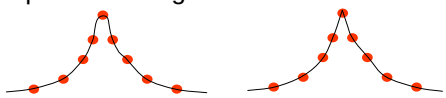
- Typical choices for θ:
  - $\theta(d) = d^{-r}$
  - $\theta(d) = e^{-d^2/h^2}$
- Note: $\theta_i = \theta\left(\|\mathbf{x} - \mathbf{p}_i\|\right)$ is fixed
- For each $\mathbf{x}$
  - Standard weighted LS problem
  - Linear iff corresponding LS is linear

Introduction & Basics

- Terms
  - Regular/Irregular, Approximation/Interpolation, Global/Local
- Standard interpolation/approximation techniques
  - Global: Triangulation, Voronoi-Interpolation, Least Squares (LS), Radial Basis Functions (RBF)
  - Local: Shepard/Partition of Unity Methods, Moving LS
- Problems
  - Sharp edges, feature size/noise
- Functional -> Manifold

Typical Problems

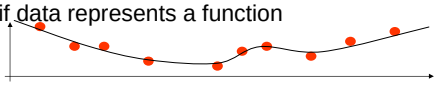
- Sharp corners/edges



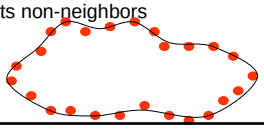
- Noise vs. feature size

Functional → Manifold

- Standard techniques are applicable if data represents a function



- Manifolds are more general
  - No parameter domain
  - No knowledge about neighbors, Delaunay triangulation connects non-neighbors



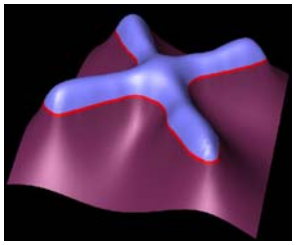
Overview

- Introduction & Basics
- Fitting Implicit Surfaces
- Surfaces from Local Frames

Implicits

- Each orientable 2-manifold can be embedded in 3-space
- Idea: Represent 2-manifold as zero-set of a scalar function in 3-space
  - Inside: $f(\mathbf{x}) < 0$
  - On the manifold: $f(\mathbf{x}) = 0$
  - Outside: $f(\mathbf{x}) > 0$

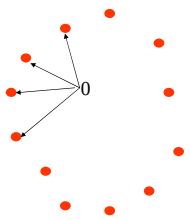
Implicits - Illustration



- Image courtesy Greg Turk

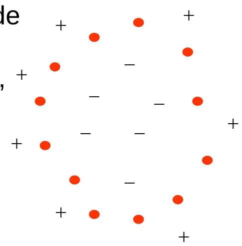
Implicits from point samples

- Function should be zero in data points
  - $f(\mathbf{p}_i) = 0$
- Use standard approximation techniques to find f
- Trivial solution: $f = 0$
- Additional constraints are needed



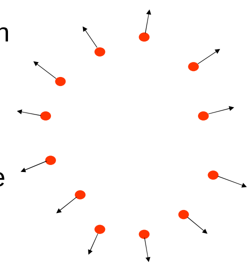
Implicits from point samples

- Constraints define inside and outside
- Simple approach (Turk, O'Brien)
  - Sprinkle additional information manually
  - Make additional information soft constraints



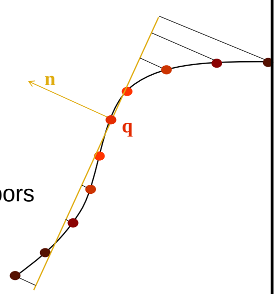
Implicits from point samples

- Use normal information
- Normals could be computed from scan
- Or, normals have to be estimated



Estimating normals

- Normal orientation (Implicits are signed)
  - Use inside/outside information from scan
- Normal direction by fitting a tangent
  - LS fit to nearest neighbors
  - Weighted LS fit
  - MLS fit

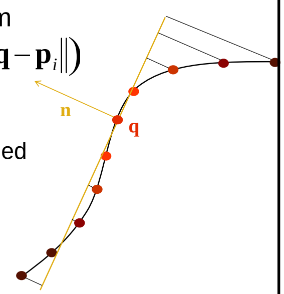


Estimating normals

- General fitting problem

$$\min_{\|\mathbf{n}\|=1} \sum_i \langle \mathbf{q}-\mathbf{p}_i, \mathbf{n} \rangle^2 \theta(\|\mathbf{q}-\mathbf{p}_i\|)$$

  - Problem is non-linear because $\mathbf{n}$ is constrained to unit sphere



Estimating normals

- The constrained minimization problem

$$\min_{\|\mathbf{n}\|=1} \sum_i \langle \mathbf{q}-\mathbf{p}_i, \mathbf{n} \rangle^2 \theta_i$$

is solved by the eigenvector corresponding to the smallest eigenvalue of the following co-variance matrix

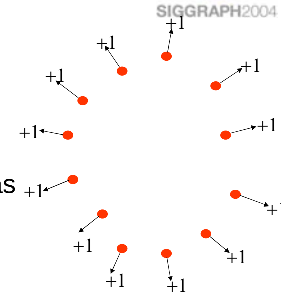
$$\sum_i (\mathbf{q}-\mathbf{p}_i)\cdot(\mathbf{q}-\mathbf{p}_i)^{\mathrm{T}} \theta_i$$

which is constructed as a sum of weighted outer products.

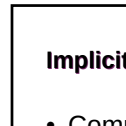
Implicits from point samples

- Compute non-zero anchors in the distance field
- Use normal information directly as constraints

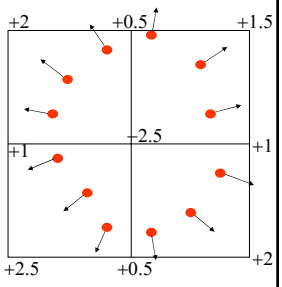
$$f(\mathbf{p}_i + \mathbf{n}_i) = 1$$



Implicits from point samples



- Compute non-zero anchors in the distance field
- Compute distances at specific points
  - Vertices, mid-points, etc. in a spatial subdivision



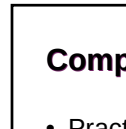
Computing Implicits

- Given N points and normals $\mathbf{p}_i, \mathbf{n}_i$ and constraints $f(\mathbf{p}_i) = 0, f(\mathbf{c}_i) = d_i$
- Let $\mathbf{p}_{i+N} = \mathbf{c}_i$
- An RBF approximation

$$f(\mathbf{x}) = \sum_i w_i \theta(\|\mathbf{p}_i - \mathbf{x}\|)$$

leads to a system of linear equations

Computing Implicits



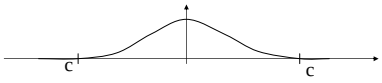
- Practical problems: $N > 10000$
- Matrix solution becomes difficult
- Two solutions
  - Sparse matrices allow iterative solution
  - Smaller number of RBFs

Computing Implicits

- Sparse matrices

$$\begin{pmatrix} \theta(0) & \theta(\|\mathbf{p}_0 - \mathbf{p}_1\|) & \theta(\|\mathbf{p}_0 - \mathbf{p}_2\|) & \cdots \\ \theta(\|\mathbf{p}_1 - \mathbf{p}_0\|) & \theta(0) & \theta(\|\mathbf{p}_1 - \mathbf{p}_2\|) & \\ \theta(\|\mathbf{p} - \mathbf{p}_0\|) & \theta(\|\mathbf{p}_2 - \mathbf{p}_1\|) & \theta(0) & \\ \vdots & & & \ddots \end{pmatrix}$$

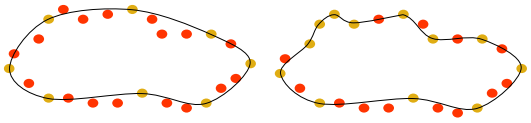
  - Needed: $d > c \rightarrow r(d) = 0, r'(c) = 0$



  - Compactly supported RBFs

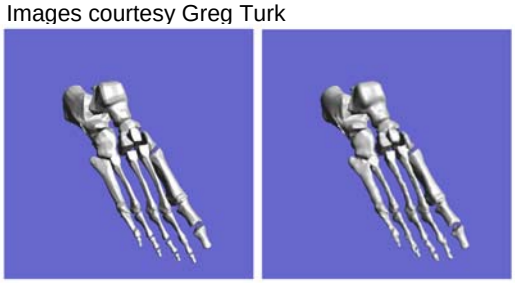
Computing Implicits

- Smaller number of RBFs
- Greedy approach (Carr et al.)
  - Start with random small subset
  - Add RBFs where approximation quality is not sufficient



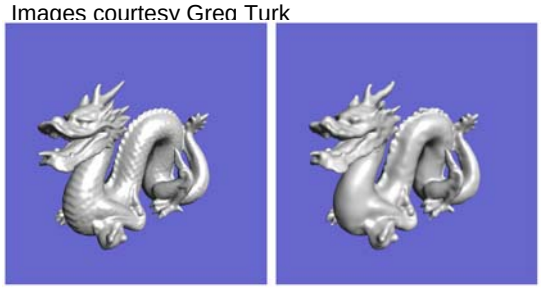
RBF Implicits - Results

- Images courtesy Greg Turk



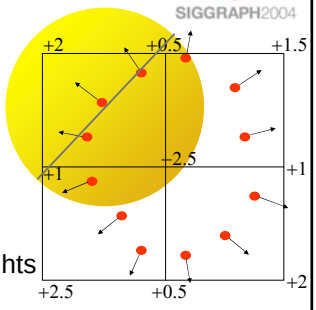
RBF Implicits - Results

- Images courtesy Greg Turk

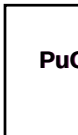


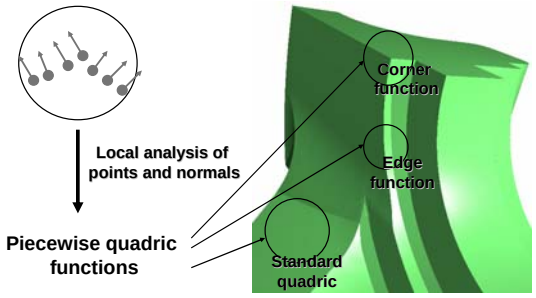
PuO Implicits

- Construct a spatial subdivision
- Compute local distance field approximations
  - e.g. Quadrics
- Blend them with local Shepard weights



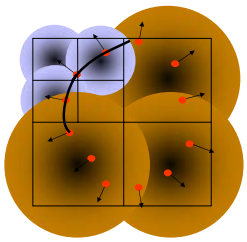
PuO Implicits: Sharp features





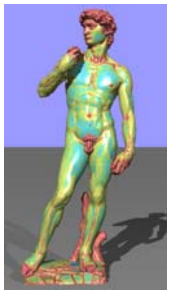
Multi-level PuO Implicits

- Subdivide cells based on local error



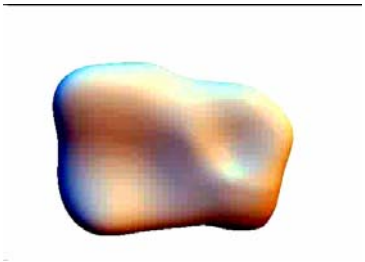
Multi-level PuO Implicits

- Local computations
  - Insensitive to number of points
- Local adaptation to shape complexity
- Sensitive to output complexity



Multi-level PuO Implicits

- Aproximation at arbitrary accuracy



Implicits - Conclusions

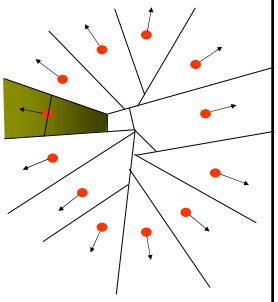
- Scalar field is underconstrained
  - Constraints only define where the field is zero, not where it is non-zero
  - Additional constraints are needed
- Signed fields restrict surfaces to be unbounded
  - All implicit surfaces define solids

Overview

- Introduction & Basics
- Fitting Implicit Surfaces
- Surfaces from Local Frames

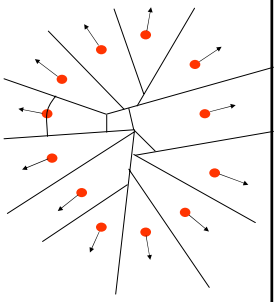
Hoppe's approach

- Use linear distance field per point
  - Direction is defined by normal
- In every point in space use the distance field of the closest point



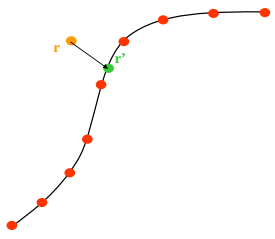
Hoppe's approach - smoother

- Direction fields are interpolated using Voronoi interpolation



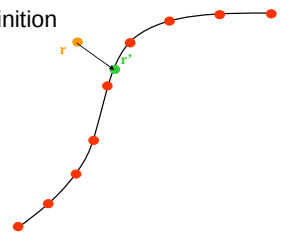
Projection

- Idea: Map space to surface
- Surface is defined as fixpoints of mapping



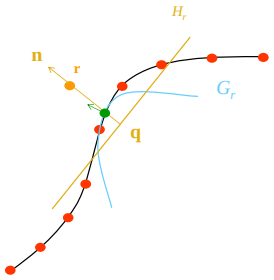
Surface definition

- Projection procedure (Levin)
  - Local polyonmial approximation
    - Inspired by differential geometry
  - "Implicit" surface definition
  - Infinitely smooth &
  - Manifold surface



Surface Definition

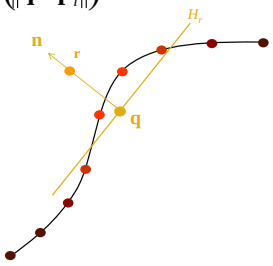
- Constructive definition
  - Input point **r**
  - Compute a local reference plane $H_r=\langle \mathbf{q},\mathbf{n}\rangle$
  - Compute a local polynomial over the plane G_r
  - Project point $\mathbf{r}'=G_r(0)$
  - Estimate normal



Local Reference Plane

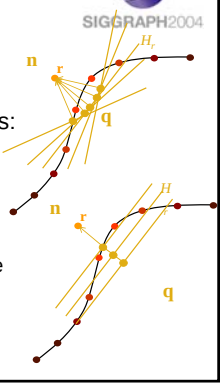
- Find plane $H_r=\langle \mathbf{q},\mathbf{n}\rangle + D$
  - $\min_{\mathbf{q},\|\mathbf{n}\|=1} \sum_i \langle \mathbf{q}-\mathbf{p}_i,\mathbf{n}\rangle^2 \theta(\|\mathbf{q}-\mathbf{p}_i\|)$
  - $\theta(\mathrm{d}) = e^{d^2/h^2}$
    - h is feature size/ point spacing
  - H_r is independent of r's distance
  - Manifold property

Weight function based on distance to q, not r



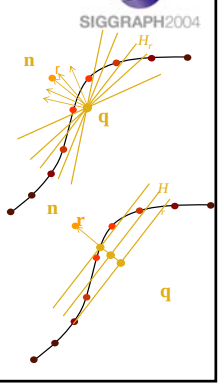
Local Reference Plane

- Computing reference plane
  - Non-linear optimization problem
- Minimize independent variables:
  - Over **n** for fixed distance $\|\mathbf{r}-\mathbf{q}\|$
  - Along n for fixed direction **n**
  - **q** changes → the weights change
  - Only iterative solutions possible



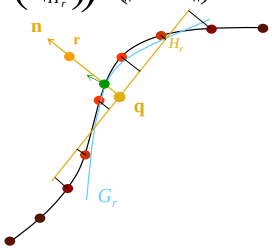
Local Reference Plane

- Practical computation
  - Minimize over **n** for fixed **q**
    - Eigenvalue problem
  - Translate **q** so that $\mathbf{r}=\mathbf{q}+\|\mathbf{r}-\mathbf{q}\|\mathbf{n}$
    - Effectively changes $\|\mathbf{r}-\mathbf{q}\|$
  - Minimize along **n** for fixed direction **n**
    - Exploit partial derivative



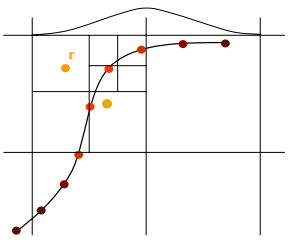
Projecting the Point

- MLS polyonomial over H_r
  - $\min_{G\in\Pi_d} \sum_i \left(\langle \mathbf{q}-\mathbf{p}_i,\mathbf{n}\rangle - G\left(\mathbf{p}_i|_{H_r}\right)\right)^2 \theta(\|\mathbf{q}-\mathbf{p}_i\|)$
  - LS problem
  - $\mathbf{r}'=G_r(0)$
  - Estimate normal



Spatial data structure

- Regular grid based on support of θ
  - Each point influences only 8 cells
- Each cell is an octree
  - Distant octree cells are approximated by one point in center of mass



Conclusions

- Projection-based surface definition
  - Surface is smooth and manifold
  - Surface may be bounded
  - Representation error mainly depends on point density
  - Adjustable feature size h allows to smooth out noise

SIGGRAPH2004

Point-Based Rendering

Matthias Zwicker, Computer Graphics Group, MIT

Point-Based Rendering

- Introduction and motivation
- Rendering by surface resampling
- Antialiasing by prefiltering
- Hardware implementation
- Applications
- Conclusions

This lecture starts with an introduction and motivation to point-based rendering. Next, point rendering is explained as a resampling process, which builds a conceptual framework for advanced rendering techniques. Basic point rendering algorithms are prone to aliasing artifacts. We present a prefiltering approach to avoid aliasing, which is based on our framework for surface resampling. We then present a hardware implementation of our rendering algorithm and we discuss various applications. We conclude the talk with a summary and directions for future research.

Motivation 1

Quake 2, 1998
10k triangles

Nvidia, 2002
millions of triangles

In recent years, we have witnessed a tremendous development of the computational power of computer graphics hardware. It was not possible to display scenes with more than couple of thousand triangles only a few years ago, which is illustrated on the left with a scene from Quake 2, a state-of-the art 3D game in 1998. Over only a few years, realism of interactive computer graphics has increased drastically. Recent hardware allows interactive rendering of highly complex surfaces, as shown for example in this demo by Nvidia on the right.

Motivation 1

- Performance of 3D hardware has exploded (e.g., GeForce FX: up to 338 million vertices per second, GeForce 6: 600 million vertices per second)
- Projected triangles are very small (i.e., cover only a few pixels)
- Overhead for triangle setup increases (initialization of texture filtering, rasterization)

Comparing the performance numbers of the last generations of GPUs, we can say that the performance of these processors has literally exploded. While, for example, the last generation of Nvidia GPUs, the GeForce FX, had a peak vertex rate of 338 million vertices per second, the current generation, the GeForce 6 that has been released this spring, has almost doubled this number to 600 million vertices per second. As a result of this huge processing power, the average projected size of triangles in typical scenes is becoming smaller and smaller. Since the image resolution has not increased significantly, the number of pixels covered by each triangle has decreased significantly. However, this means that the relative overhead for triangle setup, which includes the initialization of rasterization and texture filtering, has increased. It is amortized over fewer and fewer pixels, leading to a higher per pixel rendering cost.

Motivation 1

- Simplifying the rendering pipeline by unifying vertex and fragment processing

➡ A simpler, more efficient rendering primitive than triangles?

Triangle rendering has also become much more flexible and programmable, since modern GPUs allow the application of highly complex per-vertex and per-fragment programs. However, this requires separate functional units for vertex and fragment stages in the rendering pipeline. It would clearly be desirable to simplify the rendering pipeline by unifying the vertex and fragment processing stages. Point-based rendering has been developed with the goal to provide a rendering primitive that is more efficient than triangles for the highly complex geometry that is common today, and that allows the implementation of efficient and flexible rendering pipelines avoiding redundant functionality.

Motivation 2

- Modern 3D scanning devices (e.g., laser range scanners) acquire huge point clouds
- Generating consistent triangle meshes is time consuming and difficult

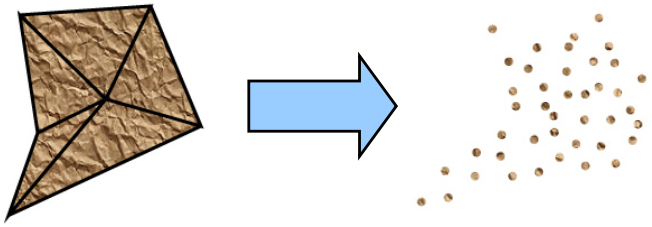
➡ A rendering primitive for direct visualization of point clouds, without the need to generate triangle meshes?

4 million pts.
[Levoy et al. 2000]

While we observe continued advances in the power of rendering and visualization systems, progress in 3D acquisition technology has also increased the complexity of available 3D objects. Modern 3D scanning devices such as laser range scanners acquire huge volumes of point data. Using such devices, the Digital Michelangelo Project by Marc Levoy and his group in Stanford has obtained highly detailed models of various statues by Michelangelo, for example the David statue that is displayed using about 4 million points on the right. To visualize this data using triangle rendering pipelines requires generating consistent triangle meshes, which is a time consuming and difficult process. Therefore, a rendering primitive that allows the direct visualization of unstructured point clouds, without the need to generate triangle meshes as a pre-process, seems very attractive.

Points as Rendering Primitives

- Point clouds instead of triangle meshes and textures as a rendering primitive [Levoy and Whitted 1985]
- 2D vector versus pixel graphics
- Points in 3D are analogous to pixels in 2D



triangle mesh with textures

point cloud, i.e., **nonuniform surface samples**

In this talk, I will present techniques that use points as an alternative to triangles as a fundamental rendering primitive. The idea of using points instead of triangle meshes and textures has first been proposed by Levoy and Whitted in a pioneering report in 1985. Think of the difference between points and triangles in 3D similar as of the difference between pixels and vector graphics in 2D. Points in 3D are analogous to pixels in 2D, replacing textured triangles or higher order surfaces by zero-dimensional elements.

Point-Based Surface Representation

- Points are ***nonuniform samples*** of the surface
- The point cloud describes:
  - 3D geometry of the surface
  - Surface reflectance properties (e.g., diffuse color, etc.)
- Points discretize geometry and appearance at the same rate
- There is no additional information, such as
  - connectivity (i.e., explicit neighborhood information between points)
  - texture maps, bump maps, etc.

Essentially, points are nonuniformly distributed samples of 3D surfaces. A point cloud describes the 3D geometry of the surface, including surface position and normal, and the surface reflectance properties, for example its diffuse color, or even a full BRDF at each point. This means that points are a purely discrete representation of the surface, in fact discretizing the geometry and appearance of the surface at the same rate. Note that in a point-based object representation, there is no additional information such as connectivity, i.e., explicit neighborhood information between points, or texture maps, bump maps, and so on.

Model Acquisition

- 3D scanning of physical objects [Matusik et al. 2002, Levoy et al. 2000]
  - See Pfister, acquisition
  - Direct rendering of acquired point clouds
  - No mesh reconstruction necessary

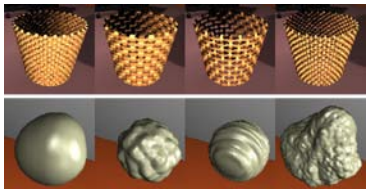
[Matusik et al. 2002]

How are point-based models acquired? Point-based surface representations arise naturally as the raw data produced by 3D scanning devices. Many such systems have been developed recently, making the direct visualization of acquired point clouds desirable and attractive. For example, the acquisition device developed by Matusik et al. is capable of scanning surfaces that are geometrically very complex, such as those shown in the images here. The images are generated using a point rendering algorithm with surface reflectance field shading. Using the points directly as rendering primitives makes mesh reconstruction unnecessary, which would be problematic for the fur of the bear or the feather boa of the angel.

Model Acquisition

- Sampling synthetic objects
  - Efficient rendering and simple LOD of complex models (trees, plants)
  - Dynamic sampling of procedural objects and animated scenes


[Zwicker et al. 2001]

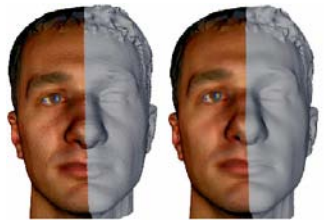

[Stamminger et al. 2001]


[Deussen et al. 2001]

Point-based surfaces can also be acquired by sampling synthetic objects, as proposed for example by Pfister et al. and Deussen et al. The goal of these approaches is to render highly complex objects such as trees, plants, or whole ecosystems, as efficiently as possible. In particular, points allow a very simple construction of level-of-detail representations. Points can also be used to dynamically sample and directly visualize procedural objects as described by Stamminger et al.

Model Acquisition

- Processing and editing of point-sampled geometry (see Pauly, Gross, afternoon session)
- Point-based content creation pipeline



spectral processing
[Pauly, Gross 2002]
(Gross, spectral processing)

point-based surface editing
[Zwicker et al. 2002]
(Pauly, Pointshop3D)

Shape modeling
[Pauly et al. 2003]
(Pauly, Pointshop3D)

Finally, recent research has extended the scope of point-based surface representations to include processing, editing, and modeling operations. These efforts provide a wide variety of operations to modify point-based surfaces. They will be described in more detail in the talks by Mark Pauly and Markus Gross later in this course. In these techniques, point-based rendering provides the back-end for interactive visualization of the modified objects. Together, the techniques provide a complete point-based content creation pipeline that never requires the reconstruction of a triangle mesh or higher order surface.

Rendering by Resampling

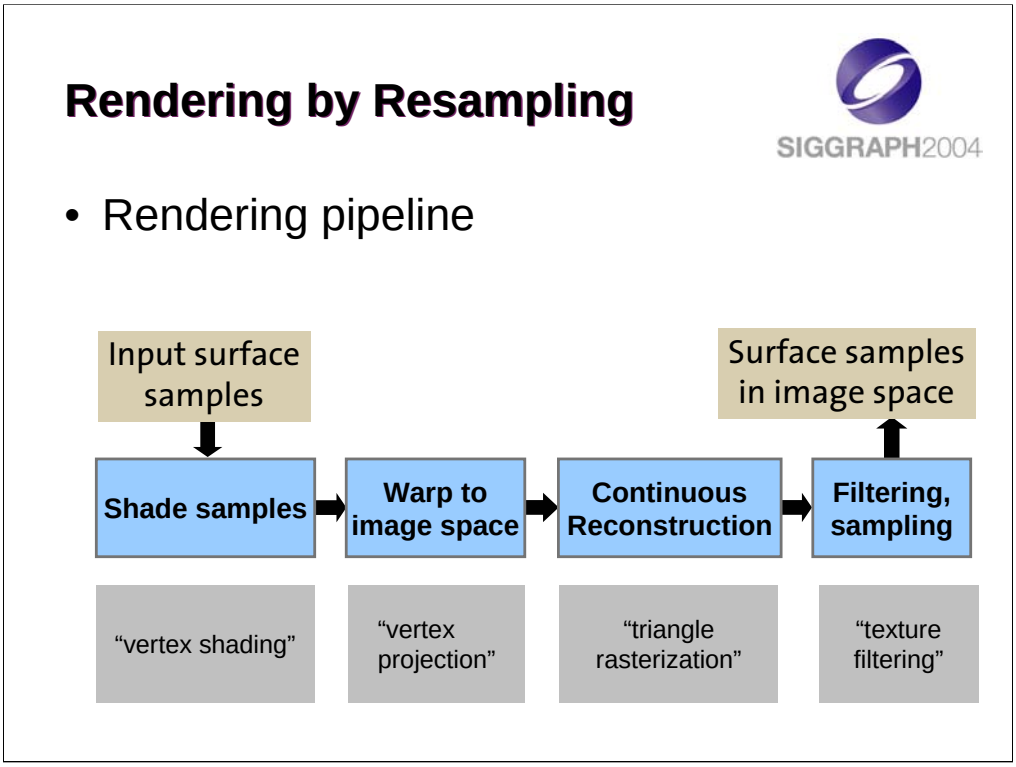
Input: nonuniform set of *surface samples*

Resampling: rendering algorithm

Output: *surface samples* on a regular image sampling grid

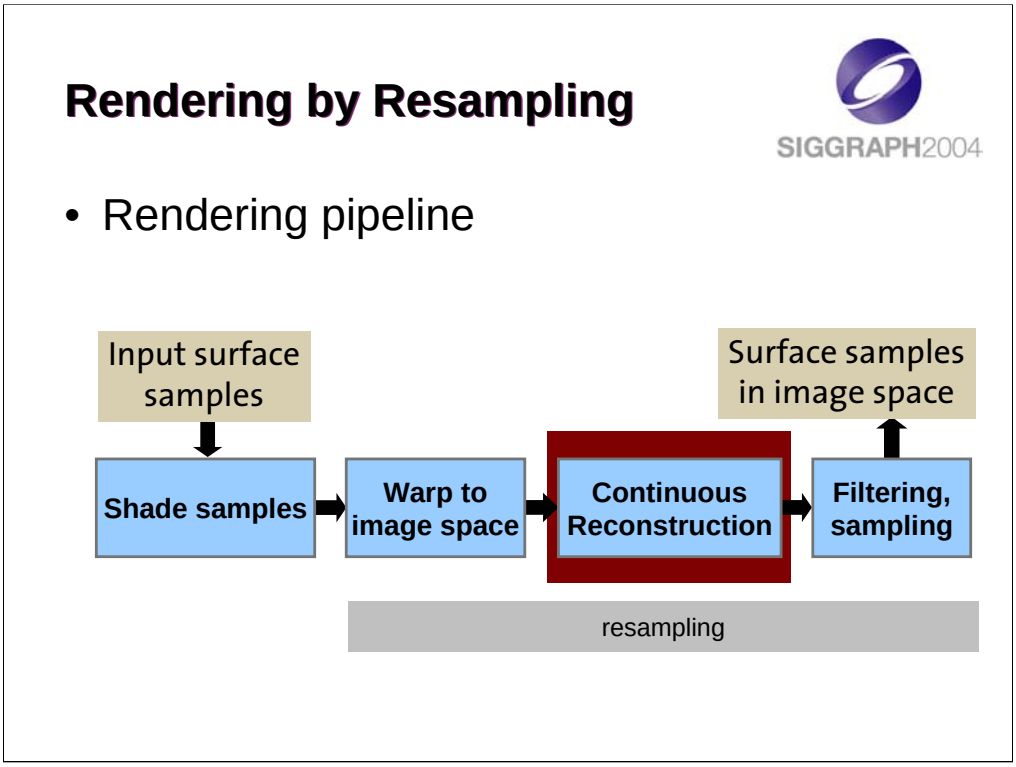
- Resampling involves reconstruction, filtering, and sampling
- The resampling approach prevents artifacts such as holes and aliasing

In this talk, I will introduce point rendering as a resampling process: As an input, we are given an unstructured point cloud, or a nonuniform set of surface samples. The output of the rendering system is again a set of surface samples, but distributed on the regular grid of the output image. The rendering algorithm converts the initial surface samples to new samples at the output positions. Hence rendering is a resampling process that consists of reconstructing a continuous surface from the input samples, filtering the continuous representation, and sampling it, or evaluating it, at the output positions. In this talk I will show how this resampling approach prevents artifacts such as holes or aliasing in point rendering algorithms.



The point rendering pipeline processes point data as follows: Input surface samples are first shaded according to any local illumination model. This stage is comparable to vertex shading in triangle rendering pipelines. In the next stage, the points are projected, or warped, to image space. Again, this is analogous to projecting triangle vertices to image space. Next, a continuous surface is reconstructed in image space, which is similar to triangle rasterization. The final stage in the pipeline is filtering and sampling the continuous representation at the output pixel positions, which is is similar to texture filtering and sampling.

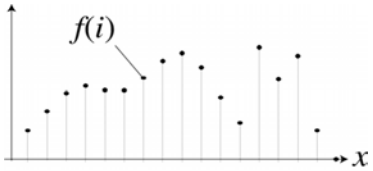
Rendering by Resampling



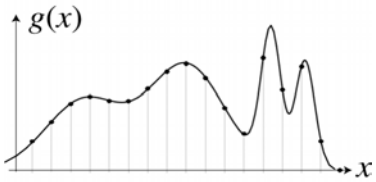
I will now focus on the final three stages of the pipeline, which essentially implement the resampling process. First, I will explain our method for surface reconstruction.

Continuous Reconstruction

- For uniform samples, use signal processing theory
- Reconstruction by convolution with low-pass (reconstruction) filter
- Exact reconstruction of band-limited signals using ideal low-pass filters



$$\tilde{f}(x) = \sum_i f(i)\delta(x-i)$$

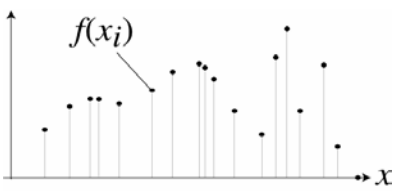


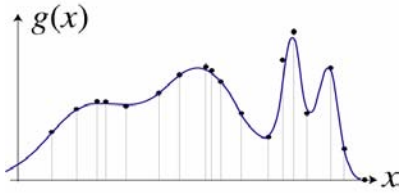
$$g(x) = (\tilde{f} \otimes h^{1/2})(x)$$
$$= \sum_i f(i)h^{1/2}(x-i)$$

The continuous reconstruction of uniformly sampled functions, or signals, is well understood and described by signal processing theory. Continuous reconstruction is achieved by convolving the sampled signal with a reconstruction or low-pass filter. On the left, the sampled signal tilde(f) is represented as a sum of scaled impulse signals, where f(i) are the samples of the original function. Convolving this with a reconstruction filter h as shown on the right leads to the representation of the reconstructed signal g as a weighted sum of shifted reconstruction filter kernels. One of the major and most practical results of signal processing theory, the so-called sampling theorem, states that band-limited signals can be exactly reconstructed using ideal low-pass filters.

Continuous Reconstruction

- Signal processing theory not applicable for **nonuniform** samples
- Local weighted average filtering
  - Normalized sum of **local** reconstruction kernels





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

$$g(x) = \frac{\sum_i f(x_i) r_i(x - x_i)}{\sum_i r_i(x - x_i)}$$

Unfortunately, signal processing theory is not applicable to nonuniform samples, and it is a non-trivial question what would be the best way to reconstruct a continuous signal from nonuniform samples. Our approach shown on the right is inspired by signal processing theory. We call it local weighted average filtering. The nonuniformly sampled signal is represented by samples f of the original signal at arbitrary positions x_i. We reconstruct a continuous signal g by building a weighted sum of reconstruction kernels r, similar as in the uniform case. However, we use a locally different kernel for each sample position, so this does not correspond to a convolution. Further, we normalize the reconstructed value at each position by the sum of the reconstruction kernels.

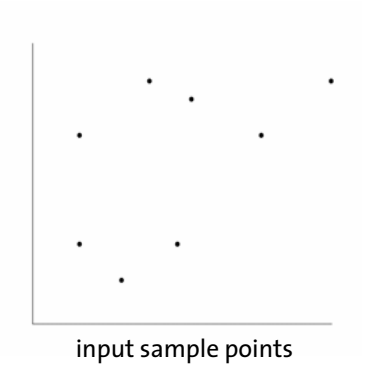
Nonuniform Reconstruction

- Local weighted average filtering
  - Simple
  - Efficient
  - No guarantees about reconstruction error
- Normalization division ensures perfect flat field response
- Choice of reconstruction kernels based on local sampling density [Zwicker 2003]

This local weighted average scheme is very simple and efficient, in particular if the reconstruction kernels have a local support. However, note that it does not guarantee any bounds on the reconstruction error, as signal processing theory does in the uniform case. Further, normalization is a crucial component of the scheme, since it ensures that the reconstruction has a perfect flat field response. This means that reconstructing a constant signal is done exactly and without any error. This is important to avoid visual artifacts. The reconstruction kernels are determined in a pre-process based on the local sampling density. However, a discussion of techniques to do so is outside the scope of this talk.

Parameterized Reconstruction

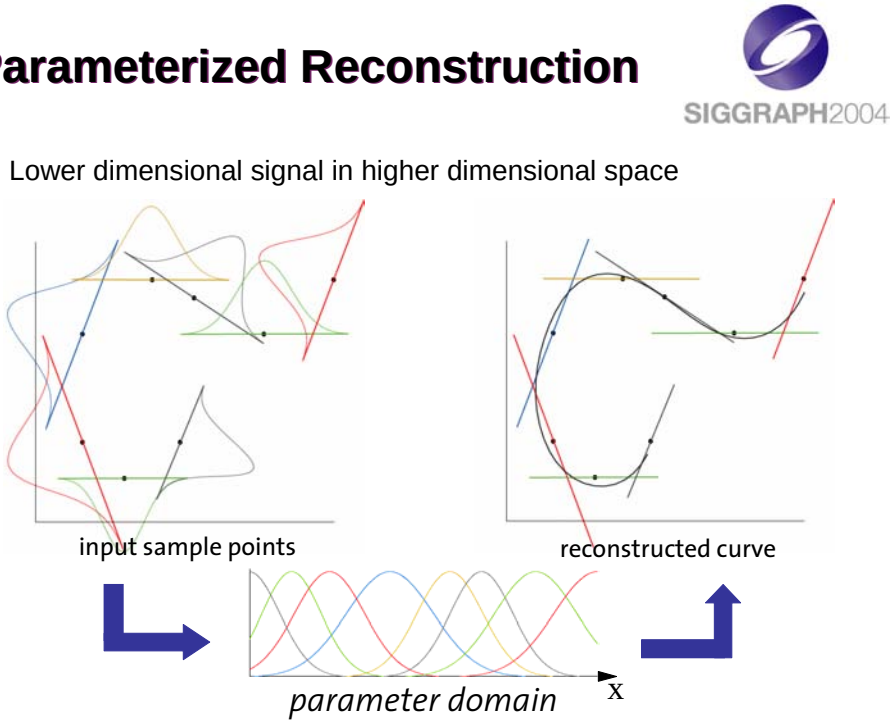
- Lower dimensional signal in higher dimensional space



common frame?

So far I have been discussing the reconstruction of signals in a functional setting. However, surfaces must be interpreted as lower dimensional signals in a higher dimensional space. In our case, these would be 2D surfaces embedded in 3D Euclidean space. Note that to perform 2D surface reconstruction as described before, we need a global, common 2D coordinate frame. Our solution is to construct such a global frame using a parameterization of the input points. I will illustrate this idea using a 1D curve reconstructed in the 2D plane. First, let us define a local frame for each point, and let us define the reconstruction kernel associated with each point in coordinates of the local frame. Note that the orientation of the local frame and the reconstruction kernel for each point is different.

Parameterized Reconstruction



We then use a parameterization to map all the local frames into one global frame of reference. The global frame of reference is also called the global parameter domain. This means that for each local reference frame we define a parameter mapping that converts local coordinates to coordinates of the global parameter domain. I will later explain how this mapping from local to global coordinates is computed for point rendering. Given all the reconstruction kernels in the parameter domain, the reconstruction equation can be evaluated as in the functional setting. However, the reconstructed curve is not a continuous 1D function, but a parameterized 1D curve embedded in the 2D plane.

Parameterized Reconstruction

- Reconstruction yields a parameterized curve

global parameter coordinate

sample value

reconstruction kernel

$$g(x) = \frac{\sum_i f(x_i) r_i(\varphi_i^{-1}(x))}{\sum_i r_i(\varphi_i^{-1}(x))}$$

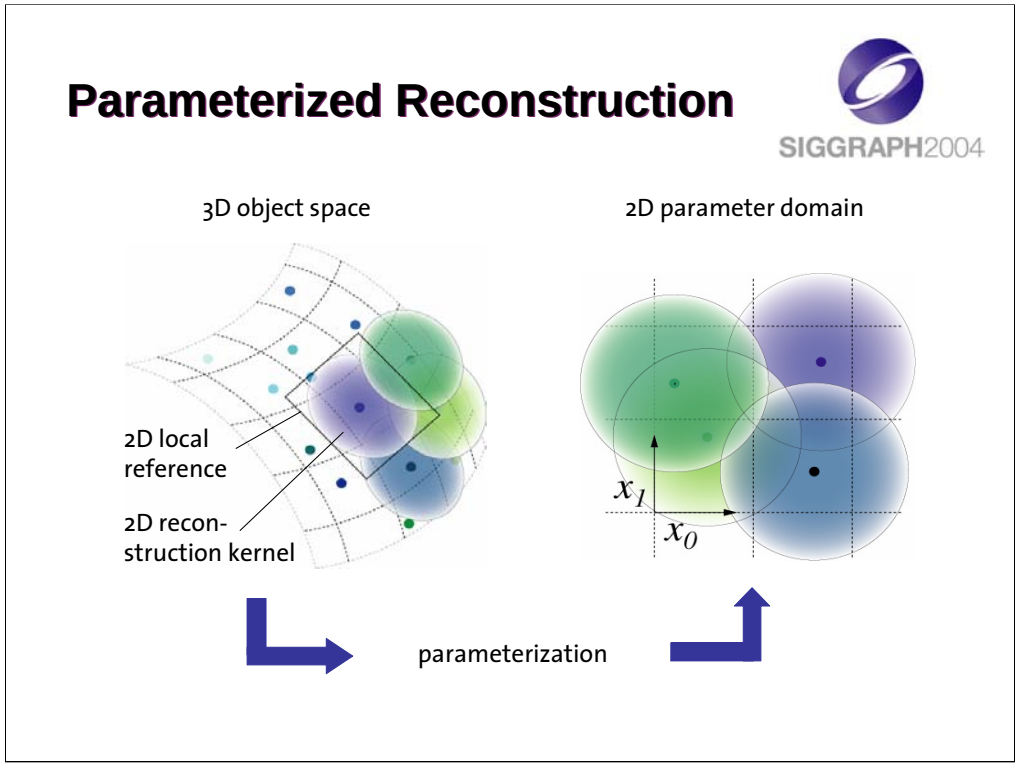
Parameterization, i.e., mapping to global parameter domain

Mathematically, the reconstruction of a parameterized curve can be expressed easily by plugging in the parameter mappings into the reconstruction equation for the functional setting. Here, x denotes the global parameter coordinate, and phi_i^(-1)(x) are the mappings from the local frames to the global domain.

Parameterized Reconstruction

- Surface samples are extended with specification of local reference frame
- Usually given by surface normal
- Reconstruction of surfaces includes reconstruction of all surface attributes
  - Color
  - Position
  - Normal

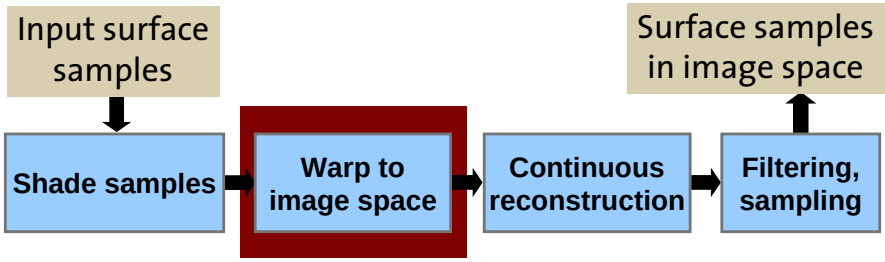
To apply this scheme to point-sampled surfaces, we need to extend the surface samples by a specification of the local reference frame. Usually, the local reference frame is defined to be perpendicular to the surface normal. Further, note that the reconstruction of surfaces includes the reconstruction of all surface attributes such as its color, position, and normal. All these attributes can be interpolated using the expression shown in the last slide.



This slides illustrates the idea of parameterized reconstruction for 2D surfaces in 3D. A 2D local reference plane is defined for each surface sample in 3D, and the 2D reconstruction kernel associated with each sample is defined on this plane. A parameter mapping for each local reference domain transforms the reconstruction kernel to a global 2D parameter domain with coordinates x_0 and x_1. Local weighted average filtering for all surface attributes is then performed in this global parameter domain.

Rendering by Resampling

- Rendering pipeline



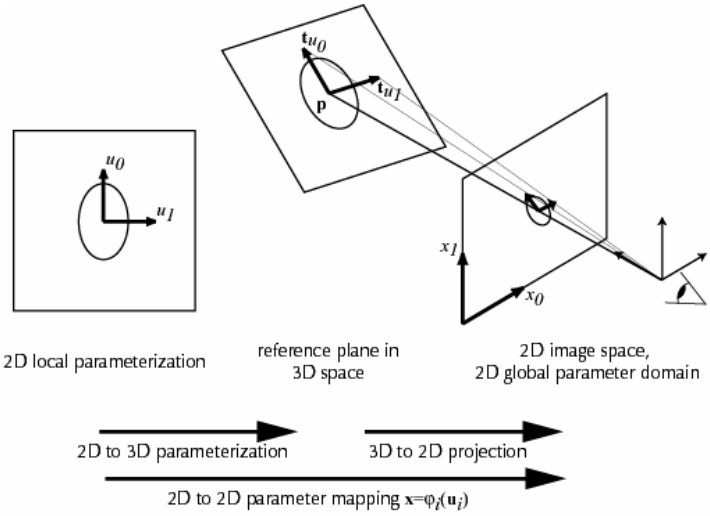
Now that I have established a method to reconstruct a continuous surface from the discrete samples, I will explain how it is applied in the point rendering pipeline. The crucial step to do this is the warping or the projection of the input sample points and their attributes to image space.

Warping

- Use image space as global parameter domain
- Warp local reference domains to image space using a 2D to 2D projective mapping
- Compute weighted sum of reconstruction kernels in image space

The link between parameterized reconstruction described before and point rendering is the idea to use image space as the global parameter domain. The parameter mappings from local to global coordinates are then given by the warp of the local reference domains to image space using a 2D to 2D projective mapping. Reconstruction is performed by computing the weighted sum of reconstruction kernels in image space.

Warping



The idea of warping the local reference planes to image space and using image space as the global parameter domain is again illustrated on this slide. The reconstruction kernels are initially defined in coordinates u_0, u_1 of the local reference plane shown on the left. The reference planes are embedded in 3D space using an anchor point p and tangent vectors t_u_0 and t_u_1, as shown in the middle. Note that the index i specifying an input point is omitted here to simplify the notation. A 2D to 2D mapping, or a 2D to 2D correspondence, is then established by perspectively projecting these planes to image space with coordinates x_0 and x_1, shown on the right. Hence, the pose of the tangent plane and its perspective projection determine the 2D to 2D parameter mapping x=phi_i(u_i).

Warping

- The projective mapping from tangent coordinates to image coordinates
  - Mapping to clip space with coordinates x_0', x_1', x_2', x_3'

$$\begin{bmatrix} x_0'(u,v) \\ x_1'(u,v) \\ x_2'(u,v) \\ x_3'(u,v) \end{bmatrix} = \mathbf{M} \cdot \begin{bmatrix} \mathbf{t}_{u0} & \mathbf{t}_{u1} & \mathbf{p} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ 1 \end{bmatrix}$$

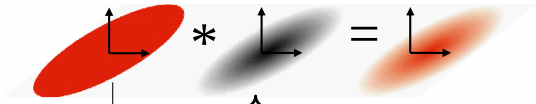
    where **M** is the 4x4 compound modelview-projection matrix
  - Projection (division by homogeneous coordinate)

$$\mathbf{x} = \varphi(\mathbf{u}) = \left[\frac{x_0'(u_0,u_1)}{x_3'(u_0,u_1)}, \frac{x_1'(u_0,u_1)}{x_3'(u_0,u_1)} \right]$$

The projective mapping from local tangent coordinates to image coordinates is easily expressed using the following matrix notation. First, a point on the tangent plane is mapped to so-called clip space. This is done by transforming a point u_0, u_1 to 3D using the definition of the tangent plane, which is expressed as a matrix vector multiplication here. This point is then mapped to clip space by multiplying it with the 4x4 modelview-projection matrix M, analogous to transforming triangle vertices. Projection to image coordinates is then performed by the division through the homogeneous coordinate x_3.

Warping

- Reconstruction kernels in image space are also called “footprints” or “splats”



$$g(j,k) = \frac{\sum_i f_i r_i(\varphi_i^{-1}(j,k))}{\sum_i r_i(\varphi_i^{-1}(j,k))}$$

output pixel coordinates

perspective projection of reference domain i

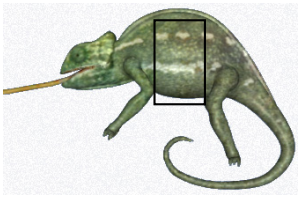
The projective 2D to 2D mapping can now be plugged into the reconstruction equation. To simplify explanations, we focus on the reconstruction of the surface color from now on. The color at an output pixel (j,k) is then given by the weighted sum of sample colors f_i multiplied by the value of the warped reconstruction kernels, normalized by the sum of the warped reconstruction kernels at the pixel. The projective mapping phi_i is derived as explained in the slide before. A warped reconstruction kernel in image space multiplied with a sample color is also called a “footprint” or “splat”.

Splatting Algorithm

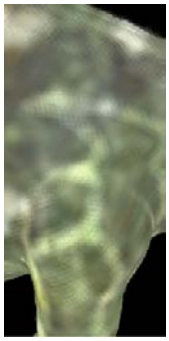

```
for each sample point {
  shade surface sample;
  splat = projected reconstruction
  kernel;
  rasterize and accumulate splat;
}
for each output pixel {
  normalize;
}
```

Hence, we also call the resulting rendering algorithm a splatting algorithm. This algorithm proceeds by iterating over all sample points: First, the point is shaded using some local illumination model. Then, the footprint or splat is computed by projecting the reconstruction kernel to image space. The splat is now rasterized and accumulated in the framebuffer. After all samples have been processed, all the values in the framebuffer have to be normalized by the accumulated weights of the warped reconstruction kernels.

Normalization



without normalization



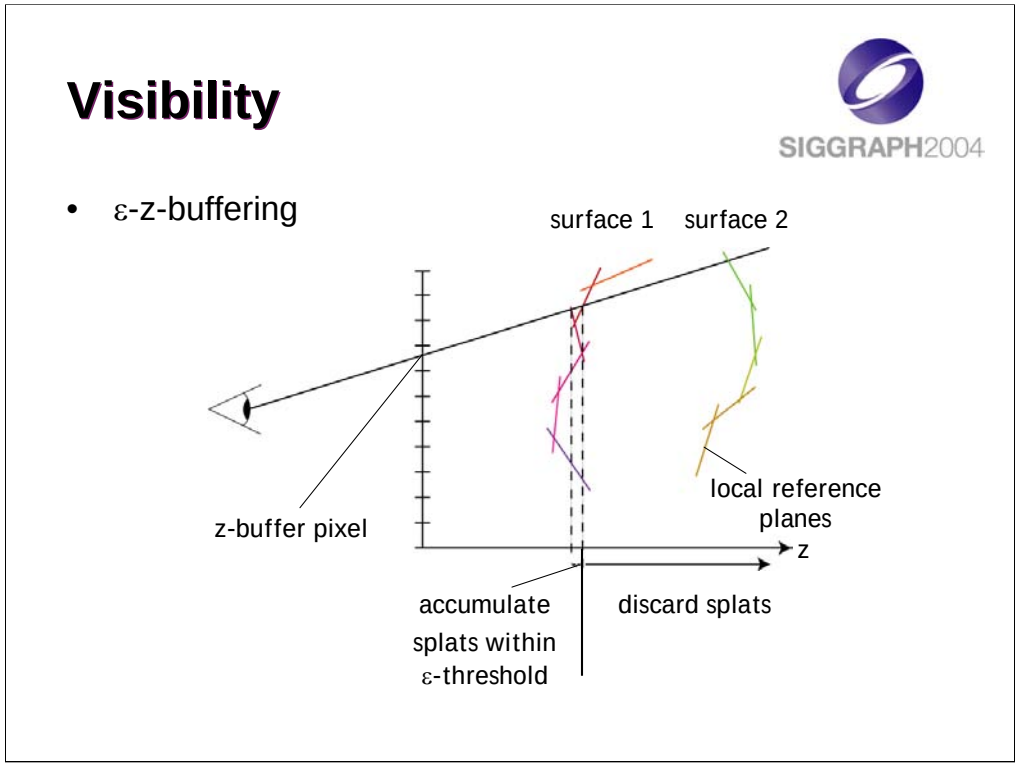
varying brightness because of irregular point distribution

with normalization

no artifacts

I want to point out two important aspects of this splatting algorithm: First, observe that the normalization pass is essential to avoid visual artifacts. Since the input points are nonuniformly distributed in 3D space, it is in general not possible to choose reconstruction kernels that sum up to one everywhere, i.e. that provide a partition of unity. This leads to a varying brightness across the reconstructed surface, as is illustrated in the close-up in the middle. Normalization guarantees a perfect flat field response of the reconstruction scheme, therefore avoiding brightness variations.

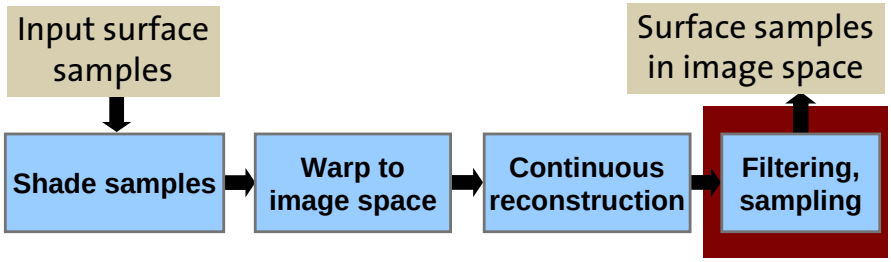
Visibility



The second issue is determining the visibility of the splats in image space. In other words, we need to determine which splats belong to the visible surface in a given pixel and therefore should be accumulated, and which ones should be discarded because they belong to an occluded surface. We solve this problem using an extended z-buffering scheme, sometimes also called epsilon-z-buffering. The idea is that in each pixel, all those splats are visible that lie within a range epsilon from the front-most contribution of any splat. All splats that have depth values that are further away than the front-most splat plus the threshold epsilon are considered to belong to a different surface, and they are discarded.

Rendering by Resampling

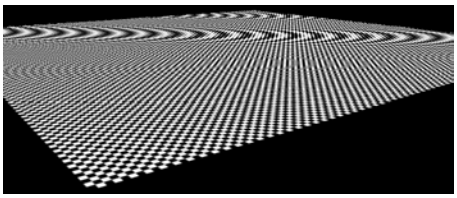
- Rendering pipeline



Now that we have reconstructed the point-based surface in image space, the final step of the rendering pipeline is to sample the reconstructed signal at the output pixel locations. However to achieve high image quality, it turns out that we need to apply some filtering to the reconstructed signal before sampling it. I will focus on this issue in the next part of the talk.

Aliasing

- Moiré patterns
- Disintegration of textures

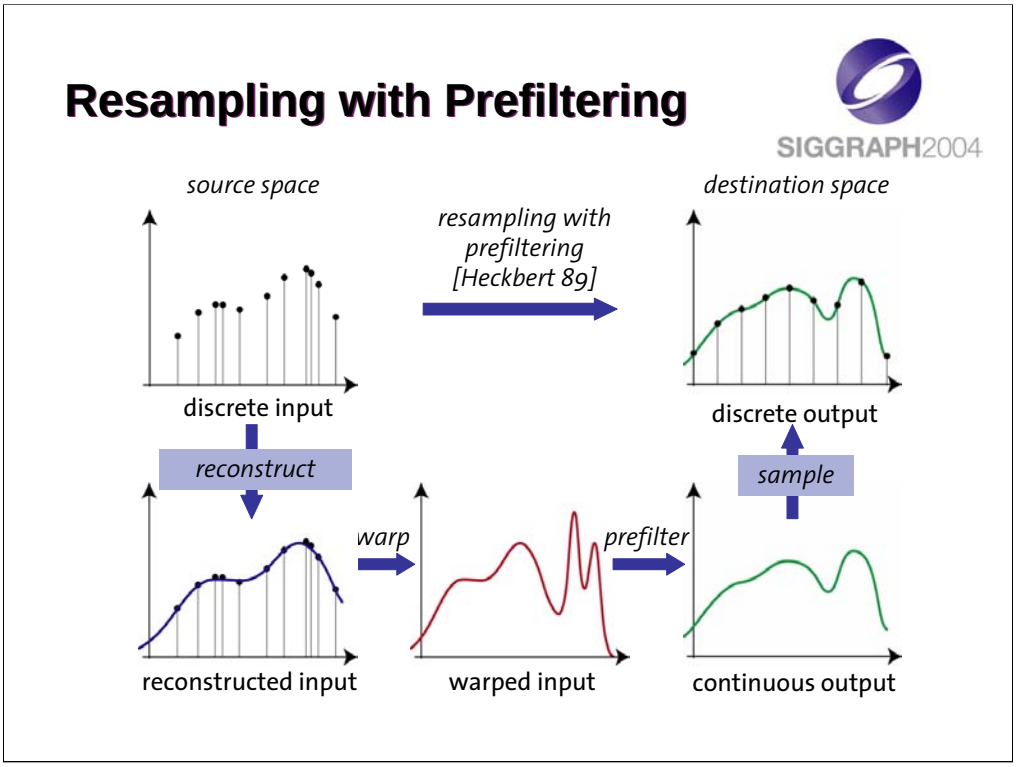


Unfortunately, rendering using warped reconstruction kernels does not prevent so-called aliasing artifacts. Aliasing artifacts include Moire patterns in images with highly structured patterns, such as the checkerboard texture shown on the right. Moire patterns are low-frequency patterns that appear in sampled versions of highly structured, high frequency signals. An other artifact that we observe is the disintegration of textures with very fine detail, such as the text pattern on the right. These artifacts are analogous to the artifacts produced by texture mapped triangle rendering with poor texture filtering.

Antialiasing

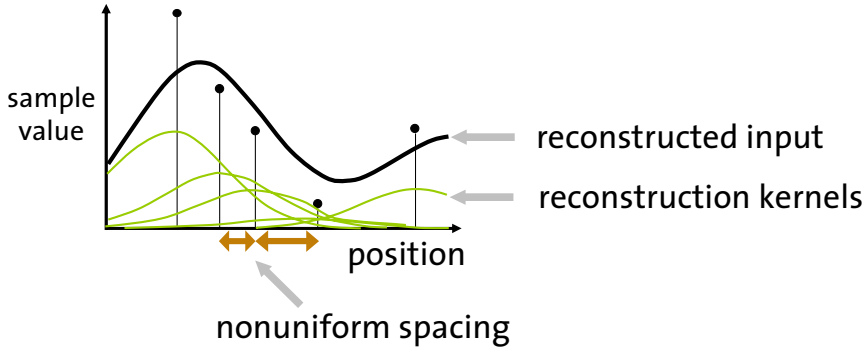
- Aliasing occurs when sampling a signal that is not band-limited to the Nyquist frequency of the sampling grid
- Aliasing is avoided by *prefiltering*, i.e., *band-limiting to the Nyquist frequency*, before sampling

From signal processing theory we know that aliasing occurs when sampling a signal that is not band-limited to the Nyquist frequency of the sampling grid. The best way to prevent aliasing is to prefilter the signal before sampling, i.e., to band-limit it to the Nyquist frequency of the sampling grid. Prefiltering techniques are common for texture filtering, and the technique that we will describe for point splatting is actually derived from a high quality texture filtering approach.

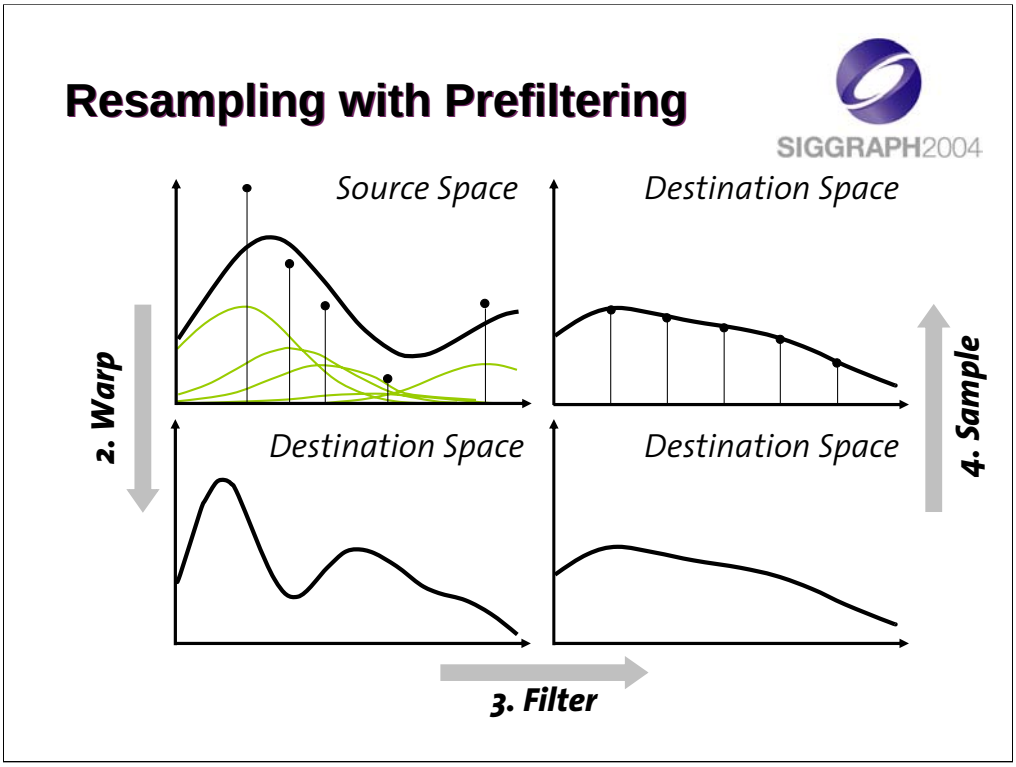


Our approach is based on the concept of resampling with prefiltering, which has been introduced to computer graphics by Paul Heckbert in 1989 in the context of texture filtering and image warping. Resampling is the process of taking a discrete input signal, defined in so-called source space, transform it to a destination space, and sample it to a new sampling grid in destination space. To avoid aliasing artifacts, resampling includes the following steps: First, a continuous signal is reconstructed in source space. The continuous signal is then warped to destination space. Next, a prefilter is applied to remove high frequencies from the warped signal, i.e., to band-limit the signal. The band-limited signal is then sampled at the new sampling grid in destination space, avoiding any aliasing artifacts.

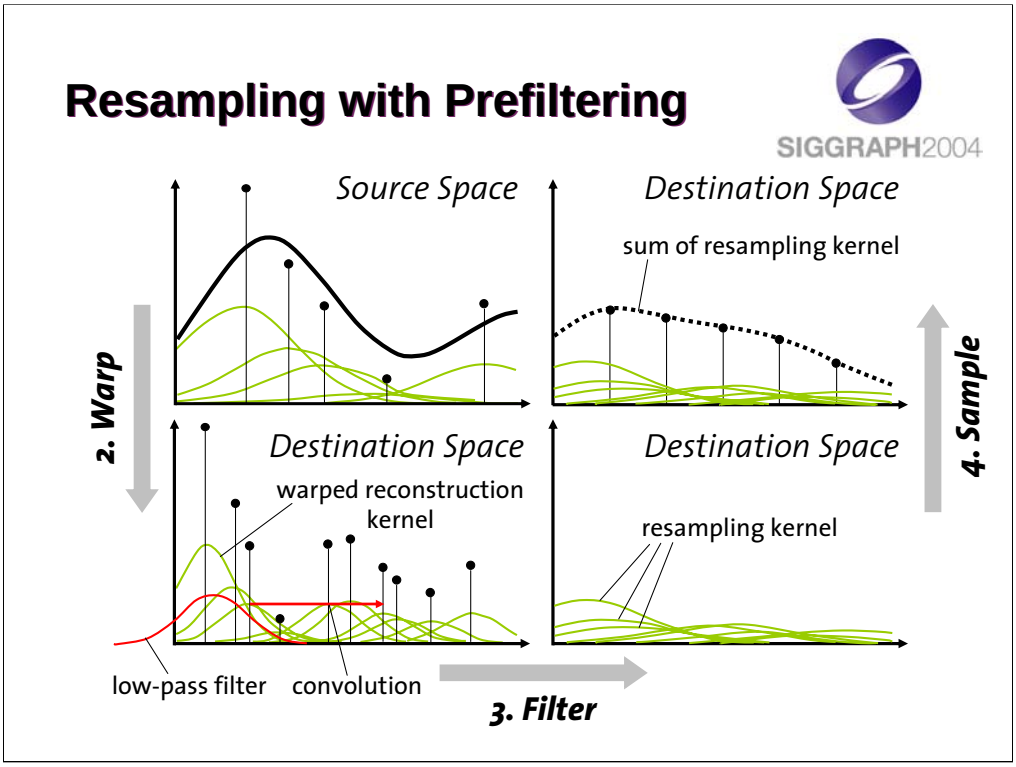
Resampling with Prefiltering



Let me describe this in some more detail in the 1D setting: We first reconstruct a signal by building a weighted sum of reconstruction kernels. Note that the kernels are nonuniformly distributed in space.



Next, the reconstructed signal is warped to destination space as described before. Here, the warp is simply a 1D dilation of the signal. The warped signal is then low-pass filtered and sampled on the output grid in destination space.



Observe that the warped signal in destination space can again be expressed as a weighted sum of warped reconstruction kernels. Band-limiting the warped signal is implemented as a convolution with a low-pass filter. Since convolution is a linear operation, we can convolve each warped reconstruction kernel with the low-pass kernel individually, instead of applying the low-pass filter to the sum of the reconstruction kernels. We call a warped, low-pass filtered reconstruction kernel a resampling kernel, as shown in the bottom right figure. Hence, the continuous output signal is a weighted sum of resampling kernels, which is evaluated at the output sampling grid.

Resampling Filters

- Ignoring normalization for now

$$g(x) = \sum_i f_i r_i(\varphi_i^{-1}(x)) \otimes h(x)$$

resampling kernel

- The image space resampling kernel combines a warped reconstruction kernel and a low-pass kernel

It is straightforward to apply this resampling framework to the reconstruction scheme described earlier in the talk. Let me ignore the normalization division for now. The output signal is given by convolving the reconstructed signal in image space with the low-pass filter. Because of the linearity of the convolution, each warped reconstruction kernel can be convolved separately, yielding a resampling kernel. So the resampling kernel essentially is a combination of a warped reconstruction kernel and a low-pass kernel.

Gaussian Resampling Filters

- Proposed by [Heckbert 89]

$$g_{\mathbf{V}}(\mathbf{x}) = \frac{1}{\sqrt{2\pi}\left|\mathbf{V}\right|^{1/2}} e^{-\frac{1}{2}\mathbf{x}^T\mathbf{V}^{-1}\mathbf{x}}$$

- Gaussians are closed under
  - linear mappings: $g_{\mathbf{V}}(\mathbf{W}^{-1}\mathbf{x}) = \left|\mathbf{W}\right| g_{\mathbf{WVW}^T}(\mathbf{x})$
  - convolutions: $g_{\mathbf{V}}(\mathbf{x}) \otimes g_{\mathbf{V}'}(\mathbf{x}) = g_{\mathbf{V}+\mathbf{V}'}(\mathbf{x})$

While resampling seems to be an elegant approach to do reconstruction and band-limiting in a unified framework, it is not clear yet how to compute these kernels efficiently. To use resampling filters in practice, Heckbert proposed to use Gaussians as reconstruction and as low-pass filter kernels. The familiar equation for a 2D Gaussian g with a 2x2 variance matrix V is given here. To compute resampling kernels efficiently, we exploit two interesting properties of Gaussian kernels: They are closed under linear mappings and under convolutions. Applying a linear mapping W to a Gaussian with variance matrix V yields an other Gaussian with variance matrix WVW^T. Similarly, convolving a Gaussian with variance matrix V with an other one with variance matrix V' leads to a new one with variance matrix V+V'.

Gaussian Resampling Filters

- Locally approximate the projective mapping using its Jacobian
  - Exact: $\mathbf{x} = \varphi(\mathbf{u}) = \left[x_0(u_0,u_1), x_1(u_0,u_1)\right]$
  - Approximation: $\mathbf{x} \approx \tilde{\varphi}(\mathbf{u}) = \mathbf{x}_0 + \mathbf{J_0} \cdot \mathbf{u}$

where $\mathbf{J}_0 = \left[\begin{array}{cc} \frac{\partial x_0(u_0,u_1)}{\partial u_0} & \frac{\partial x_0(u_0,u_1)}{\partial u_1} \\ \frac{\partial x_1(u_0,u_1)}{\partial u_0} & \frac{\partial x_1(u_0,u_1)}{\partial u_1} \end{array}\right]_{(0,0)}$

Note that since the projective mapping phi from local tangent coordinates to image space is not linear, we can not use it directly to warp the Gaussian reconstruction kernels to image space. Instead, we use a locally linear approximation of this mapping by computing its linear Taylor expansion. This involves the Jacobian J_0, which is the 2x2 matrix consisting of its partial derivatives evaluated at u=0.

Gaussian Resampling Filters

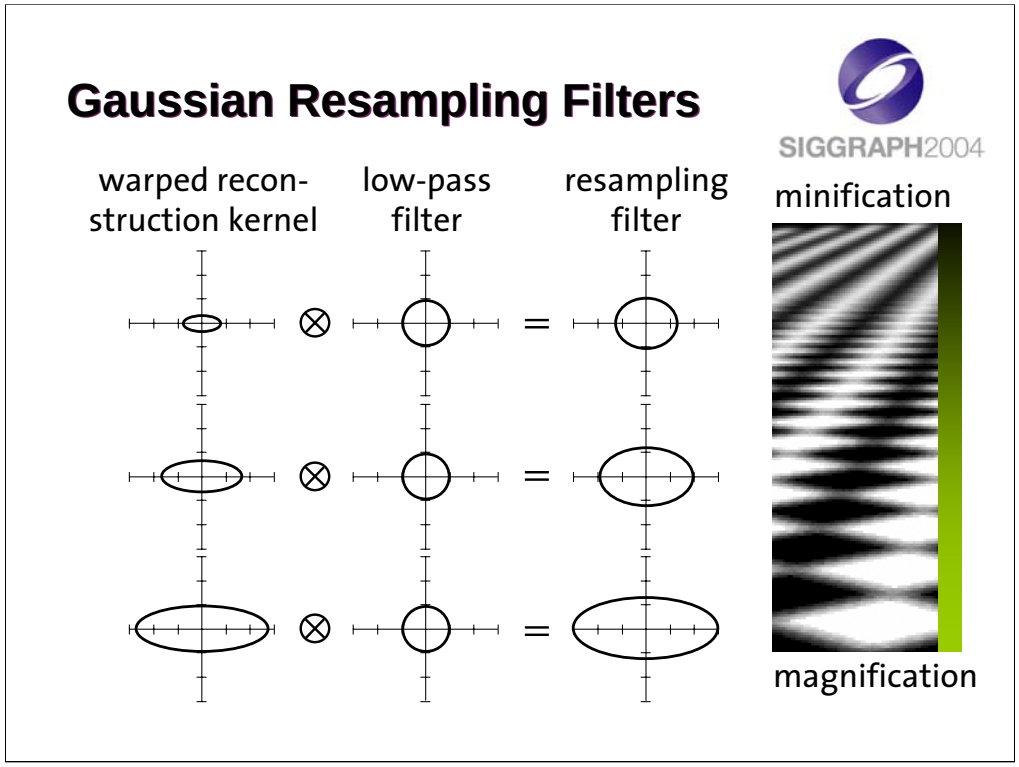
- Using the affine approximation and the exploiting the properties described before

$$g(x) = \sum_i f_i r_i(\tilde{\varphi}_i^{-1}(x)) \otimes h(x)$$

$$= \sum_i f_i \frac{1}{\left|\mathbf{J}_i^{-1}\right|} g_{\mathbf{J}_i \mathbf{V}_i \mathbf{J}_i^T + \mathbf{H}}(\mathbf{x})$$

Gaussian resampling kernel
(**EWA resampling kernel**)

Using the affine approximation and exploiting the properties of Gaussians described before, we can now express the warped and band-limited reconstruction kernel as a single Gaussian kernel. Heckbert has also called this kernel the EWA resampling kernel, EWA standing for elliptical weighted average. The variance matrix of this kernel is composed of the original variance V_i matrix of the reconstruction kernel in tangent space, the Jacobian J_i of the mapping from tangent to image space, and the variance matrix of the low-pass kernel H, which is usually a 2x2 identity matrix. The Gaussian resampling kernel can be computed efficiently, since it only involves one 2x2 matrix inversion and a couple of 2x2 matrix multiplications and additions.



The effects of Gaussian resampling kernels on the splat shape can be visualized and explained very intuitively. In the left column of this table I depict the shapes of the warped reconstruction kernels in image space. Note that the ellipse corresponds to a contour line of the Gaussian kernel. The shape and size of the reconstruction kernel depends on the projection of the local tangent plane. In case of minification, shown in the top left, the reconstruction kernel will be “smaller” than the pixel spacing, which is indicated on the axes of the figures. In other words, the reconstruction kernel contains higher frequencies than can be captured by the pixel grid, which leads to aliasing artifacts. Convolution with the low pass filter, shown in the middle column, solves this problem. Note that the low-pass filter is independent of the projection, but its size is determined by the spacing of the pixel grid. The resulting resampling kernels are depicted in the right column. Observe that in the case of minification, shown in the top row, the shape of the resampling kernel is largely determined by the low-pass kernel, blurrying the reconstruction kernel and removing its high frequencies. In the middle column, a case of anisotropic minification and magnification is shown. This leads to an anisotropic scaling of the reconstruction kernel as shown on the right. In other words, this filtering technique is equivalent to a so-called anisotropic texture filter. The bottom row illustrates magnification of the reconstruction kernel. In this case, convolution with the low-pass kernel does not change the shape of the kernel significantly, avoiding unnecessary blurrying the output image.

Image Quality Comparison

- EWA texture filtering [Heckbert 89]

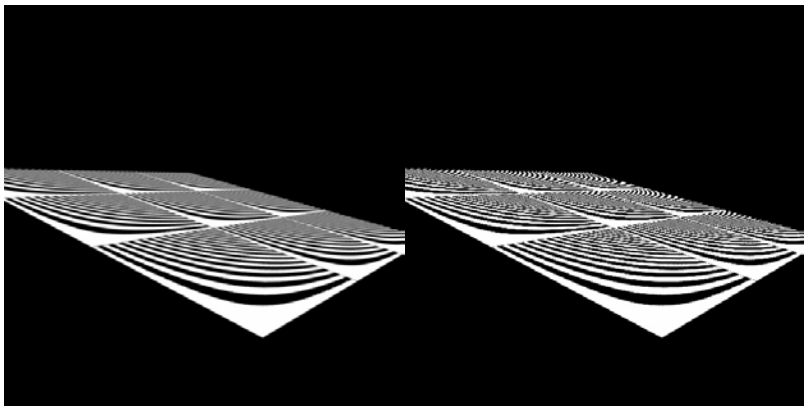
surface splatting

EWA texture filtering
equivalent quality

In fact, splatting with Gaussian resampling kernels is mathematically equivalent to EWA texture filtering as originally proposed by Heckbert in 1989. As a result, both approaches provide the same high image quality due to true anisotropic texture filtering.

Image Quality Comparison

- Ellipse splatting (no low-pass filter)



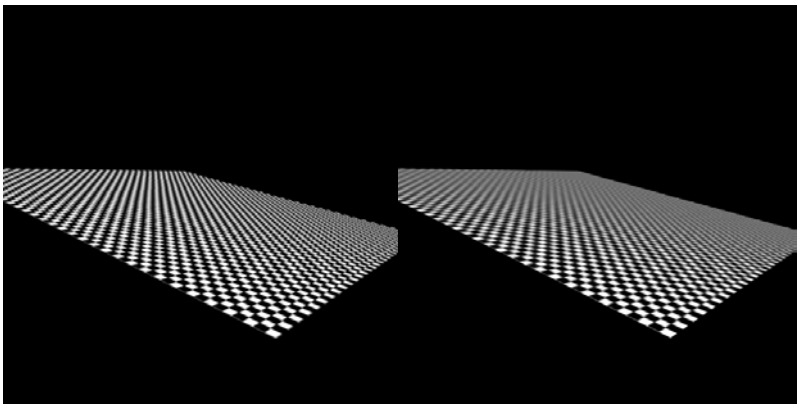
surface splatting

splatting reconstruction kernels, *aliasing*!

Here I compare splatting resampling kernels on the left with splatting reconstruction kernels without low-pass filtering on the right. Clearly, this leads to strong aliasing artifacts, visible as Moire patterns on the right. In contrast, those artifacts are completely avoided when using prefiltered resampling kernels.

Image Quality Comparison

- Trilinear mipmapping



surface splatting
anisotropic filtering

trilinear mipmapping
isotropic filtering

This is one more comparison of surface splatting on the left with trilinear mipmapping on the right, a popular texture filtering technique. While surface splatting provides anisotropic filtering, trilinear mipmapping is an isotropic filter. However, the amount of filtering required in this example is different in the horizontal and vertical direction. Trilinear mipmapping filters conservatively in both directions, leading to excessive blurriness in the horizontal direction in this example. While the structure of the checkerboard texture is preserved along the horizontal direction on the left, it is completely lost on the right side due to isotropic filtering.

Implementation

- Software implementation „surface splatting“ [Zwicker et al. 2001]
  - Edge antialiasing
  - Order independent transparency
- LDC tree data structure [Pfister et al. 2000]
  - Extension of layered depth images
  - Hierarchical rendering
  - Optimized warping

The following slides show some results that were generated using a software implementation of our technique. This implementation also features edge antialiasing and order independent transparency to render semi-transparent surface. It is based on the LDC tree data structure, which is an extension of layered depth images. This data structure allows for hierarchical rendering and optimized warping.

Results

- Scanned head (429‘075 points)

This first example is a data set that was acquired using a Cyberware laser range scanner. The scanner obtained roughly 400’000 surface samples, which are visualized directly using our splatting approach.

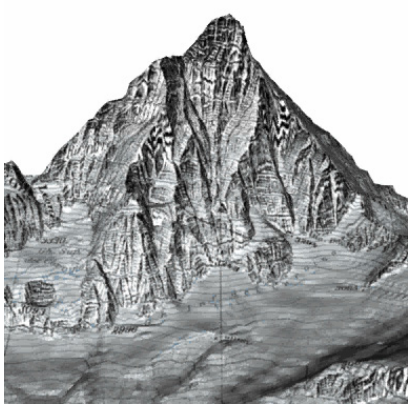
Results

- Helicopter (987‘552 points)

Here we show a complex object that was sampled from a synthetic model. It contains almost one million points. This animation also illustrates rendering of semi-transparent surfaces.

Results

- Matterhorn (4‘782‘011 points)



This is a data set that we acquired from a textured digital terrain model. It contains almost five million points. This animation is to demonstrate the high image quality of our splatting approach. The texture contains very fine detail, which is rendered without flickering or any other aliasing artifacts.

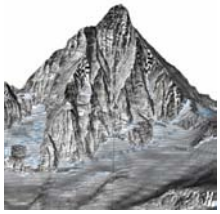
Performance

- Software implementation [Zwicker et al. 2001]
  - 1.1 GHz AMD Athlon, 1.5 Gbyte RAM

Data	#Points	256 x 256	512 x512
Scanned head	429‘075	1.3 fps	0.7 fps
Helicopter	987‘552	0.6 fps	0.3 fps
Matterhorn	4‘782‘011	0.2 fps	0.1 fps

Scanned head

Helicopter



Matterhorn

Here is some performance data of the software implementation that was measured on a 1.1 GHz AMD Athlon system. With current systems, rendering performance is increased about a factor of two. We achieved an average of roughly 500’00 splats per second for the models shown before. Note that the performance also depends significantly on the output image resolution. Clearly, this performance is not sufficient for interactive rendering of complex objects. Therefore, we have also attempted to accelerate surface splatting by exploiting current programmable computer graphics hardware, which I will describe in the final part of this talk.

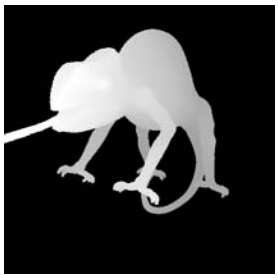
Hardware Implementation

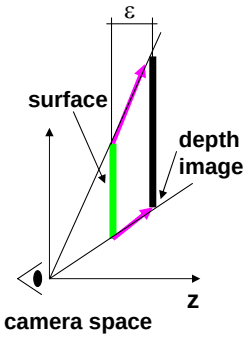
- Challenges
  - Extended z-buffering with ε-threshold
  - Per pixel normalization
- Hardware implementation requires a *three pass algorithm* on current GPUs

An implementation of our splatting algorithm on current GPUs faces two challenges. First, the extended z-buffering scheme with an epsilon threshold is not supported directly in current hardware. Second, the per pixel normalization must be done after all splats have been processed and hence it requires an additional pass over the framebuffer. So all implementations on current hardware require three rendering passes, which I will describe in some more detail in the following.

First Pass

- Draw depth image with a small depth offset ε away from the viewpoint
- Perform regular z-buffering (depth tests and updates), no color updates





In the first pass, we draw a depth image into the framebuffer, which is offset away from the viewpoint by a small depth offset epsilon. We apply regular z-buffering with depth tests and updates, but no color writes. The depth image is visualized on the left. On the right, we show how the depth offset is applied by shifting rendering primitives along viewing rays away from the viewer.

Second Pass

- Draw colored splats with additive blending enabled
- Perform depth tests, but no updates
- Accumulate
  - Weighted colors of visible splats in the color channels
  - Weights of visible footprint functions in the alpha channel

In the second rendering pass, we draw colored with additive alpha blending enabled. In this pass, we perform depth tests, using the depth image computed in the first pass, but no updates. Hence, all splats that lie behind the depth image are discarded. This pass accumulates the weighted colors of the visible splats in the RGB color channels, and the weights of the visible footprint functions in the alpha channel of the framebuffer.

Third Pass

- Normalization of the color channels by dividing through the alpha channel
- Implemented by
  - using the framebuffer as a texture
  - drawing a screen filling quad with this texture
  - performing the normalization in a fragment program

The third pass computes the normalization by the accumulated footprint weights by dividing the color channels through the alpha channel. This is implemented by using the framebuffer as a texture. We draw a screen filling quad with this texture, such that there is a one-to-one correspondence between pixels in the framebuffer and texels. Normalization is then performed in a simple fragment program.

Drawing Splats

- Based OpenGL points (`GL_POINT`)
- Vertex program computes *conic matrix* for the exponent of the Gaussian

$$\mathbf{C} = \left(\mathbf{J}_i \mathbf{V}_i \mathbf{J}_i^T + \mathbf{H}\right)^{-1}$$

  and a *bounding box* that contains the ellipse

$$\mathbf{x}^T \mathbf{C} \mathbf{x} = r^2$$

  where *r* is the cutoff radius
- The conic matrix **C** is passed as a texture coordinate to the fragment stage
- The bounding box is used to set the OpenGL point size

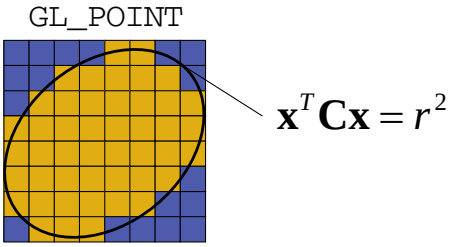
Independent of the three pass rendering approach described before, there are several ways how individual splats can be drawn into the framebuffer. Here we present a method that is based on OpenGL point primitives, which avoids representing each point as a quad or triangle as has been proposed before. Our approach relies heavily on vertex and fragment programs, but it is a quite straightforward implementation of the equations for the Gaussian resampling kernel derived before. In the vertex program, we compute the 2x2 conic matrix for each splat, which is the matrix that appears in the exponent of the Gaussian. This requires the computation of the Jacobian J, the addition of the low-pass kernel H, and the inversion of the resulting matrix. We also compute the size of a bounding box that encloses the ellipse defined by the conic matrix with a certain radius r^2. Here, r is a cutoff radius that clamps the Gaussian kernel to a finite support. The conic matrix C is then passed as a texture coordinate to the fragment stage. Further, the size of the bounding box is used to set the appropriate OpenGL point size.

Drawing Splats

- The fragment program evaluates

$$d^2 = \mathbf{x}^T \mathbf{C} \mathbf{x}$$

  where **x** are pixel coordinates accessible through the `wpos` input argument (Cg, NVidia)
- Discard fragments if *d^2 > r^2*

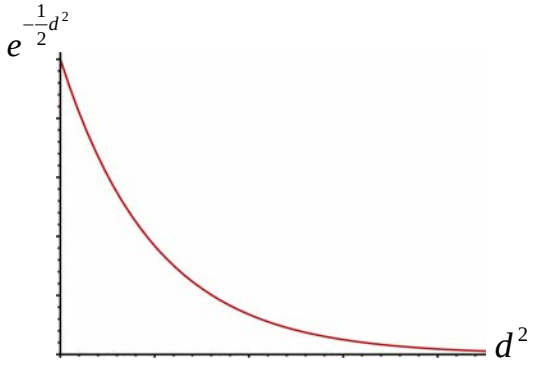


The fragment program visits each pixel covered by the point primitive and evaluates the quadratic form (x^T)Cx, where x are pixel coordinates that are accessible through the wpos input argument. Note that this currently works only with NVidia cards. We then discard all pixels that yield a value d^2 > r^2, which results in rasterizing an ellipse. Note that by choosing the point size correctly, we guarantee that the ellipse always lies completely within the point primitive.

Drawing Splats

- Use *d^2* to look up the value of the Gaussian in a 1D texture, i.e.,

$$tex(d^2) = e^{-\frac{1}{2}d^2}$$



Finally, to evaluate the Gaussian kernel at each pixel, we use d^2 to look up the value of the Gaussian in a 1D texture. Note that the texture is indexed directly by d^2 instead of d, avoiding the computation of the square root.

Performance

- NVidia GeForce FX (NV35) [Zwicker et al. 2004]

Image Resolution	Mio. Splats/Sec.
512x512	3.1
1024x1024	2.8

Chameleon Salamander Wasp Fiesta

In an implementation on an NVidia GeForce FX, we obtained a rendering performance of about 3 million splats per second. Performance is still dependent on the output image resolution, meaning that the renderer is fragment processing limited for higher output resolutions.

Performance

- Low-quality hardware accelerated point splatting
  - up to 50 million points per second on GeForce4 or Radeon 9700 [Dachsbacher et al. 2003]

Note that it has been demonstrated that with low-quality hardware accelerated point splatting much higher performance can be obtained. With suitable data structures, Dachsbacher et al. achieved a throughput of up to 50 million points per second on GeForce4 or Radeon 9700 GPUs. These approaches use a single rendering pass and avoid computationally expensive vertex and fragment programs, trading-off performance for high filtering quality provided by surface splatting.

Variations of Surface Splatting

- Alternative approximation of perspective projection [Räsänen 2002, Zwicker 2004]
- Approximation using look-up tables [Botsch et al. 2002]
- Efficient hardware implementation [Botsch et al. 2003]
- Non-Gaussian kernels [Pajarola et al. 2003]

For further reading, let me point out several interesting variations of the surface splatting algorithm that have been presented recently. Rasanen and Zwicker introduced an alternative approximation of the perspective projection that is not based on the Jacobian. Rather, it is based on the exact projective mapping of conic sections using homogeneous coordinates, yielding more accurate splat shapes than the approximation based on the Jacobian. Botsch et al. presented an approximation of surface splatting using pre-computed look-up tables, avoiding the on-the-fly computation of the splat kernels. They also described an efficient hardware implementation that also uses the approach based on OpenGL points described before. Finally, Pajarola et al. proposed to use non-Gaussian kernels to improve filtering quality.

Applications of Point Rendering

- Direct visualization of point clouds from 3D scanners [Rusinkiewicz et al. 2000, Matusik et al. 2002, Xu et al. 2004]
- Real-time 3D reconstruction and rendering for virtual reality applications [Gross et al. 2003]

[Rusinkiewicz et al. 2000]

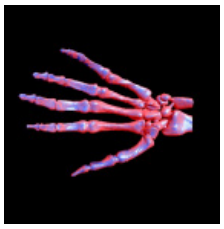
[Matusik et al. 2002]

[Gross et al. 2003]

Point rendering has found a variety of applications. Several researchers have proposed to use point clouds for direct visualization of data acquired by 3D scanners [Rusinkiewicz et al. 2000, Matusik et al. 2002, Xu et al. 2004]. Related to this is using points as a surface representation and rendering primitive in a real time acquisition and rendering environment for virtual reality [Gross et al. 2003].

Applications of Point Rendering

- Hybrid point and polygon rendering systems [Chen et al. 2001, Cohen et al. 2001]
- Rendering animated scenes [Wand et al. 2002]



[Chen et al. 2001]

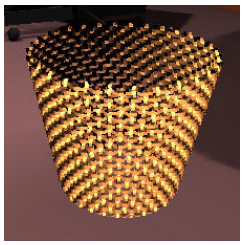
[Wand et al. 2002]

Various systems have been proposed to combine traditional rendering primitives with points to get the best of both worlds [Chen et al. 2001, Cohen et al. 2001]. Points have also been proposed to render large animated scenes[Wand et al. 2002].

Applications of Point Rendering

- Interactive display of huge meshes [Wand et al. 2001, Deussen et al. 2002]
- On the fly sampling and rendering of procedural objects [Stamminger et al. 2001]
- Modeling and editing [Zwicker et al. 2002, Pauly et al. 2003]


[Deussen et al. 2002]


[Stamminger et al. 2001]


[Pauly et al. 2003]

Points are suitable to build up efficient hierarchies for huge scenes with millions of rendering primitives, which has been explored to interactively display ecosystems with plants and trees [Deussen et al. 2002] or other massive outdoor scenes [Wand et al. 2001]. Finally, points are investigated by several researchers as a primitive for modeling and editing [Stamminger et al. 2001, Zwicker et al. 2002, Pauly et al. 2003].

Conclusions

- Point rendering as a resampling process
- High quality point splatting with antialiasing using a prefiltering approach
- Performance
  - 3 million high quality splats per second with hardware support [Zwicker et al. 2004]
  - 1 million points per second in software
- Various applications for point splatting have been explored

In this talk, I have presented point rendering as a resampling process. Resampling leads to high quality point splatting. It avoids aliasing artifacts using a prefiltering approach and yields image quality equivalent anisotropic texture filtering. The splatting algorithm can be implemented in software or using programmable vertex and fragment stages of modern graphics processors. We have obtained a rendering performance of three million splats per second with hardware support, and roughly one million splats in software. Point rendering has been adopted in various applications, because mesh reconstruction is not necessary and level-of-detail and hierarchy construction is very simple even for highly complex geometries.

Future Work

- Rendering hardware
  - Dedicated point rendering hardware
  - Extension of current hardware
- Efficient approximations of exact EWA splatting
  - Higher performance without loss of image quality
- Unifying rendering primitive for all sorts of surface representations
  - Rendering architecture for on the fly sampling and rendering

In the future, we would like to think about dedicated rendering hardware for points, or how to extend existing hardware to more efficiently support points. We would also like to investigate efficient approximations of exact EWA splatting that do not impact image quality, but provide higher rendering performance. Finally, points could be used as a unifying rendering primitive for all sorts of surface representations in a rendering architecture that performs on the fly sampling and rendering.

Acknowledgments

- Markus Gross, Mark Pauly, CGL at ETH Zurich
- Hanspeter Pfister, Jeroen van Baar, MERL Cambridge
- Liu Ren, Carnegie Mellon University
- Fredo Durand, MIT Graphics Group

References

- [Levoy and Whitted 1985] *The use of points as a display primitive*, technical report, University of North Carolina at Chapel Hill, 1985
- [Heckbert 1986] *Fundamentals of texture mapping and image warping*, Master‘s Thesis, 1986
- [Grossman and Dally 1998] *Point sample rendering*, Eurographics workshop on rendering, 1998
- [Levoy et al. 2000] *The digital Michelangelo project*, SIGGRAPH 2000
- [Rusinkiewicz et al. 2000] Qsplat, SIGGRAPH 2000
- [Pfister et al. 2000] *Surfels: Surface elements as rendering primitives*, SIGGRAPH 2000
- [Zwicker et al. 2001] *Surface splatting*, SIGGRAPH 2001
- [Chen et al. 2001] *POP: A Hybrid Point and Polygon Rendering System for Large Data*, IEEE Visualization 2001
- [Cohen et al. 2001] *Hybrid simplification: combining multi-resolution polygon and point rendering*, IEEE Visualization 2001
- [Wand et al. 2001] *The Randomized z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes,* SIGGRAPH 2001

References

- [Stamminger et al. 2001] Interactive sampling and rendering for complex and procedural geometry, Rendering Techniques 2001
- [Deussen et al. 2002] *Interactive Visualization of Complex Plant Ecosystems.* IEEE Visualization 2002
- [Wand et al. 2002] *Multi-Resolution Rendering of Complex Animated Scenes.* Eurographics 2002
- [Botsch et al. 2002] *Efficient high quality rendering of point sampled geometry*, Eurographics Workshop on Rendering 2002
- [Zwicker et al. 2002] *EWA Splatting*, IEEE TVCG 2002
- [Ren et al. 2002] *Object space EWA splatting: A hardware accelerated approach to high quality point rendering*, Eurographics 2002
- [Räsänen 2002] *Surface Splatting: Theory, Extensions and Implementation*, MSc thesis 2002
- [Matusik et al. 2002] *Image-based 3D photography using opacity hulls.* SIGGRAPH 2002, July 2002.

References

- [Zwicker et al. 2002] Pointshop 3D: An interactive system for editing point-sampled surfaces. SIGGRAPH 2002
- [Pauly et al. 2003] Shape modeling with point-sampled geometry. Siggraph 2003
- [Pajarola et al. 2003] *Object-Space Point Blending and Splatting*. In Sketches & Applications Catalog SIGGRAPH 2003
- [Botsch et al. 2003] *High-Quality Point-Based Rendering on Modern GPUs.* Pacific Graphics 2003
- [Gross et al. 2003] *blue-c: A Spatially Immersive Display and 3D Video Portal for Telepresence.* SIGGRAPH 2003
- [Dachsbacher et al. 2003] *Sequential point trees*. SIGGRAPH 2003
- [Zwicker et al. 2004] *Perspective accurate splatting*. Graphics Interface 2004
- [Xu et al. 2004] *Stylized rendering of 3D scanned realworld environments*. NPAR 2004

SIGGRAPH2004

Efficient Data Structures

Marc Stamminger, University of Erlangen-Nuremberg

Introduction

- point rendering
  - how to adapt point densities ?
    - *for a given viewing position, how can we get n points that suffice for that viewer ?*
  - how to render the points ?
    - *given n points, how can we render an image from them ?*

When using points for efficient rendering, we have to consider two issues.

First, how to adapt the point densities for a given camera to avoid holes but not to waste resources.

Second, how to render the selected points. We can use software or hardware based approaches for point rendering or splatting. When using hardware acceleration, it is very important to restrict data transfer between main memory and the GPU to a minimum.

Introduction

- how to render the points ?
  - project point to pixel, set pixel color
  - hardware acceleration
    - >80 mio. points per second (Radeon 9700)
    - very fast, but bad support for splatting

The simplest way of rendering the points is to project them to the screen and set the corresponding pixel.

Graphics hardware can do this using the point primitive very efficiently, at least as long as we don‘t want to have sophisticated filtering and splatting techniques. Current graphics hardware can render almost 100 million points per second.

Introduction

- `for ( i = 0; i < N; i++ ) renderPointWithNormalAndColor (x[i],y[i],z[i],nx[i],ny[i],nz[i],…);`
  → ~10 mio points per second
- `for ( i = 0; i < N; i++ ) renderPoint(x[i],y[i],z[i]);`
  → ~20 mio points per second
- `float *p = {...} renderPoints(p,N);`
  → >80 mio points per second

immediate mode

→ *best performance with sequential processing of large chunks !*

In this talk, we want to use hardware rendering for the points. This is by far the fastest way to render points, and it can be foreseen that future hardware will support high-quality splatting techniques better.

For hardware rendering, it is vital to think about the geometry submission to the GPU.

When using e.g. OpenGLs immediate mode, we have a tremendous overhead of functions calls and also memory bandwith problems. The more attributes we associate with a single point sample, the worse it gets.

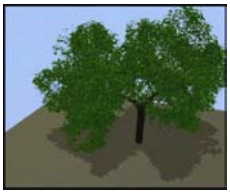
When rendering point primitives with a normal and color value, we can achieve a point rate of approximately 10 million splats per second. Using coordinates only, we achieve about 20 million points per second, so memory bandwith is the bottleneck.

The shown numbers are not really measured, they only give an idea of the magnitude.

The most performant rendering is achieved by processing large chunks of data at once, in particular, if this data is already stored in video memory.

Intro

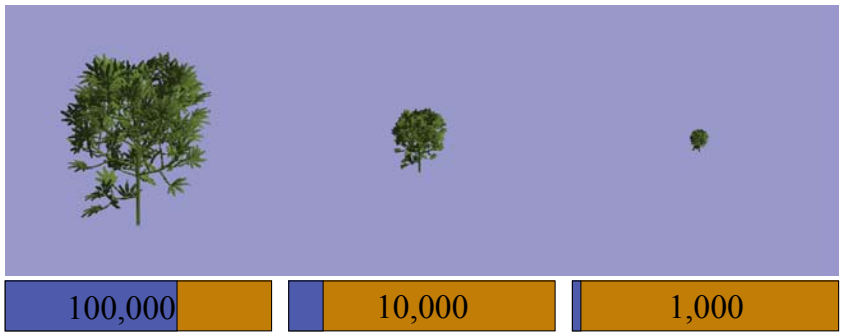
- in this talk: two approaches
  - eco system rendering with points, lines, and triangles



  - sequential point trees

In this talk, we will present two techniques for point rendering on graphics hardware. The first one is optimal for objects like plants, with complex, unconnected geometry. The second one is better suited for complex, but connected geometry.

Eco System Rendering

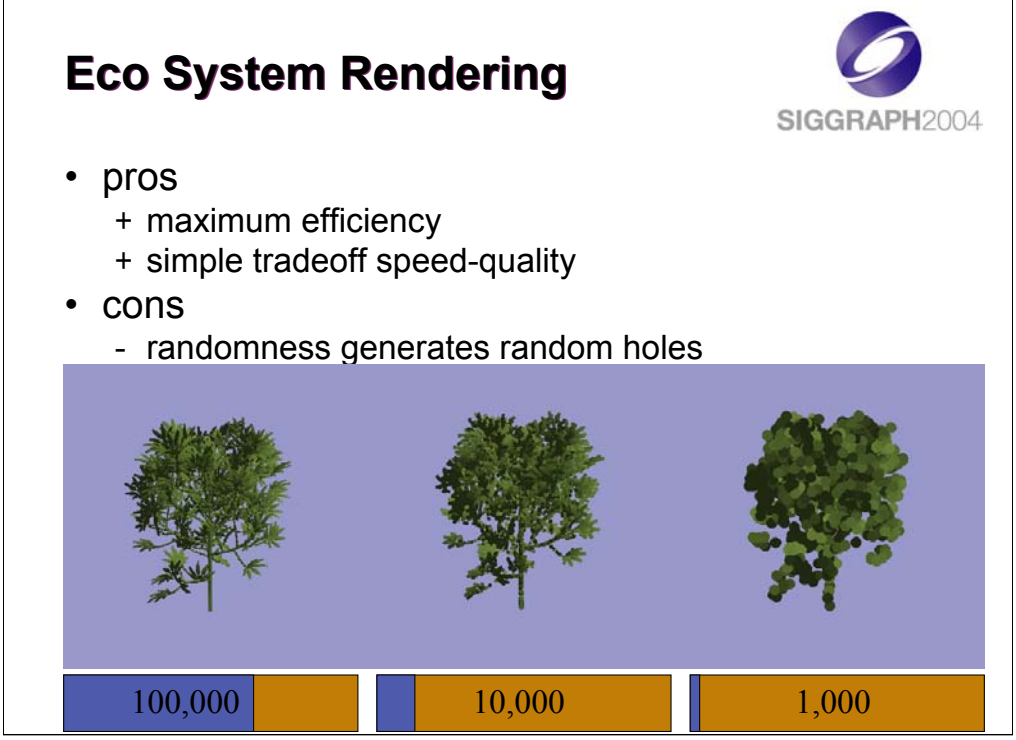


- point list
  - array of random points
- for every frame
  - compute *n,* render first *n*

Let‘s start with the rendering of complex objects such as plants. Our idea is simple: we precompute a list of random points on the object and store them together with their normal and color in an array. Because the points are randomly chosen, we can get *n* random points on the object by simply chosing the first *n* points.

So, for every frame we compute an appropriate number of points n and render the first n points from the array. According to the observations before, this rendering of a continuous segment of the array can be done at maximum speed by the GPU.

Eco System Rendering



By increasing the point size, less points are required and speed can be easily balanced versus quality.

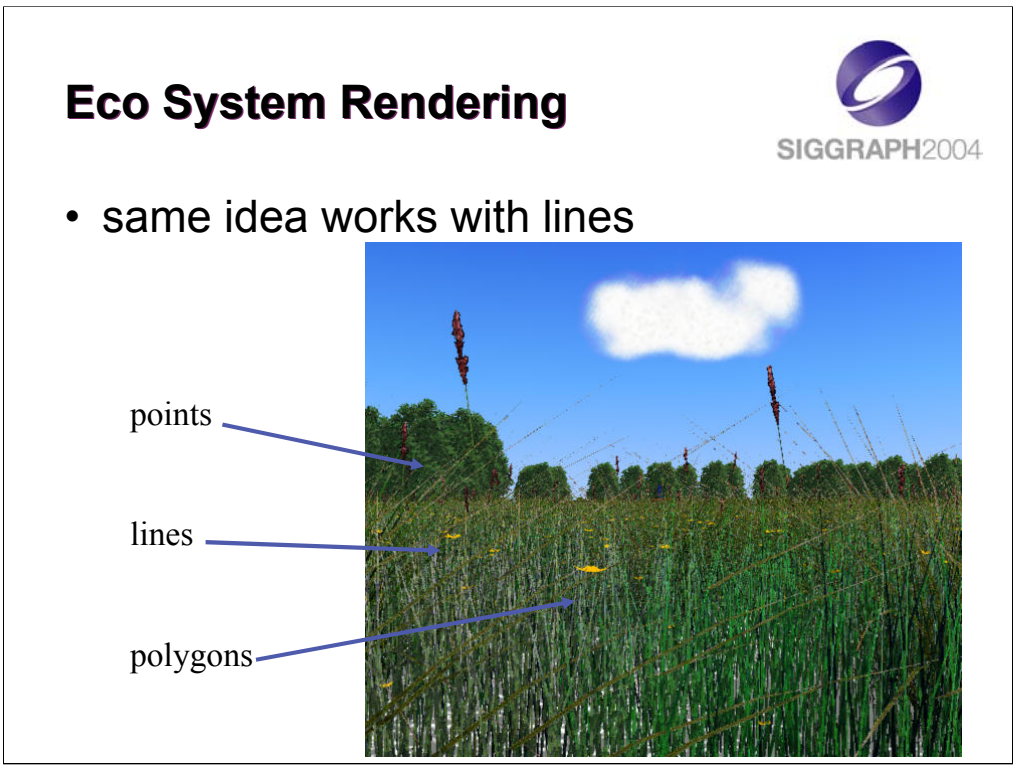
The disadvantage is that the randomness generates holes. Avoiding these holes requires to increase the number of points enormously, which results in a breakdown of speed.

Eco System Rendering

- ideal for plants
  - very high complexity
  - artifacts from holes acceptable
  - other LOD-methods fail for complex topology
- bad for smooth surfaces
  - holes are unacceptable
  - triangles more appropriate
  - better LODs available

However, for plants with many small surfaces (leaves), holes are not noticable and can be tolerated. Interestingly, such unconnected topologies like trees are particularily difficult for most other level of detail methods. For smooth, large, closed surfaces, this approach is hardly useful and other techniques are better suited.

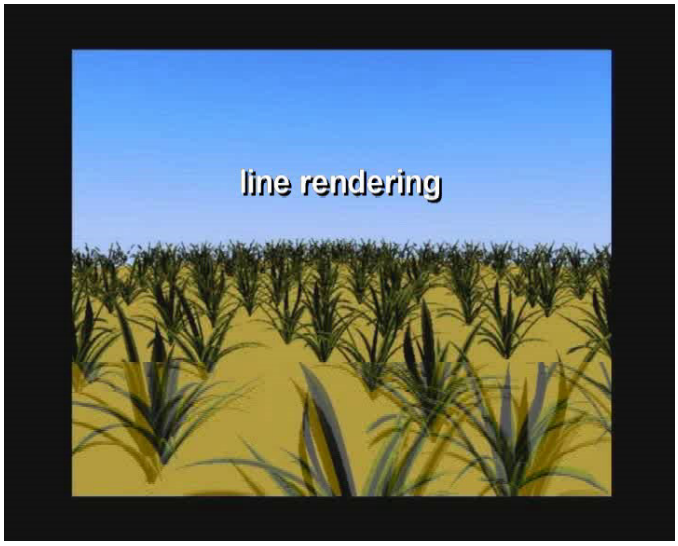
Eco System Rendering



A similar idea works with lines. We can replace long, thin geometry in the distance by lines, e.g. gras blades.

Eco System Rendering

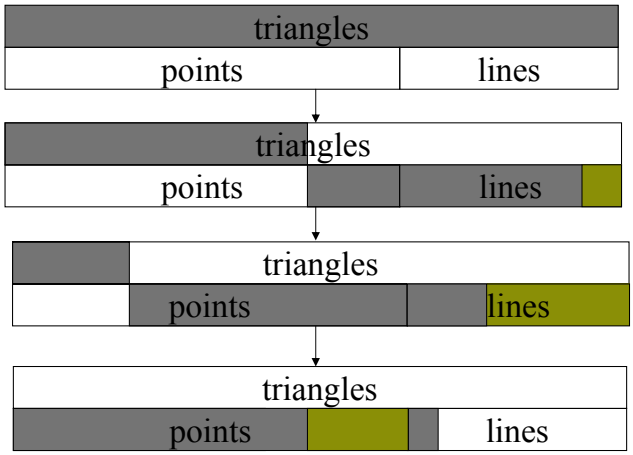
- line rendering



Eco System Rendering

SIGGRAPH2004

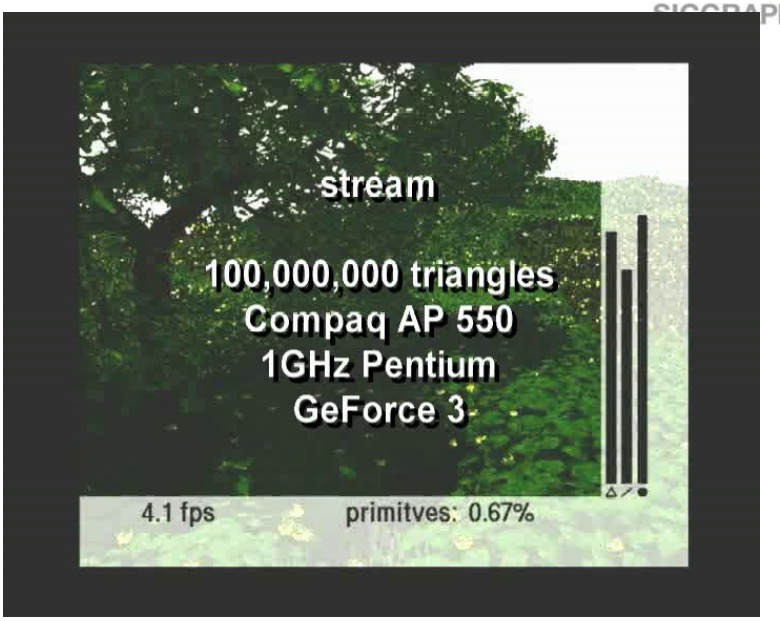
- improved level-of-detail



We start our approach with a triangle model. Part of the triangles can be replaced by lines, the rest by a set of points. These sets of lines and points are precomputed or generated during the evaluation of the procedural plant model.

When we are close to the object, the full triangle model is rendered. When the object is moved away, at some point the triangles that can be represented by the lines are replaced by the line set. Furthermore, we start replacing more and more triangle by their corresponding points. By this, we avoid that the triangles are immediately replaced by points and we thus avoid the popping artifacts. When all triangles are replaced by points, we can reduce the point number for distant objects.

Eco System Rendering



Sequential Point Trees

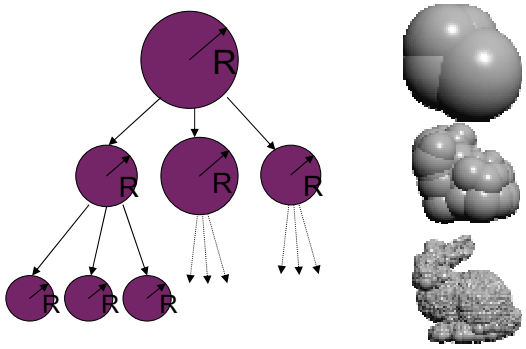
- sequential point trees
  - adaptive point selection *and*
  - sequential processing
- rendering
  - pre-computed point and attribute list
  - render continuous segments only

The second technique presented in this talks are sequential point trees. SPTs are a data structure that allows us to render levels of detail also of closed, smooth surfaces without holes. SPTs can also be processed almost completely on the GPU with very high point throughput.

As before, we precompute an array of points on the object and only process a continuous part of this list.

Hierarchical Processing

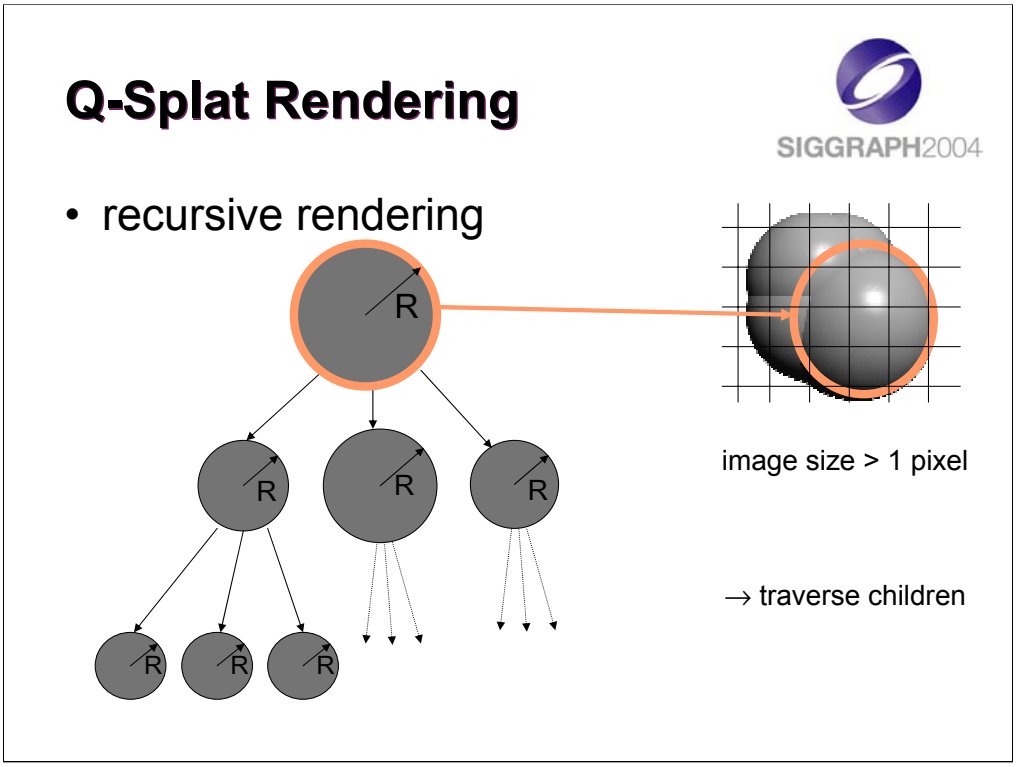
- Q-Splat
  - Rusinkiewicz et al., Siggraph 2000
  - hierarchical point rendering based on bounding sphere hierarchy



© Rusinkiewicz

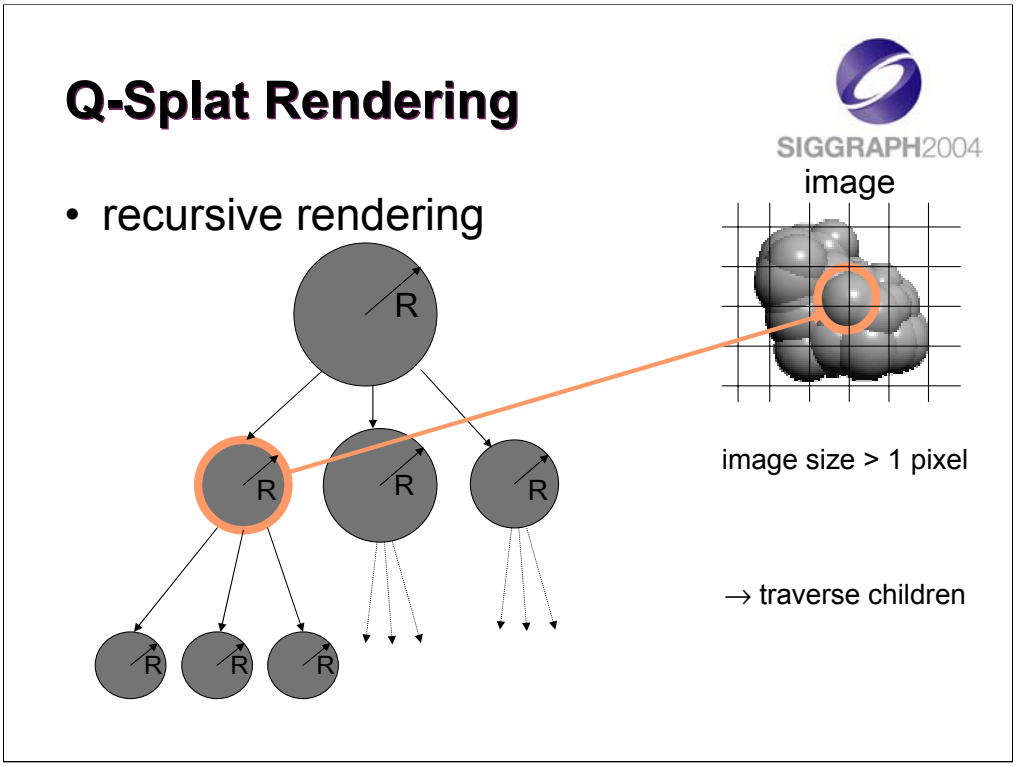
Our method is based on the Q-Splat as introduced by Rusinkiewicz et al at Siggraph 2000.

The Q-Splat does hierarchical point rendering using a bounding sphere hierarchy. Every node represents a part of the objects surface with a position, a radius and a normal. During rendering this hierarchy is traversed by the CPU and the tree data structure is used for frustum culling, backface culling and level of detail selection.



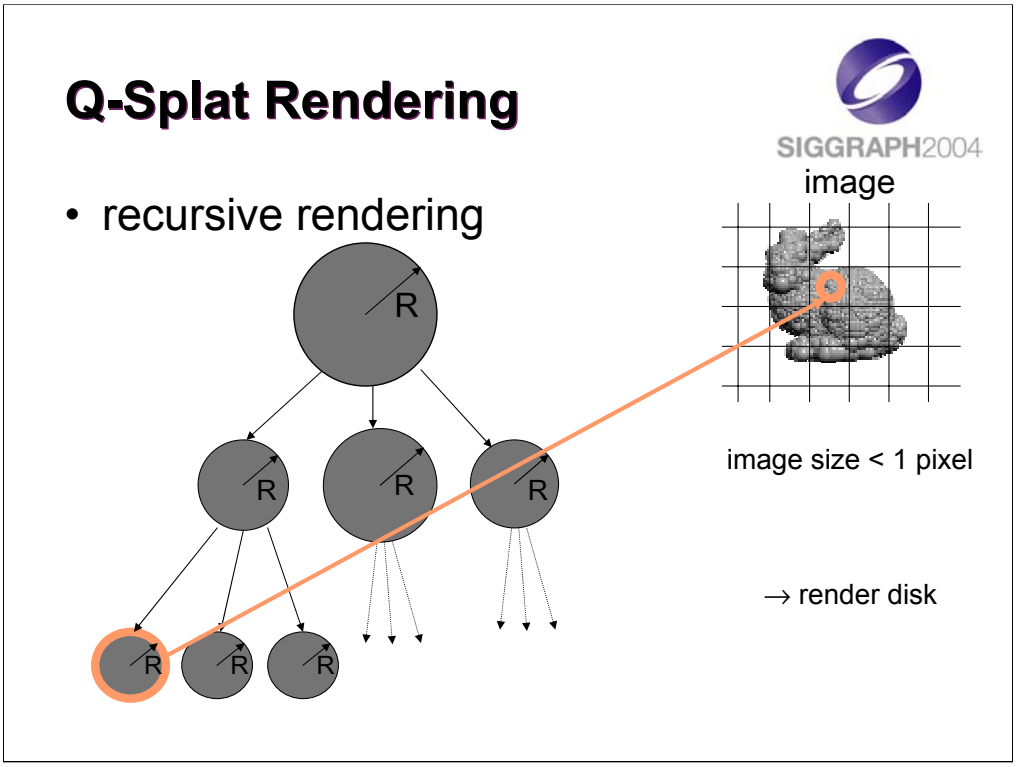
The traversal of the sphere hierarchy always begins at the root node.

The bounding sphere of the currently considered node is projected onto the image plane. If its size is larger than a user defined threshold, e.g. 1 pixel, the image error is too large and the child nodes are traversed.



As we can see in this illustration, the image size of the node in the next hierarchy level is still slighty above our threshold.

So we skip this node, too…



Finally, the image size of this node is clearly below our threshold and it is rendered. Typically we render disks or simple opaque squares instead of spheres.

Hierarchical Processing

- straightforward, but drawbacks
- not continuously stored in array
- not sequential
- traversal by CPU, rendering by GPU
- CPU is bottleneck

→ sequential version ?

This kind of data processing is very simple and straightforward, but unfortunately it has some major drawbacks for use with nowadays graphics hardware.

The rendered points are not continuously stored in an array and are not processed sequentially. The CPU traverses the tree and invokes the rendering of independent nodes. This ends up in a bottleneck and the power of the GPU is not fully used.

So what we would like to have, is a sequential version of the adaptive point selection. So we can offload more work to the GPU and bypass the bottlenecks.

Sequential Point Trees

- Q-Splat: render node if
  - image size ≤ threshold ***and***
  - image size of parent > threshold
  - image size = radius / view distance
- store with node $n.d_{min}$ = n.radius / threshold
- render node n if
  - view distance(n) ≥ $n.d_{min}$ and
  - view distance(parent) < $parent.d_{min}$

To develop a sequential version of the traversal, let‘s have a look what happens with a single node of the Q-Splat tree.

A node is rendered only if it‘s image size is below a certain threshold AND the image size of it‘s parent node is above the threshold.

This second criterion is false, if a predecessor is already small enough to be rendered, so the hierarchical traversal of the sub-tree would be skipped.

The image size of a node can be computed very easily. It is simply the radius of the sphere over the view distance.

Instead of regarding the image size of a node, we use another equivalent measure. For each node we can derive a minimum distance d_min, which equals to its radius over the threshold. If the node is closer to the viewer than d_min, its image size is too large and we don‘t want to render it.

Then we can redraft our criterion: a node is rendered, if it‘s view distance is above or equal to d_min, and the view distance of its parent node is smaller than the parents‘ d_min value.

Sequential Point Trees

- if we assume

  $\texttt{view distance(n)} \approx \texttt{view distance(parent)}$

- and set:

  $n.d_{max} = parent.d_{min}$

- criterion for node n:

  $n.d_{min} \leq \texttt{view distance(n)} < n.d_{max}$

image size small enough

parent image size too big

For now let‘s assume, that the view distances of child nodes and their parent nodes are roughly equal.

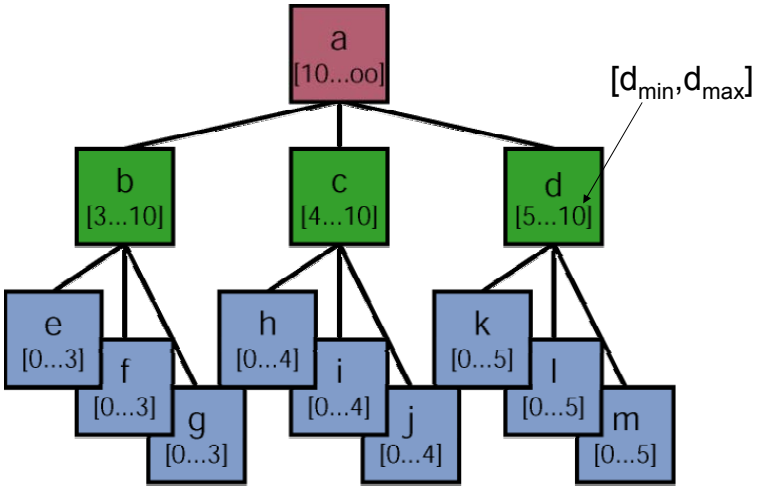
In addition to the d_min value, we store a d_max value for each node, which is simply the d_min value of its parent node.

In this way, we can get rid of the hierarchical data structure and define a two-sided test as rendering criterion for a single node.

The d_min boundary guarantees, that the image size of the node is small enough, the d_max boundary discards this node if its parent node should be rendered instead.

Sequential Point Trees

- example tree



This slide shows an example tree with 13 nodes. Each node brings along ist d_min d_max interval as shown.

Sequential Point Trees

- loop over all tree nodes
- first sequential version

```
foreach tree node n
  d = view distance(n);
  if ( n.d_min ≤ d < n.d_max )
    renderPoint(n);
```

With this criterion we can formulate the rendering as a simple loop over all tree nodes.

For each node, we calculate its view distance and render the node, if it passes the interval test.

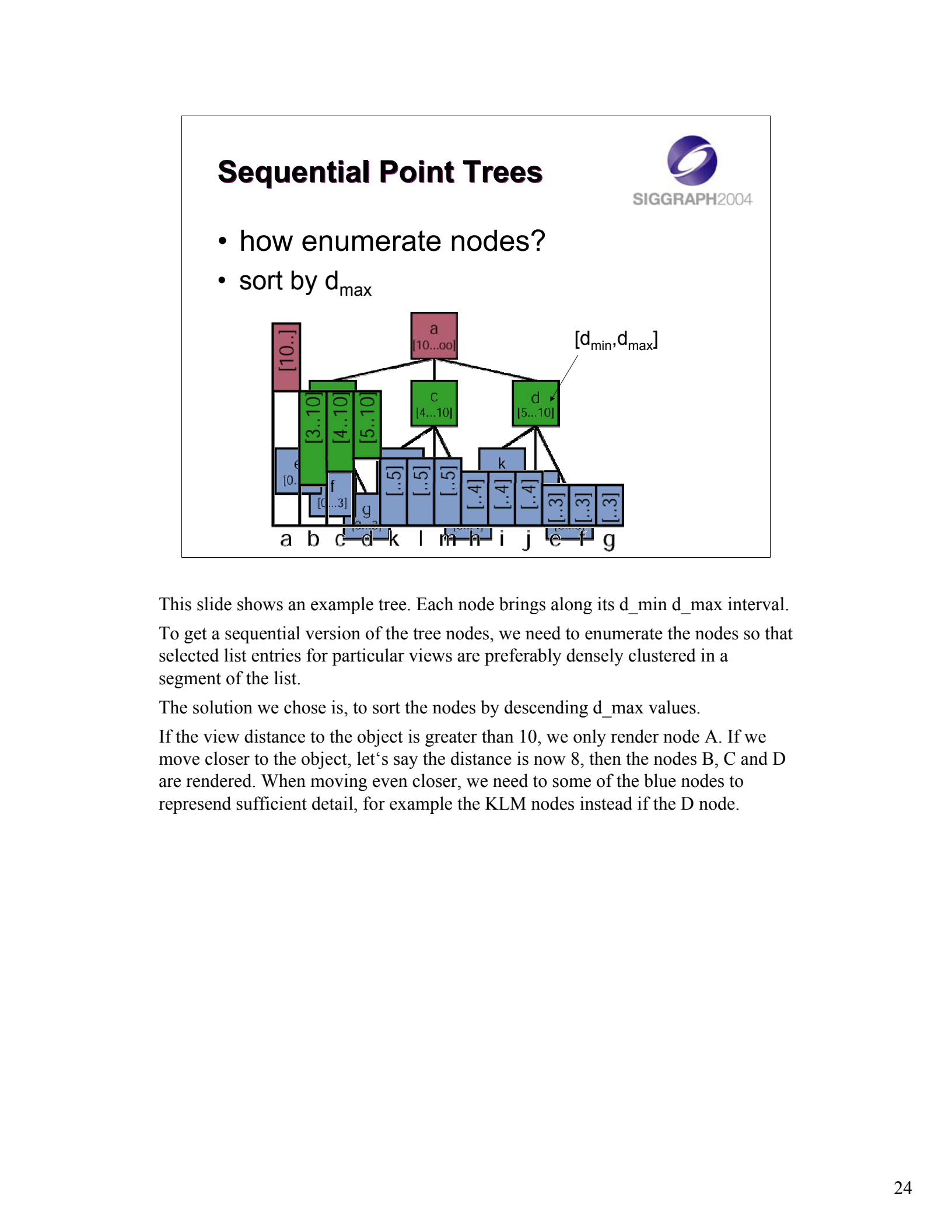
Sequential Point Trees

- account for
  view distance(n) ≠
  view distance(parent):
  - $n.d_{max} = parent.d_{min} + |parent-n|$
  - partially parent and some children selected
  - overdraw, but no visible artifacts

In general the view distance of a node and the view distance of its parent node are not equal. So our previous assumption can cause rendering errors. To prevent this, we introduce an interval overlap and increase a node‘s d_max value by the distance between the child and the parent node.

This overlap ensures that no holes appear, but it also means that for some nodes both, the node and some of its childrens are selected. This results in overdraw and very slightly reduced performance, but we did not experience visible artefacts from it.

Sequential Point Trees



- how enumerate nodes?
- sort by d_{max}

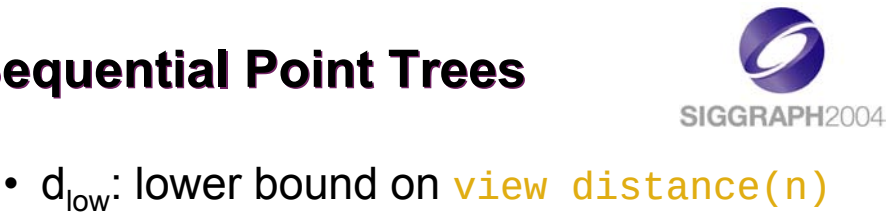
This slide shows an example tree. Each node brings along its d_min d_max interval.

To get a sequential version of the tree nodes, we need to enumerate the nodes so that selected list entries for particular views are preferably densely clustered in a segment of the list.

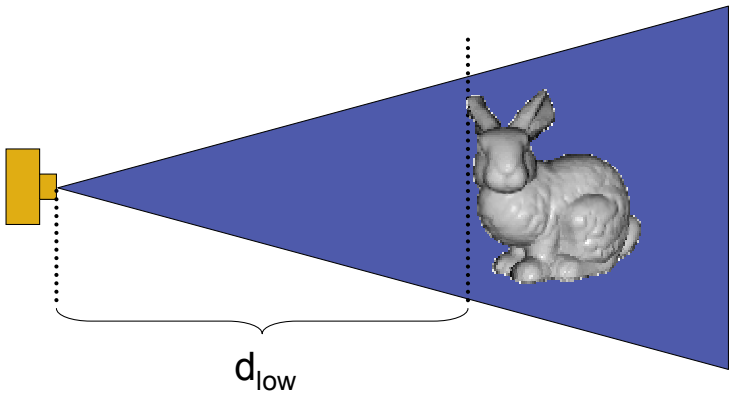
The solution we chose is, to sort the nodes by descending d_max values.

If the view distance to the object is greater than 10, we only render node A. If we move closer to the object, let‘s say the distance is now 8, then the nodes B, C and D are rendered. When moving even closer, we need to some of the blue nodes to represend sufficient detail, for example the KLM nodes instead if the D node.

Sequential Point Trees



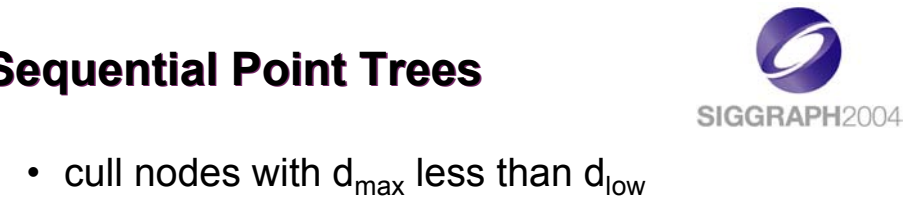
- d_{low}: lower bound on `view distance(n)`



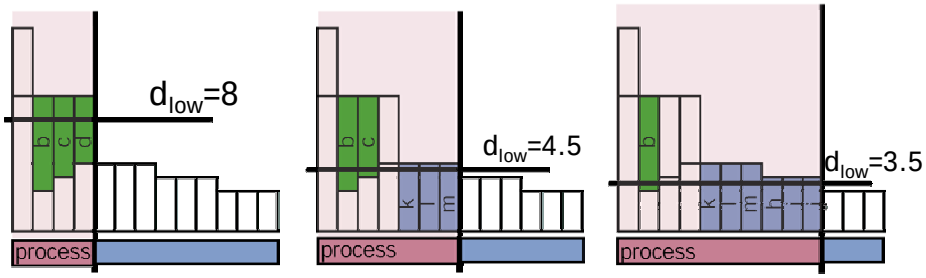
During rendering, we calculate a lower bound of the node‘s view distances for the whole object. This can be done very easily, for example by using bounding volumes.

This value is denoted by d_low.

Sequential Point Trees



- cull nodes with d_{max} less than d_{low}
- interval test for remaining prefix
  - view distance is not constant for all nodes !



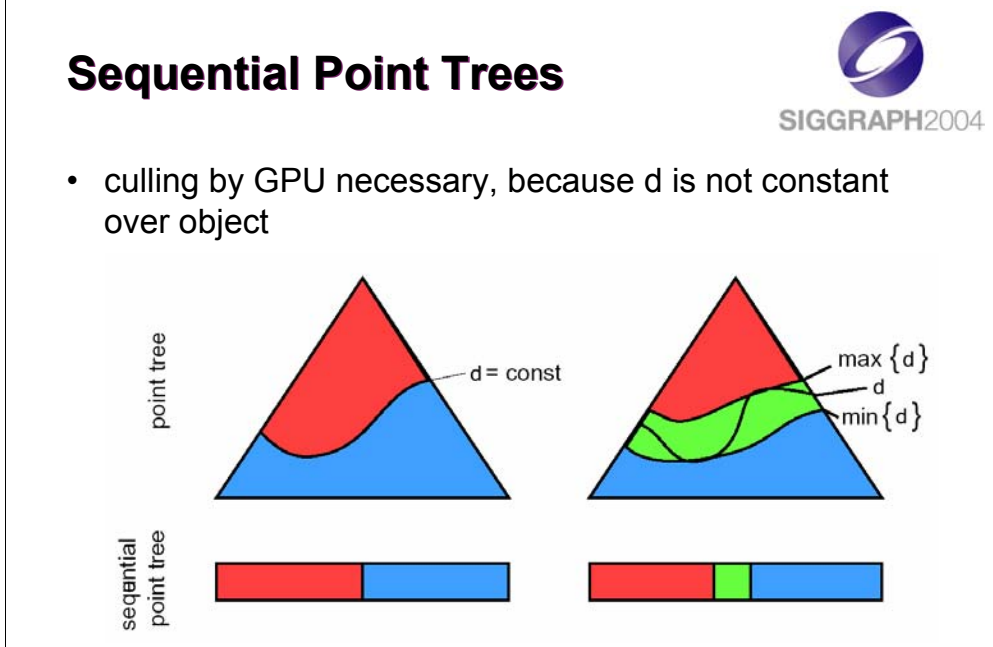
All nodes, which have a d_max value smaller than our lower bound can be immediatly skipped. These are the nodes at the tail of the list.

Only the nodes in a prefix of the list MAY pass the interval test.

On this slide, you can see the prefixes resulting from different three different d_low values for our example tree.

Since the view distance is not constant for all nodes and the prefix contains nodes not suitable for the current view, we still need culling for individual nodes by the GPU.

Sequential Point Trees



Up to now, we assumed that the view distance is the same for all points. In this case, every view distance d defines front of points in the hierarchy that is to be rendered for this view distance. The SPT is sorted such that this point front and all points above them are the prefix of the SPT array. In fact, the view distance varies over the object. By this, we have an additional (green) interval, where we between the point front for the minimum and maximum view distance. For points inside this interval, we have to evaluate the view distance and make the test again. The point front to be rendered is in this interval region.

Sequential Point Trees

- CPU does per frame:
  - compute d_{low}
  - binary search: last node with $d_{max} \geq d_{low}$
  - send draw command for prefix to GPU
  - store SPT in video memory
- GPU then does for every node n
  - compute d = view distance(n)
  - if $n.d_{min} \leq d < n.d_{max}$
    - render node

This slide summarizes all tasks needed for the rendering.

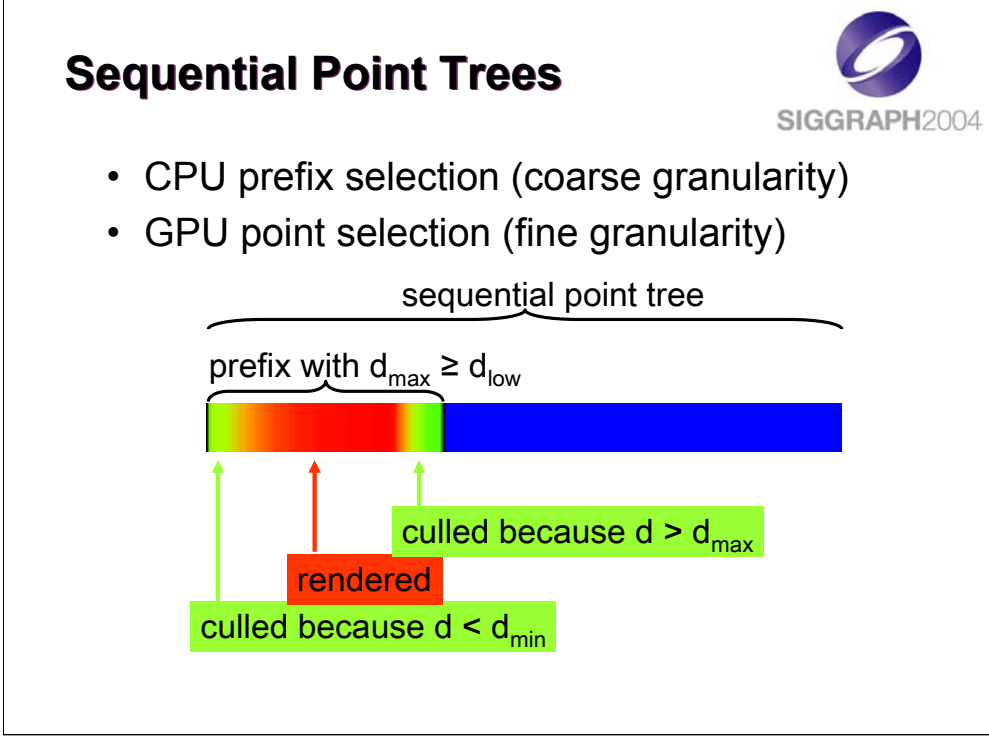
For each frame, the CPU computes the lower bound of node distances.

Using a binary search, the prefix can be determined very quickly and the CPU can send the draw command to the GPU.

The sequential point tree is preferably stored in video memory as a point primitive list, where the d_min d_max interval is stored as an attribute for each point.

The GPU, using vertex programs or vertex shaders, computes the view distance for each node. The view distance is simply the z value of the clip coordinates of the point. If a node passes the test, it is rendered, otherwise, the point primitive is culled by moving it to infinity.

Sequential Point Trees



So the CPU does fast coarse granularity culling and the GPU selects single points for rendering.

This slide shows the whole sequential point tree and the selected prefix.

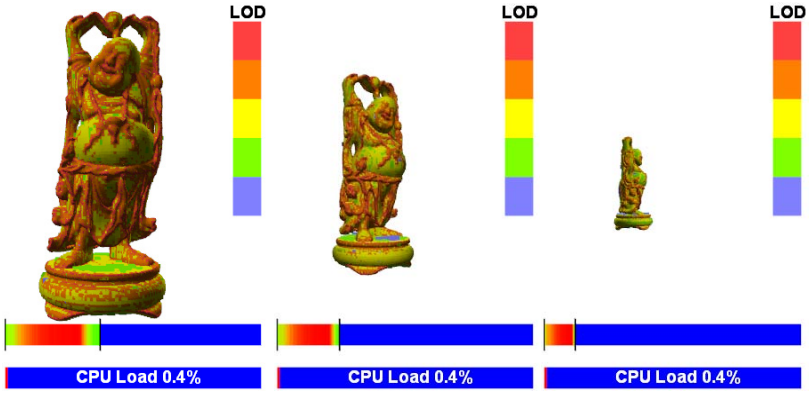
Nodes at the beginning of the prefix may be culled because they are too far up the hierachy and thus too large. They are outside the d_min boundary. They are not really critical, because in a tree with branching factor four, only ¾ of the nodes are leaves. We could make an equivalent search for a left boundary, however, the list is not sorted for d_min, so this search is costly and we will not find a tight bound.

At the end of the prefix nodes are culled, because they are to far down the hierarchy. They are only within the prefix, because the prefix length is determined conservatively.

Both tests have to be done by the GPU as they require the exact view distance of each node.

Sequential Point Trees

- example



Sequential Point Trees

- Results
  - culling by GPU: only 10 - 40%
  - on a 2.4 GHz Pentium with Radeon 9700:
  - CPU-Load < 20% (usually much less)
  - > 50 Mio points *after* culling

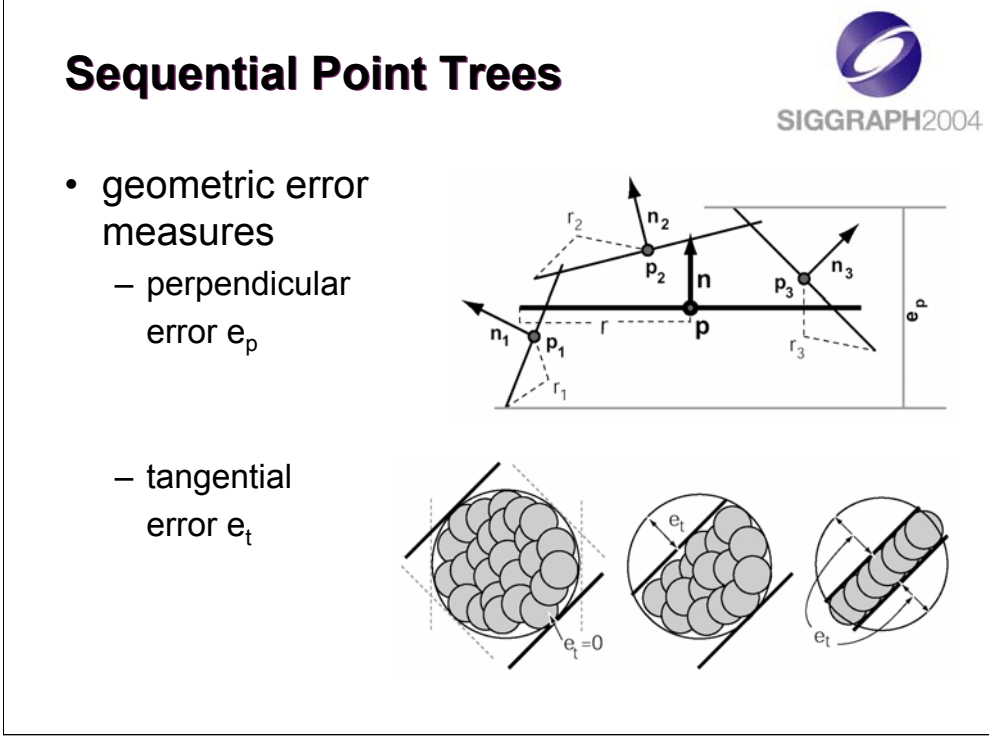
Before we‘re going more into detail of the error measurements and improvements, let‘s have a look at the results.

The percentage of points culled by the GPU depends on the variation of the view distances over the object. In typical examples this fraction is 10 to 40% of the prefix.

On a 2,4 GHz Pentium 4 processor with a radeon 9700, the CPU load is always below 20%. Usually it is much less. When rendering a single sequential point tree, the CPU load is even below 1%.

The effective point rate, that means the number of points which pass the interval test and are rendered, is more than 50 million points per second.

Sequential Point Trees



Let‘s have a look at further improvements…

Every node in the hierarchy can be approximated by a disk with a certain center, normal and diameter.

To introduce a more sophisticated error measure, we distinguish two different types of geometric errors:

1.) The first one is the perpendicular error. It is the minimum distance between two planes parallel to a parent disk, that encloses all child disks. It captures the fact, that errors along edges are less acceptable.

2.) Second, we introduce the tangential error. It looks at the projection of the child disks onto the parent disk. It measures if the parent disk covers and unnecessary large area, resulting in typical errors at surface edges. We measure it by fitting two parallel slabs of varying orientation around the projected child disks. The tangential error is defined as the difference of the disk diameter and the width of the tightest slab.

Sequential Point Trees

- error projected into image
  - e_p scales with $\sin(\alpha)/d$
    e_t scales with $\cos(\alpha)/d$
    α=angle(normal, view direction)
- single geometric error
  $e_g = \max_\alpha\{e_p \sin\alpha + e_t \cos\alpha\} = (e_p^2 + e_t^2)^{1/2}$
- e_g scales with 1/d

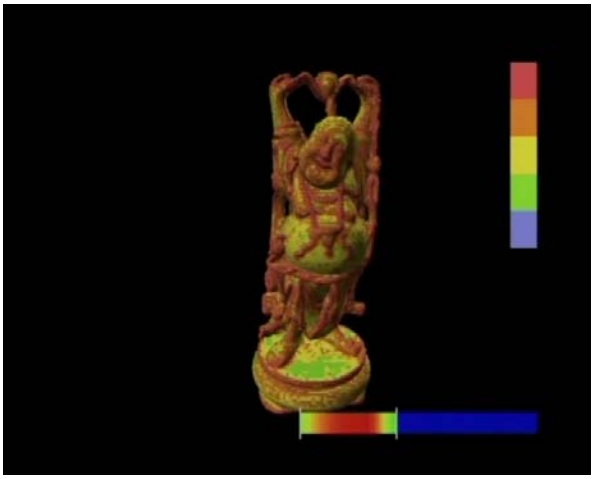
When the geometric errors are projected into the image, then the image space counterpart of the perpendicular error is propotional to the sine of alpha and 1 over d.

The tangential error in image space is proportional with the cosine of alpha and 1 over d. Where alpha is the angle between the disk normal and the view direction.

To fit these error measures into the sequential point trees, it‘s important to obtain a single geometric error e_g, as shown on the slide, which scales with 1 over the view distance.

Sequential Point Trees

- less, but larger splats in less detailed regions



Results:

In flat parts of the surface, less but larger splats are rendered, here shown in yellow and green.

More detailed parts require more point samples shown in red.

The bottom right bar, shows the sequential point tree and the prefix sent to the GPU in red and green. If the object moves away, the prefix becomes smaller. The blue fraction of the list is culled by the CPU.

Sequential Point Trees

- what about colors ?
- in flat textured regions
  - washing out texture detail
- add texture criterion
  - if significant color variation in child nodes: increase error to node diameter

Sequential Point Trees can also contain color information. Every leaf of the point hierarchy represents a part of a surface and an average color can be assigned.

For inner nodes of the hierarchy the color values of the child nodes are averaged. With color averaging, we have to reconsider our error measure. In flat regions with small geometric error, large splats are rendered and the texture detail is washed out.

To avoid this, we increase a node‘s error to the points diameter, if the color of child nodes varies significantly.

Sequential Point Trees

- combine errors
- $e_{tex} = \begin{cases} \text{diameter} & \text{if texture varies} \\ 0 & \text{else} \end{cases}$
- $e_{com} = \max(e_{tex}, e_g)$

  ⇒ image error = e_{com} / view distance
  ⇒ $n.d_{min} = n.e_{com}$ / threshold

Both errors scale with 1 over d, and can be combined to a single error measure to fit into the sequential point tree concept.

The texture error e_tex equals zero or the node`s diameter, depending on the color variation of the child nodes.

The combined error e_com is then the maximum of e_tex and the geometric error e_g.

The image error is then e_com over the view distance. And a node`s d_min value for the interval test is then simply e_com over the threshold.

Sequential Point Trees

- enforce small splats, reduce blurring to threshold
- point densities adapt to texture detail

This enforces small splats and the blurring is reduced to the error threshold. With this measure, point densities adapt to texture detail as you can see on the left image.

Sequential Point Trees

- hybrid point-polygon rendering
- don‘t render large triangles with points

Sequential point trees can be extended to hybrid point-polygon rendering.

In these images, we can see a very artificial example of a pillar. On the left image, it is smooth shaded, on the right hand side, the parts shown in red are rendered using points, the parts with less curvature are rendered as triangles, shown in gray.

Sequential Point Trees

- combine with polygonal rendering
  - for every triangle
    - compute d_{max} = longest side / threshold
    - remove all points from triangle with smaller d_{max}
  - sort triangles for d_{max}
  - during rendering
    - send triangles with $d_{max} < d_{low}$ to GPU
    - on the GPU (vertex program)
      - test $d < d_{max}$
      - cull by alpha-test
      - border case: differently classified vertices
        ⇒ partially rendered triangles

Rendering a triangle is reasonable as long as its longest side has an image size above our error threshold.

Thus we can compute a d_max value for each triangle.

We can remove all point samples of a triangle which have a smaller d_max value than the triangle. Because it will always pay off to render the triangle instead of these points.

Like the point hierarchy, we sort all triangles according to their d_max value.

During rendering, the required prefix of the triangle list is determind in the same way as for the point list.

A vertex program evaluates the d_max condition for every vertex and puts the result into the alpha value of the vertex. Culling is then done by an alpha-test. By this, triangles with differently classified vertices are rendered partially. Since this is a border case, the corresponding point samples are also rendered and resulting holes are automatically filled.

Sequential Point Trees

- pros
  - very simple!
  - continuous level of detail
  - mostly on the GPU
  - GPU runs at maximum efficiency
- cons
  - no hierarchical view frustum culling
  - currently: bad splatting support by GPU

The sequential point trees are very simple and easy to implement and provide continuous level of detail rendering.

Since almost all work is moved to the GPU, the CPU load is very low and can be used for other tasks.

If the point list is stored in video memory, the GPU runs at maximum efficiency and the rendering speed is limited by the geometry processing, when using rather simple splatting techniques.

The sequential point trees do not allow hierarchical frustum culling within an object. Objects can only be culled on the whole or have to be split into several point trees. And this is, what we did for backface culling: We split up all points samples into several normal clusters and only render front facing clusters. This requires slightly more computation, but skips about 50 percent of the point samples.

Unfortunately contemporary graphic hardware has bad support for splatting techniques.

Sequential Point Trees

- video: sculpture garden

In this video, we can see a real-life example captured from a Radeon 9700. The sculptures and the trees are rendered using sequential point trees, the other objects in the scene and the sky is rendered as textured triangles.

The red-blue bar shows the very low CPU load required for this scene.

The flickering seen at the sculptures results from the simplistic splatting technique we used for this demo.

Questions ?

…thank you for your attention !

contact: marc.stamminger@cs.fau.de
url: www9.cs.fau.de/Research/Rendering

SIGGRAPH2004

Spectral Processing of Point-Sampled Geometry

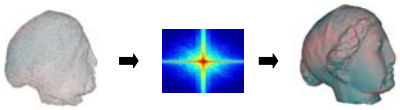
Markus Gross

Overview

- Introduction
- Fourier transform
- Spectral processing pipeline
- Applications
  - Spectral filtering
  - Adaptive subsampling
- Summary

Introduction

- Idea: Extend the Fourier transform to manifold geometry



⇨ Spectral representation of point-based objects

⇨ Powerful methods for digital geometry processing

Introduction

- Applications:
  - Spectral filtering:
    - Noise removal
    - Microstructure analysis
    - Enhancement
  - Adaptive resampling:
    - Complexity reduction
    - Continuous LOD

Fourier Transform

- 1D example:

$$X_n = \sum_{k=1}^{N} x_k e^{-j2\pi\frac{nk}{N}}$$

output signal; input signal; spectral basis function

- Benefits:
  - Sound concept of frequency
  - Extensive theory
  - Fast algorithms

Fourier Transform

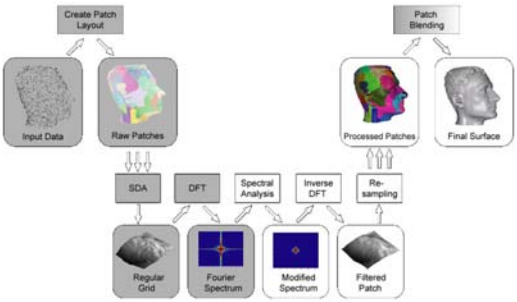
- Requirements:
  - Fourier transform defined on Euclidean domain
    - ⇨ we need a global parameterization
  - Basis functions are eigenfunctions of Laplacian operator
    - ⇨ requires regular sampling pattern so that basis functions can be expressed in analytical form (fast evaluation)
- Limitations:
  - Basis functions are globally defined
    - ⇨ Lack of local control

Approach

- Split model into patches that:
  - are parameterized over the unit-square
    - ⇨ mapping must be continuous and should minimize distortion
  - are re-sampled onto a regular grid
    - ⇨ adjust sampling rate to minimize information loss
  - provide sufficient granularity for intended application (local analysis)

⇨ process each patch individually and blend processed patches

Spectral Pipeline



Patch Layout Creation

Clustering ⇨ Optimization

Samples ⇨ Clusters ⇨ Patches

Patch Layout Creation

- Iterative, local optimization method
- Merge patches according to quality metric:

$$\Phi = \Phi_S \cdot \Phi_{NC} \cdot \Phi_B \cdot \Phi_{Reg}$$

Φ_S ⇨ patch Size

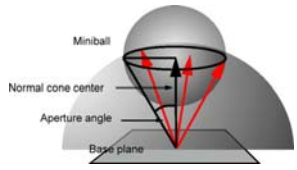
Φ_{NC} ⇨ curvature

Φ_B ⇨ patch boundary

Φ_{Reg} ⇨ spring energy regularization

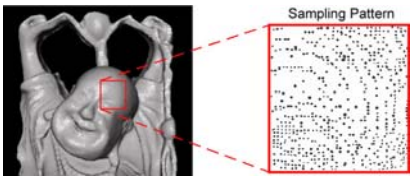
Patch Layout Creation

- Parameterize patches by orthogonal projection onto base plane
- Bound normal cone to control distortion of mapping using smallest enclosing sphere



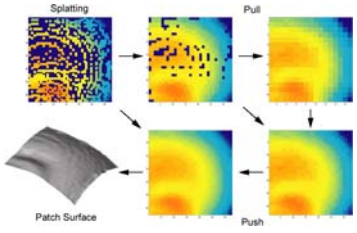
Patch Resampling

- Patches are irregularly sampled



Patch Resampling

- Resample patch onto regular grid using hierarchical push-pull filter (scattered data approximation)



Spectral Analysis

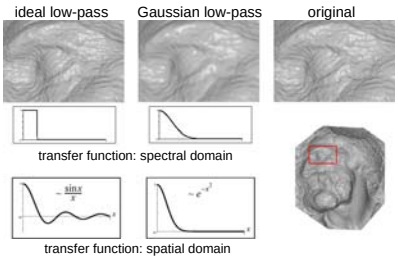
- 2D discrete Fourier transform (DFT)
  - ⇨ Direct manipulation of spectral coefficients
- Filtering as convolution:

$$F(x \otimes y) = F(x) \cdot F(y)$$

  - ⇨ Convolution: $O(N^2)$ ⇨ multiplication: $O(N)$
- Inverse Fourier transform
  - ⇨ Filtered patch surface

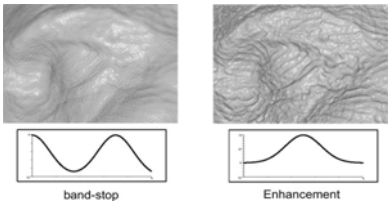
Spectral Filters

- Smoothing filters



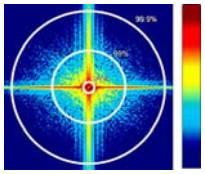
Spectral Filters

- Microstructure analysis and enhancement



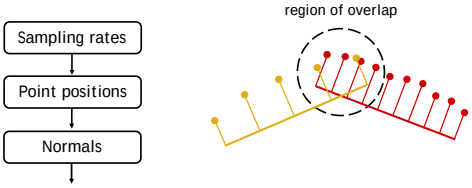
Spectral Resampling

- Low-pass filtering
  - ⇨ Band-limitation
- Regular Resampling
  - ⇨ Optimal sampling rate (sampling theorem)
  - ⇨ Error control (Parseval's theorem)



Reconstruction

- Filtering can lead to discontinuities at patch boundaries
  - ⇨ Create patch overlap, blend adjacent patches

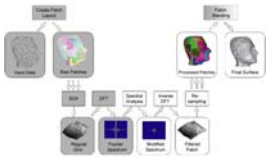


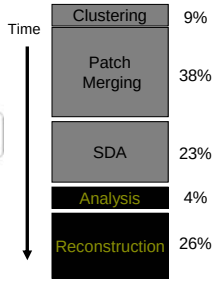
Reconstruction

- Blending the sampling rate

blended sampling rate in region of patch overlap → discretized sampling rate on regular grid → pre-computed sampling patterns

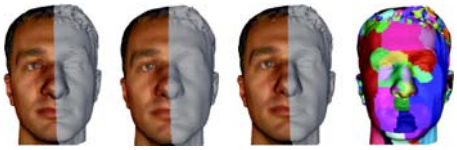
Timings





Applications

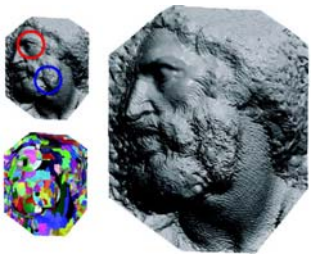
- Surface Restoration



Original | Gaussian low-pass | Wiener filter | Patch layout

Applications

- Interactive filtering



Applications

- Adaptive Subsampling

4,128,614 pts. = 100% | 287,163 pts. = 6.9%

Applications - Watermarking

- Spectral embedding and readout

Input model | Patching & spectral decomposition | Combined attack (noise, clip, affine transform)

Cotting, Weyrich, Pauly, Gross, SMI 04

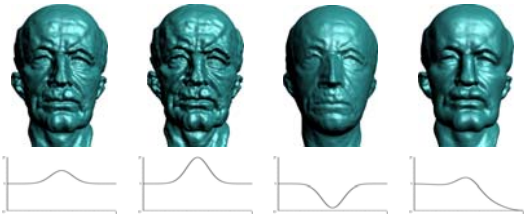
Extensions

- Scale spaces: levels of smoothness

- Sequence of progressively smoothed models
- Partitioning into spectral bands

Filtering

Pauly, Kobbelt, Gross, ACM TOG, to appear



Summary

- Versatile spectral decomposition of point-based models
- Effective filtering and spectral analysis
- Adaptive resampling
- Efficient processing of large point-sampled models

Reference

- Pauly, Gross: *Spectral Processing of Point-sampled Geometry*, SIGGRAPH 2001

SIGGRAPH2004

Surface Simplification

Mark Pauly Stanford University

Overview

- Introduction
- Local surface analysis
- Simplification methods
- Error measurement
- Comparison

Introduction

- Point-based models are often sampled very densely
- Many applications require coarser approximations, e.g. for efficient
  - Storage
  - Transmission
  - Processing
  - Rendering

⇨ We need simplification methods for reducing the complexity of point-based surfaces

Introduction

- Example: Level-of-detail (LOD) rendering

10k 20k 60k 200k 2000k

Introduction

- We transfer different simplification methods from triangle meshes to point clouds:
  - Hierarchical clustering
  - Iterative simplification
  - Particle simulation
- Depending on the intended use, each method has its pros and cons (see comparison)

Local Surface Analysis

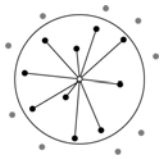
- Cloud of point samples describes underlying (manifold) surface
- We need:
  - Mechanisms for locally approximating the surface ⇨ MLS approach
  - Fast estimation of tangent plane and curvature ⇨ principal component analysis of local neighborhood

Neighborhood

- No explicit connectivity between samples (as with triangle meshes)
- Replace geodesic proximity with spatial proximity (requires sufficiently high sampling density!)
- Compute neighborhood according to Euclidean distance

Neighborhood

- K-nearest neighbors



  - Can be quickly computed using spatial data-structures (e.g. kd-tree, octree, bsp-tree)
  - Requires isotropic point distribution

Neighborhood

- Improvement: Angle criterion (Linsen)

  - Project points onto tangent plane
  - Sort neighbors according to angle
  - Include more points if angle between subsequent points is above some threshold

Neighborhood

- Local Delaunay triangulation (Floater)

  - Project points into tangent plane
  - Compute local Voronoi diagram

Covariance Analysis

- Covariance matrix of local neighborhood N:

$$\mathbf{C} = \begin{bmatrix} \mathbf{p}_{i_1} - \bar{\mathbf{p}} \\ \cdots \\ \mathbf{p}_{i_n} - \bar{\mathbf{p}} \end{bmatrix}^T \cdot \begin{bmatrix} \mathbf{p}_{i_1} - \bar{\mathbf{p}} \\ \cdots \\ \mathbf{p}_{i_n} - \bar{\mathbf{p}} \end{bmatrix}, \quad i_j \in N$$

- with centroid $\bar{\mathbf{p}} = \frac{1}{|N|} \sum_{i \in N} \mathbf{p}_i$

Covariance Analysis

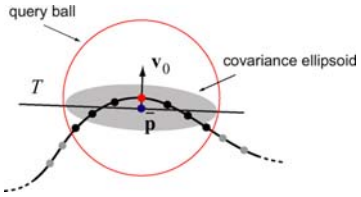
- Consider the eigenproblem:

$$\mathbf{C} \cdot \mathbf{v}_l = \lambda_l \cdot \mathbf{v}_l, \qquad l \in \{0,1,2\}$$

- C is a 3x3, positive semi-definite matrix
  - ⇨ All eigenvalues are real-valued
  - ⇨ The eigenvector with smallest eigenvalue defines the least-squares plane through the points in the neighborhood, i.e. approximates the surface normal

Covariance Analysis

- Covariance ellipsoid spanned by the eigenvectors scaled with corresponding eigenvalue



Covariance Analysis

- The total variation is given as:

$$\sum_{i \in N} |\mathbf{p}_i - \bar{\mathbf{p}}|^2 = \lambda_0 + \lambda_1 + \lambda_2$$

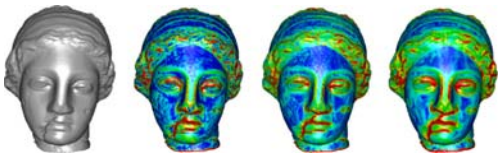
- We define surface variation as:

$$\sigma_n(\mathbf{p}) = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2}, \qquad \lambda_0 \le \lambda_1 \le \lambda_2$$

  - Measures the fraction of variation along the surface normal, i.e. quantifies how strong the surface deviates from the tangent plane ⇨ estimate for curvature

Covariance Analysis

- Comparison with curvature:



original mean curvature variation n=20 variation n=50

Surface Simplification

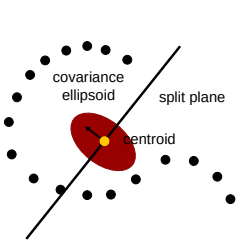
- Hierarchical clustering
- Iterative simplification
- Particle simulation

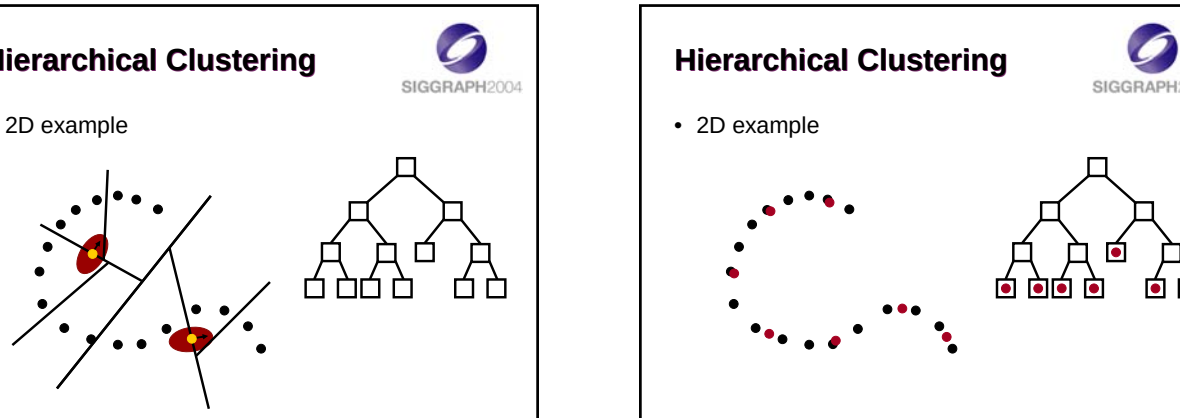
Hierarchical Clustering

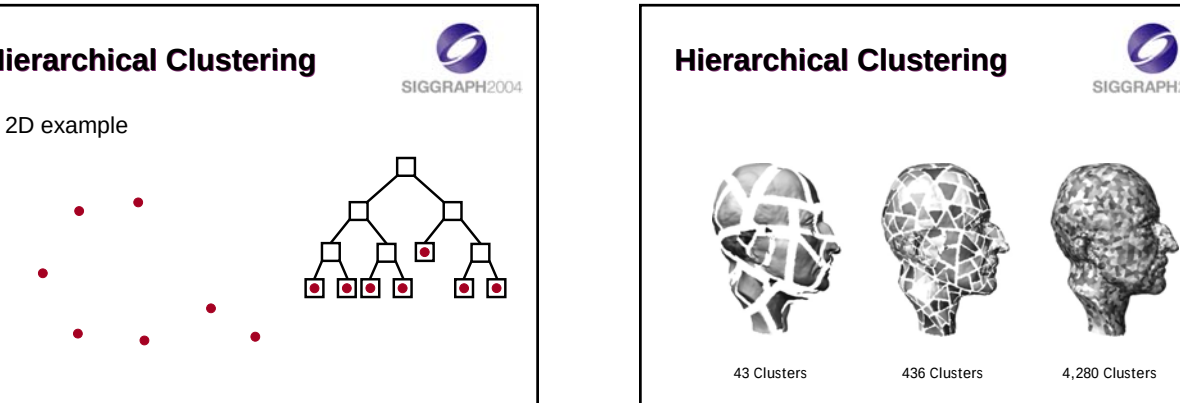
- Top-down approach using binary space partition:
- Split the point cloud if:
  - Size is larger than user-specified maximum or
  - Surface variation is above maximum threshold
- Split plane defined by centroid and axis of greatest variation (= eigenvector of covariance matrix with largest associated eigenvector)
- Leaf nodes of the tree correspond to clusters
- Replace clusters by centroid

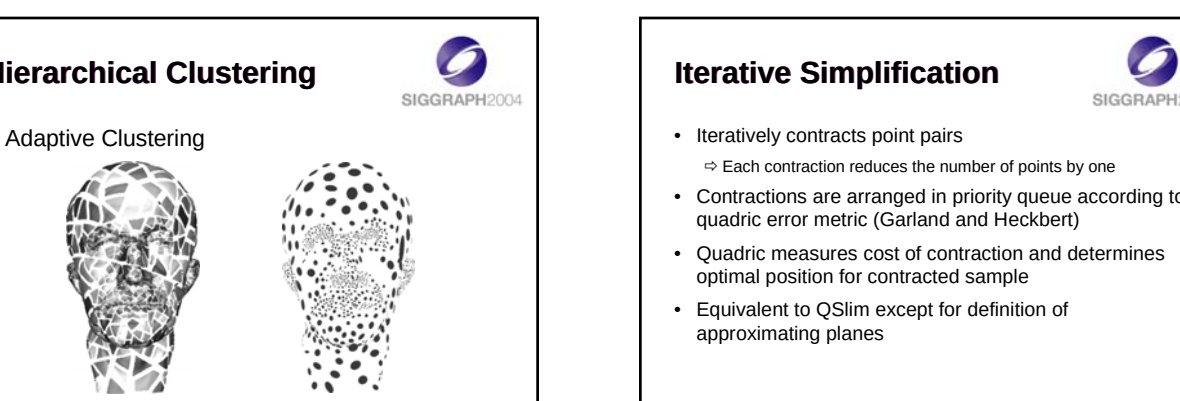
Hierarchical Clustering

- 2D example









Hierarchical Clustering

- 2D example

Hierarchical Clustering

- 2D example

Hierarchical Clustering

- 2D example

Hierarchical Clustering

43 Clusters 436 Clusters 4,280 Clusters

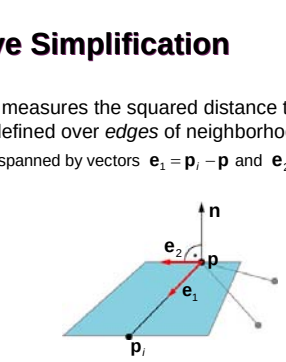
Hierarchical Clustering

- Adaptive Clustering

Iterative Simplification

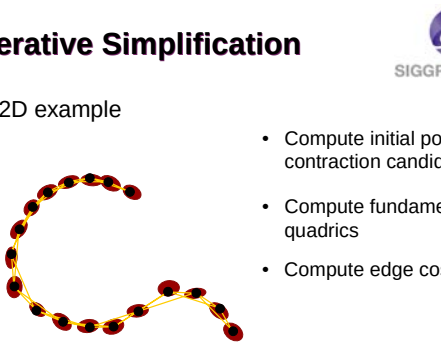
- Iteratively contracts point pairs
  - ⇨ Each contraction reduces the number of points by one
- Contractions are arranged in priority queue according to quadric error metric (Garland and Heckbert)
- Quadric measures cost of contraction and determines optimal position for contracted sample
- Equivalent to QSlim except for definition of approximating planes

Iterative Simplification



- Quadric measures the squared distance to a set of planes defined over *edges* of neighborhood
  - plane spanned by vectors $\mathbf{e}_1 = \mathbf{p}_i - \mathbf{p}$ and $\mathbf{e}_2 = \mathbf{e}_1 \times \mathbf{n}$

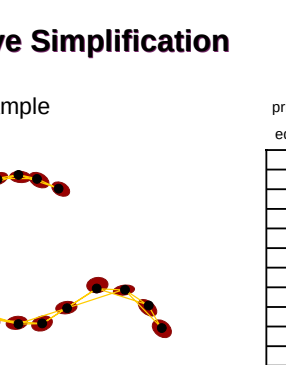
Iterative Simplification



- 2D example

- Compute initial point-pair contraction candidates
- Compute fundamental quadrics
- Compute edge costs

Iterative Simplification

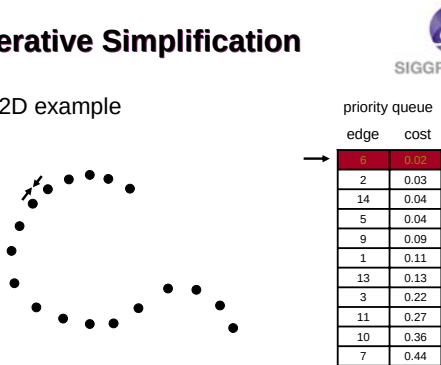


- 2D example

priority queue

edge	cost
6	0.02
2	0.03
14	0.04
5	0.04
9	0.09
1	0.11
13	0.13
3	0.22
11	0.27
10	0.36
7	0.44
4	0.56

Iterative Simplification

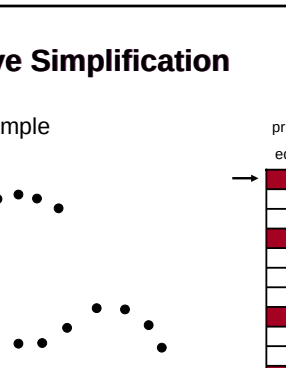


- 2D example

priority queue

edge	cost
6	0.02
2	0.03
14	0.04
5	0.04
9	0.09
1	0.11
13	0.13
3	0.22
11	0.27
10	0.36
7	0.44
4	0.56

Iterative Simplification

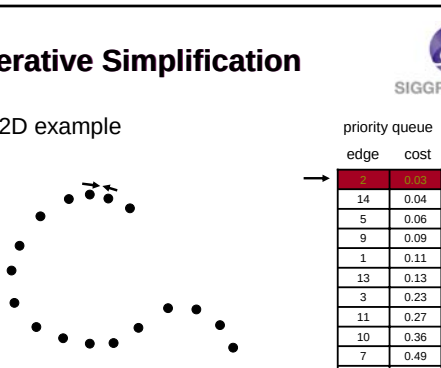


- 2D example

priority queue

edge	cost
6	0.02
2	0.03
14	0.04
5	0.06
9	0.09
1	0.11
13	0.13
3	0.23
11	0.27
10	0.36
7	0.49
4	0.56

Iterative Simplification



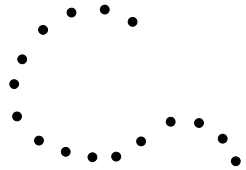
- 2D example

priority queue

edge	cost
2	0.03
14	0.04
5	0.06
9	0.09
1	0.11
13	0.13
3	0.23
11	0.27
10	0.36
7	0.49
4	0.56

Iterative Simplification

- 2D example

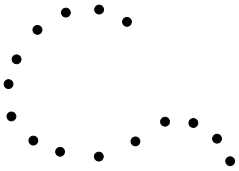


priority queue

edge	cost
2	0.03
14	0.04
5	0.06
9	0.09
1	0.11
13	0.13
3	0.23
11	0.27
10	0.36
7	0.49
4	0.56

Iterative Simplification

- 2D example

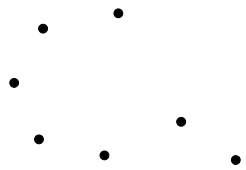


priority queue

edge	cost
14	0.04
5	0.06
9	0.09
1	0.11
13	0.13
3	0.23
11	0.27
10	0.36
7	0.49
4	0.56

Iterative Simplification

- 2D example



priority queue

edge	cost
11	0.27
10	0.36
7	0.49
4	0.56

Iterative Simplification

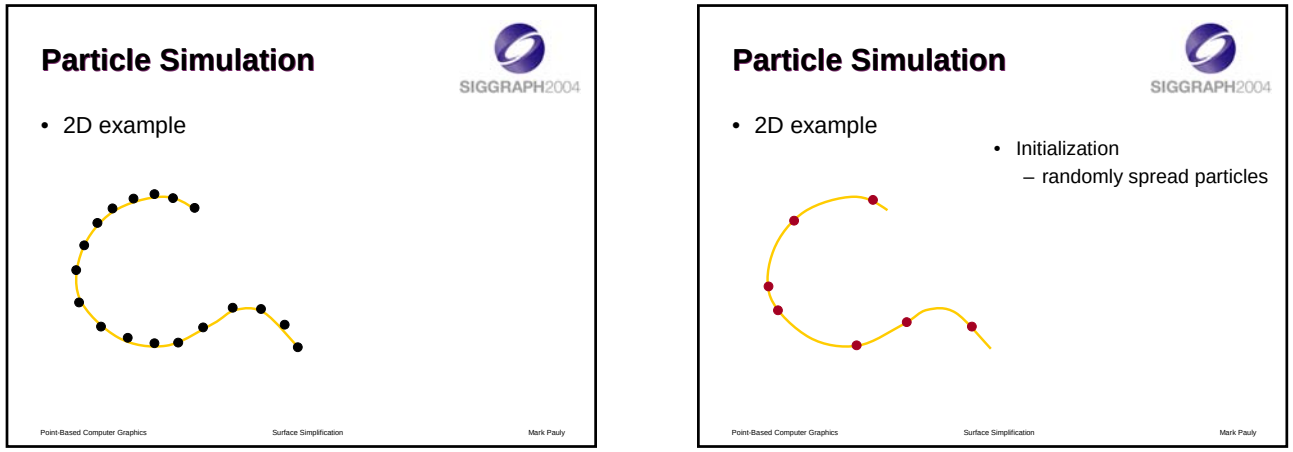
original model (296,850 points) | simplified model (2,000 points) | remaining point pair contraction candidates

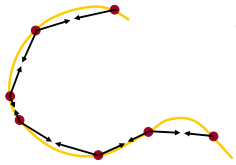
Particle Simulation

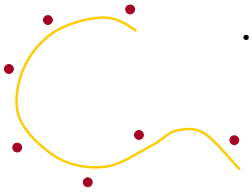
- Resample surface by distributing particles on the surface
- Particles move on surface according to inter-particle repelling forces
- Particle relaxation terminates when equilibrium is reached (requires damping)
- Can also be used for up-sampling!

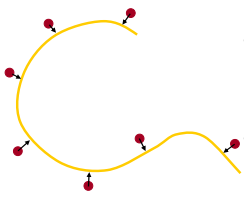
Particle Simulation

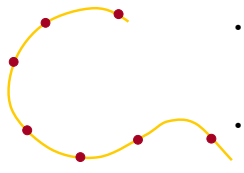
- Initialization
  - Randomly spread particles
- Repulsion
  - Linear repulsion force $F_i(\mathbf{p}) = k(r - \|\mathbf{p} - \mathbf{p}_i\|) \cdot (\mathbf{p} - \mathbf{p}_i)$
  - ⇨only need to consider neighborhood of radius r
- Projection
  - Keep particles on surface by projecting onto tangent plane of closest point
  - Apply full MLS projection at end of simulation











Particle Simulation

- 2D example

Particle Simulation

- 2D example

- Initialization
  - randomly spread particles

Particle Simulation

- 2D example

- Initialization
  - randomly spread particles
- Repulsion
  - linear repulsion force

$$F_i(\mathbf{p}) = k(r - \|\mathbf{p} - \mathbf{p}_i\|) \cdot (\mathbf{p} - \mathbf{p}_i)$$

Particle Simulation

- 2D example

- Initialization
  - randomly spread particles
- Repulsion
  - linear repulsion force

$$F_i(\mathbf{p}) = k(r - \|\mathbf{p} - \mathbf{p}_i\|) \cdot (\mathbf{p} - \mathbf{p}_i)$$

Particle Simulation

- 2D example

- Initialization
  - randomly spread particles
- Repulsion
  - linear repulsion force

$$F_i(\mathbf{p}) = k(r - \|\mathbf{p} - \mathbf{p}_i\|) \cdot (\mathbf{p} - \mathbf{p}_i)$$

- Projection
  - project particles onto surface

Particle Simulation

- 2D example

- Initialization
  - randomly spread particles
- Repulsion
  - linear repulsion force

$$F_i(\mathbf{p}) = k(r - \|\mathbf{p} - \mathbf{p}_i\|) \cdot (\mathbf{p} - \mathbf{p}_i)$$

- Projection
  - project particles onto surface

Particle Simulation

- Adaptive simulation
  - Adjust repulsion radius according to surface variation ⇨ more samples in regions of high variation

variation estimation

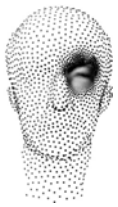
simplified model (3,000 points)

Particle Simulation

- User-controlled simulation
  - Adjust repulsion radius according to user input

uniform

original



selective

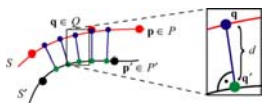
Particle Simulation

Measuring Error

- Measure the distance between two point-sampled surfaces using a sampling approach
- Maximum error: $\Delta_{max}(S,S') = \max_{\mathbf{q}\in Q} d(\mathbf{q},S')$
  - ⇨ Two-sided Hausdorff distance
- Mean error: $\Delta_{avg}(S,S') = \frac{1}{|Q|}\sum_{\mathbf{q}\in Q} d(\mathbf{q},S')$
  - ⇨ Area-weighted integral of point-to-surface distances
- Q is an up-sampled version of the point cloud that describes the surface S

Measuring Error

- $d(\mathbf{q},S)$ approximates the distance of point $\mathbf{q}$ to surface S using the MLS projection operator



Measuring Error

original

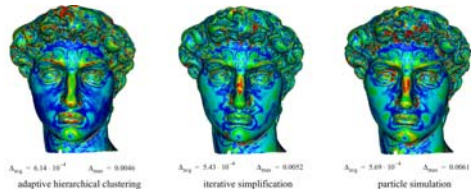
simplified

upsampled

error

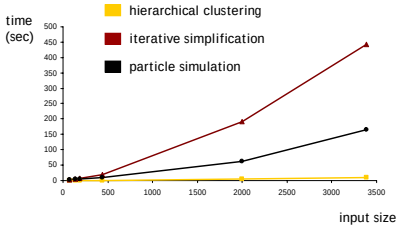
Comparison

- Error estimate for Michelangelo's David simplified from 2,000,000 points to 5,000 points



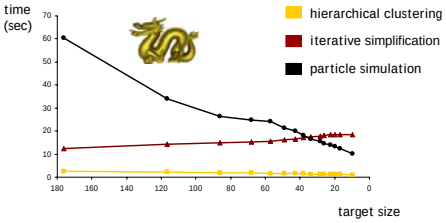
Comparison

- Execution time as a function of input model size (reduction to 1%)



Comparison

- Execution time as a function of target model size (input: dragon, 535,545 points)

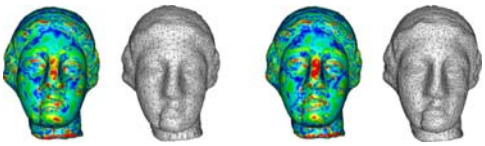


Comparison

- Summary

	Efficiency	Surface Error	Control	Implementation
Hierarchical Clustering	+	-	-	+
Iterative Simplification	-	+	o	o
Particle Simulation	o	+	+	-

Point-based vs. Mesh Simplification



point-based simplification with subsequent mesh reconstruction

mesh reconstruction with subsequent mesh simplification (QSlim)

⇨ point-based simplification saves an expensive surface reconstruction on the dense point cloud!

References

- Pauly, Gross: *Efficient Simplification of Point-sampled Surfaces*, IEEE Visualization 2002
- Shaffer, Garland: *Efficient Adaptive Simplification of Massive Meshes*, IEEE Visualization 2001
- Garland, Heckbert: *Surface Simplification using Quadric Error Metrics*, SIGGRAPH 1997
- Turk: *Re-Tiling Polygonal Surfaces*, SIGGRAPH 1992
- Alexa et al. *Point Set Surfaces*, IEEE Visualization 2001

An Interactive System for Point-based Surface Editing (Part I)

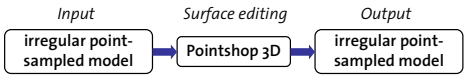
Markus Gross

Overview

- Introduction
- Pointshop3D System Components
  - Point Cloud Parameterization
  - Resampling Scheme
  - Editing Operators
- Summary
- See course notes

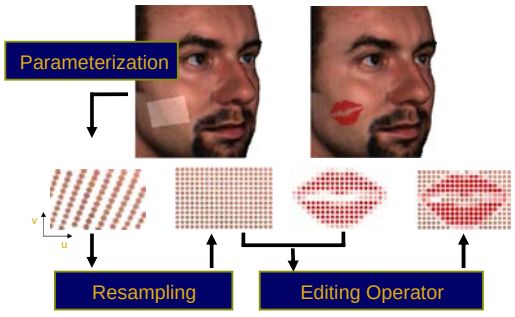
Pointshop 3D

- Interactive system for point-based surface editing
- Generalize 2D photo editing concepts and functionality to 3D point-sampled surfaces
- Use 3D surface pixels (*surfels*) as versatile display and modeling primitive



- **Does not require intermediate triangulation**

Concept



Key Components

- Point cloud parameterization Φ
  - brings surface and brush into common reference frame
- Dynamic resampling Ψ
  - creates one-to-one correspondence of surface and brush samples
- Editing operator Ω
  - combines surface and brush samples

$$S' = \Omega(\Psi(\Phi(S)), \Psi(B))$$

modified surface | original surface | brush

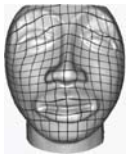
Parameterization

- Constrained minimum distortion parameterization of point clouds

$$\mathbf{u} \in [0,1]^2 \Rightarrow X(\mathbf{u}) = \begin{bmatrix} x(\mathbf{u}) \\ y(\mathbf{u}) \\ z(\mathbf{u}) \end{bmatrix} = \mathbf{x} \in P \subset R^3$$

Parameterization

contraints = matching of feature points



minimum distortion = maximum smoothness

Parameterization

- Find mapping X that minimizes objective function:

$$C(X) = \sum_{j \in M} \underbrace{(X(\mathbf{p}_j) - \mathbf{x}_j)^2}_{\text{fitting constraints}} + \varepsilon \underbrace{\int_P \gamma(\mathbf{u}) d\mathbf{u}}_{\text{distortion}}$$

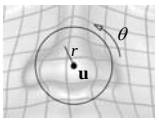
brush points

surface points

Parameterization

- Measuring distortion

$$\gamma(\mathbf{u}) = \int_\theta \left(\frac{\partial^2}{\partial r^2} X_{\mathbf{u}}(\theta, r) \right)^2 d\theta$$



– Integrates squared curvature using local polar re-parameterization

$$X_{\mathbf{u}}(\theta, r) = X\left(\mathbf{u} + r \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} \right)$$

Parameterization

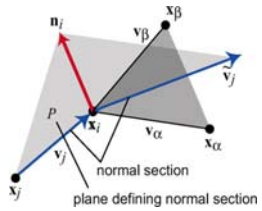
- Discrete formulation:

$$\tilde{C}(U) = \sum_{j \in M} (\mathbf{p}_j - \mathbf{u}_j)^2 + \varepsilon \sum_{i=1}^{n} \sum_{j \in N_i} \left(\frac{\partial U(\mathbf{x}_i)}{\partial \mathbf{v}_j} - \frac{\partial U(\mathbf{x}_i)}{\partial \tilde{\mathbf{v}}_j} \right)^2$$

– Approximation: mapping is piecewise linear

Parameterization

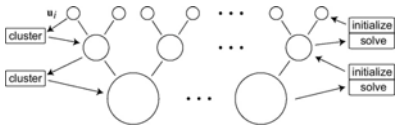
- Directional derivatives as extension of divided differences based on k-nearest neighbors



Parameterization

- HIerarchical solver for efficient computation of resulting sparse linear least squares problem

$$\tilde{C}(U) = \sum_j \left(\mathbf{b}_j - \sum_{i=1}^{n} a_{j,i} \mathbf{u}_i \right)^2 = \|\mathbf{b} - A\mathbf{u}\|^2$$



Reconstruction

- Parameterized scattered data approximation

fitting functions

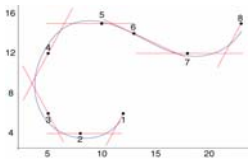
weight functions

$$X(\mathbf{u}) = \frac{\sum_i \Phi_i(\mathbf{u}) r_i(\mathbf{u})}{\sum_i r_i(\mathbf{u})}$$

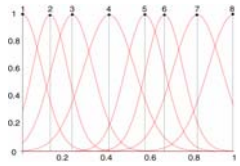
normalization

- Fitting functions
  - Compute local fitting functions using local parameterizations
  - Map to global parameterization using global parameter coordinates of neighboring points

Reconstruction



reconstruction with linear fitting functions



weight functions in parameter space

Reconstruction

- Reconstruction with linear fitting functions is equivalent to surface splatting!
  - Use the surface splatting renderer to reconstruct our surface function (see chapter on rendering)
- Provides:
  - Fast evaluation
  - Anti-aliasing (Band-limit the weight functions before sampling using Gaussian low-pass filter)
  - Distortions of splats due to parameterization can be computed efficiently using local affine mappings

Sampling

- Different sampling strategies:
  - Resample the brush, i.e., sample at the original surface points
  - Resample the surface, i.e., sample at the brush points
  - Adaptive resampling, i.e., sample at surface or brush points depending on the respective sampling density
- Dynamic sampling: see following chapter (Mark Pauly)

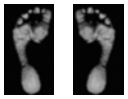
Editing Operators

- Painting
  - Texture, material properties, transparency

Editing Operators

- Sculpting
  - Carving, normal displacement

texture map

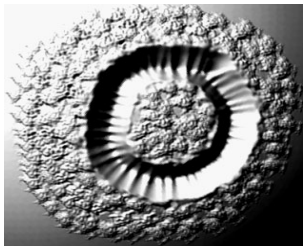


displacement maps

carved and texture mapped point-sampled surface

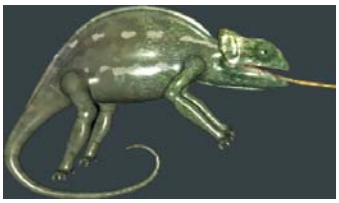
Editing Operators

- Engraving surface detail



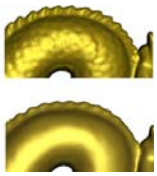
Editing Operators

- Filtering appearance and geometry



Editing Operators

- Filtering appearance and geometry
  - Scalar attributes, geometry



Advanced Processing

- Multiscale feature extraction

Summary

- Pointshop3D provides a versatile platform for research in point based graphics
- Uses points (3-dimensional pixels) as a graphics primitive
- Generalizes 2D image editing tools to point sampled geometry
- A variety of plug-ins for model cleaning, filtering, watermarking etc.
- Version 2 supports advanced modeling operations. More to come…

Reference

- Zwicker, Pauly, Knoll, Gross: *Pointshop3D: An interactive system for Point-based Surface Editing*, SIGGRAPH 2002

pointshop

- check
  http://graphics.ethz.ch/pointshop3d/

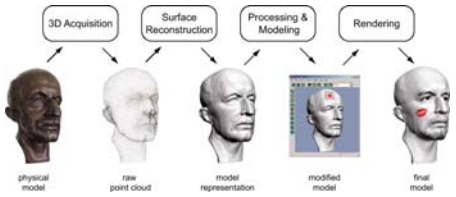
SIGGRAPH2004

Shape Modeling

Mark Pauly Stanford University

Motivation

- 3D content creation pipeline



Motivation

- Surface representations
  - Explicit surfaces (B-reps)
    - Polygonal meshes
    - Subdivision surfaces
    - NURBS

    → ☺ - Efficient rendering, ☺ - Sharp features, ☺ - Intuitive editing
  - Implicit surfaces
    - Level sets
    - Radial basis functions
    - Algebraic surfaces

    → ☺ - Boolean operations, ☺ - Changes of topology, ☺ - Extreme deformations

Motivation

- Surface representations
  - Explicit surfaces (B-reps)
    - Polygonal meshes
    - Subdivision surfaces
    - NURBS

    → ⊗ - Boolean operations, ⊗ - Changes of topology, ⊗ - Extreme deformations
  - Implicit surfaces
    - Level sets
    - Radial basis functions
    - Algebraic surfaces

    → ⊗ - Efficient rendering, ⊗ - Sharp features, ⊗ - Intuitive editing

Motivation

- Surface representations
  - Explicit surfaces (B-reps)
    - Polygonal meshes
    - Subdivision surfaces
    - NURBS
  - Implicit surfaces
    - Level sets
    - Radial basis functions
    - Algebraic surfaces

  → Hybrid Representation
    - Explicit cloud of point samples
    - Implicit dynamic surface model

Motivation

- Point cloud representation
  - Minimal consistency requirements for extreme deformations (dynamic re-sampling)
  - Fast inside/outside classification for boolean operations and collision detection
  - Explicit modeling and rendering of sharp feature curves
  - Integrated, intuitive editing of shape and appearance

Interactive Modeling

- Interactive design and editing of point-sampled models
  - Shape Modeling
    - Boolean operations
    - Free-form deformation
  - Appearance Modeling
    - Painting & texturing
    - Embossing & engraving

Surface Model

- Goal: Define continuous surface from a set of discrete point samples

discrete set of point samples $P = \{ p_i, c_i, m_i, \dots \}$

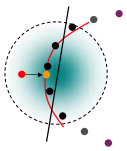
continuous surface S interpolating or approximating P

Surface Model

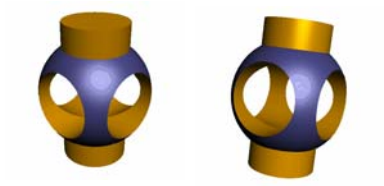
- Moving least squares (MLS) approximation (Levin, Alexa et al.)
  - Surface defined as stationary set of projection operator Ψ_P ⇨ implicit surface model

$$S_P = \left\{ x \in \mathbf{R}^3 \,\middle|\, \Psi_P(x) = x \right\}$$

  - Weighted least squares optimization
    - Gaussian kernel function
      - local, smooth
      - mesh-less, adaptive



Boolean Operations



Boolean Operations

- Create new shapes by combining existing models using union, intersection, or difference operations
- Powerful and flexible editing paradigm mostly used in industrial design applications (CAD/CAM)

Boolean Operations

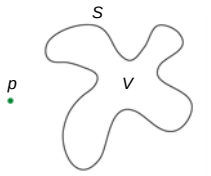
- Easily performed on implicit representations
  - Requires simple computations on the distance function
- Difficult for parametric surfaces
  - Requires surface-surface intersection
- Topological complexity of resulting surface depends on geometric complexity of input models

Boolean Operations

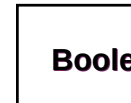
- Point-Sampled Geometry
  - Classification
    - Inside-outside test using signed distance function induced by MLS projection
  - Sampling
    - Compute exact intersection of two MLS surfaces to sample the intersection curve
  - Rendering
    - Accurate depiction of sharp corners and creases using point-based rendering

Boolean Operations

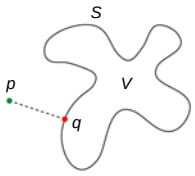
- Classification:
  - given a smooth, closed surface *S* and point *p*. Is *p* inside or outside of the volume *V* bounded by *S*?



Boolean Operations

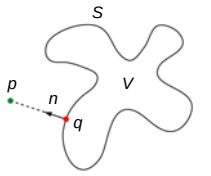


- Classification:
  - given a smooth, closed surface *S* and point *p*. Is *p* inside or outside of the volume *V* bounded by *S*?
  1. find closest point *q* on *S*



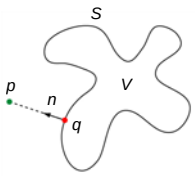
Boolean Operations

- Classification:
  - given a smooth, closed surface *S* and point *p*. Is *p* inside or outside of the volume *V* bounded by *S*?
  1. find closest point *q* on *S*
  2. $d=(p-q)\cdot n$ defines signed distance of *p* to *S*



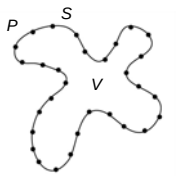
Boolean Operations

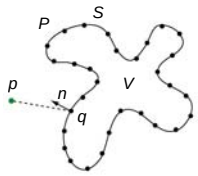
- Classification:
  - given a smooth, closed surface *S* and point *p*. Is *p* inside or outside of the volume *V* bounded by *S*?
  1. find closest point *q* on *S*
  2. $d=(p-q)\cdot n$ defines signed distance of *p* to *S*
  3. classify *p* as
     - inside *V*, if $d < 0$
     - outside *V*, if $d > 0$

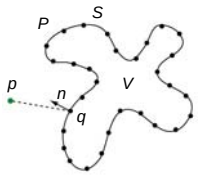


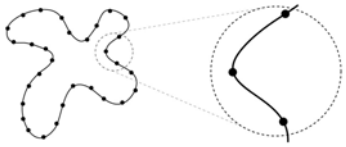
Boolean Operations

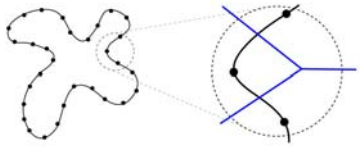
- Classification:
  - represent smooth surface *S* by point cloud *P*

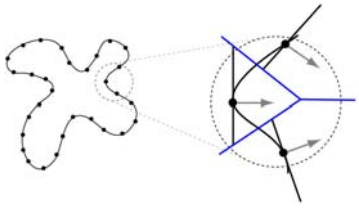


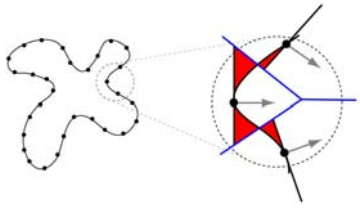












Boolean Operations

- Classification:
  - represent smooth surface *S* by point cloud *P*

1. find closest point *q* in *P*

Boolean Operations

- Classification:
  - represent smooth surface *S* by point cloud *P*

1. find closest point *q* in *P*
2. classify *p* as
   - inside *V*, if $(p-q)\cdot n < 0$
   - outside *V*, if $(p-q)\cdot n > 0$

Boolean Operations

- Classification:
  - piecewise constant surface approximation leads to false classification close to the surface

Boolean Operations

- Classification:
  - piecewise constant surface approximation leads to false classification close to the surface

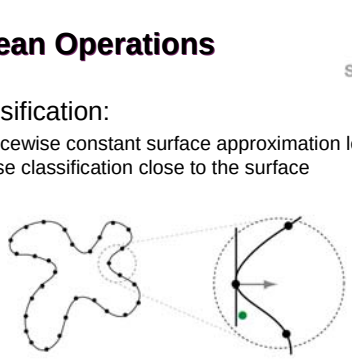
Boolean Operations

- Classification:
  - piecewise constant surface approximation leads to false classification close to the surface

Boolean Operations

- Classification:
  - piecewise constant surface approximation leads to false classification close to the surface

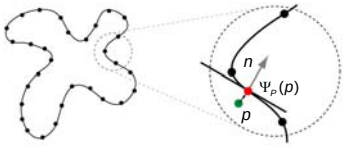
Boolean Operations



- Classification:
  - piecewise constant surface approximation leads to false classification close to the surface

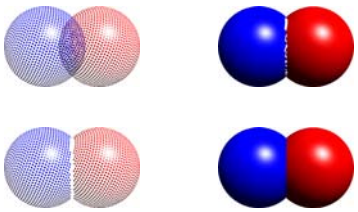
Boolean Operations

- Classification:
  - use MLS projection of *p* for correct classification



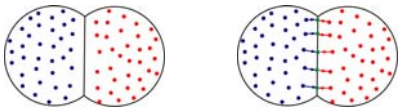
Boolean Operations

- Sampling the intersection curve



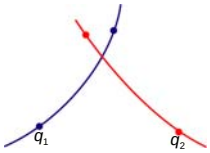
Boolean Operations

- Newton scheme:
  1. identify pairs of closest points



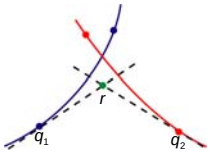
Boolean Operations

- Newton scheme:
  1. identify pairs of closest points



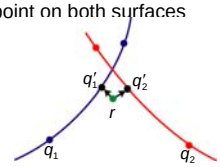
Boolean Operations

- Newton scheme:
  1. identify pairs of closest points
  2. compute closest point on intersection of tangent spaces



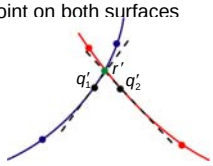
Boolean Operations

- Newton scheme:
  1. identify pairs of closest points
  2. compute closest point on intersection of tangent spaces
  3. re-project point on both surfaces



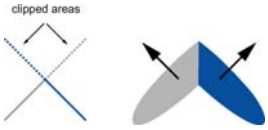
Boolean Operations

- Newton scheme:
  1. identify pairs of closest points
  2. compute closest point on intersection of tangent spaces
  3. re-project point on both surfaces
  4. iterate



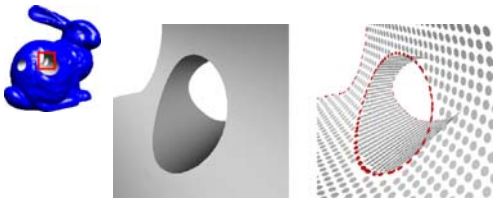
Boolean Operations

- Rendering sharp creases
  - represent points on intersection curve with two surfels that mutually clip each other



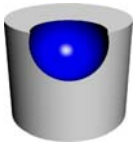
Boolean Operations

- Rendering sharp creases



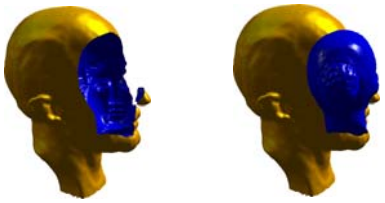
Boolean Operations

- Rendering sharp creases
  - easily extended to handle corners by allowing multiple clipping

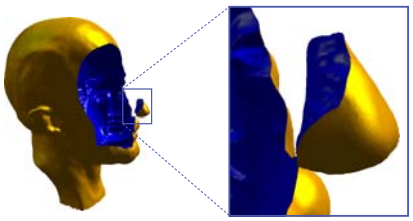


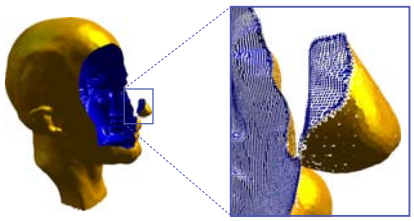
Boolean Operations

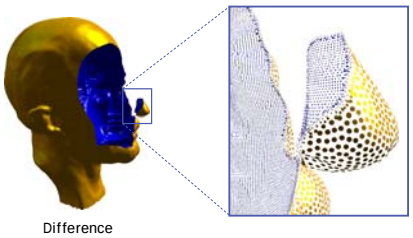
- Rendering sharp creases

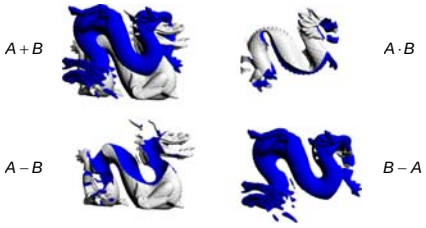


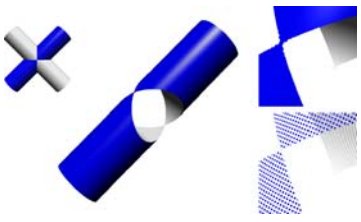
Difference Union











Boolean Operations

- Rendering sharp creases

Boolean Operations

- Rendering sharp creases

Boolean Operations

- Rendering sharp creases

Boolean Operations

- Boolean operations can create intricate shapes with complex topology

---

Boolean Operations

- Singularities lead to numerical instabilities (intersection of almost parallel planes)

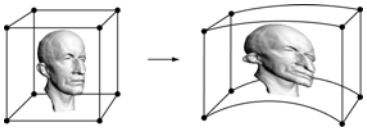
Particle-based Blending

- Boolean operations create sharp intersection curves
- Particle simulation to create smooth transition
  - Repelling force to control particle distribution
  - Normal potentials to control particle orientation

Free-form Deformation

Free-form Deformation

- Smooth deformation field $F:\mathbf{R}^3 \rightarrow \mathbf{R}^3$ that warps 3D space
- Can be applied directly to point samples



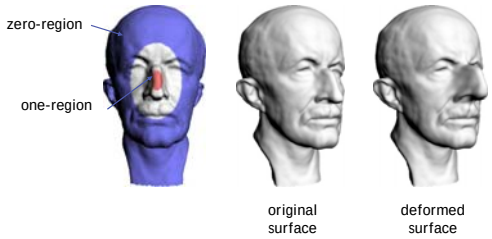
Free-form Deformation

- How to define the deformation field?
  - ⇨ Painting metaphor
- How to detect and handle self-intersections?
  - ⇨ Point-based collision detection, boolean union, particle-based blending
- How the handle strong distortions?
  - ⇨ Dynamic re-sampling

Free-form Deformation

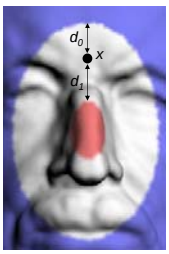
- Intuitive editing paradigm using painting metaphor
  - Define rigid surface part (zero-region) and handle (one-region) using interactive painting tool
  - Displace handle using combination of translation and rotation
  - Create smooth blend towards zero-region

Free-form Deformation



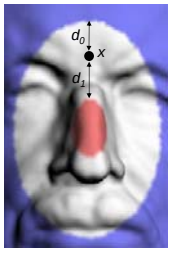
Free-form Deformation

- Definition of deformation field:
  - Continuous scale parameter t_x
    - $t_x = \beta(d_0 / (d_0 + d_1))$
    - d_0: distance of x to zero-region
    - d_1: distance of x to one-region
  - Blending function
    - β: $[0,1] \rightarrow [0,1]$
    - $\beta \in C^0$, $\beta(0) = 0$, $\beta(1) = 1$
  - $t_x = 0$ if x in zero-region
  - $t_x = 1$ if x in one-region



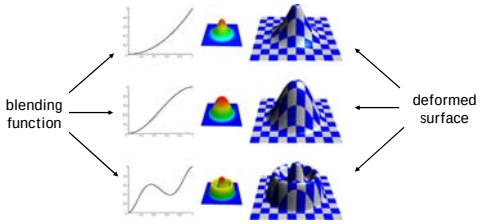
Free-form Deformation

- Definition of deformation field:
  - Deformation function
    - $F(x) = F_T(x) + F_R(x)$
  - Translation
    - $F_T(x) = x + t_x \cdot v$
  - Rotation
    - $F_R(x) = M(t_x) \cdot x$



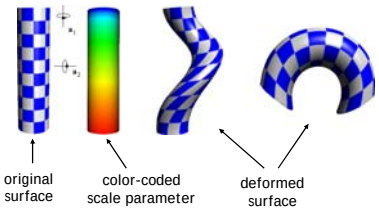
Free-form Deformation

- Translation for three different blending functions



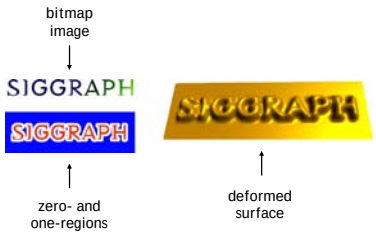
Free-form Deformation

- Rotational deformation along two different rotation axes



Free-form Deformation

- Embossing effect

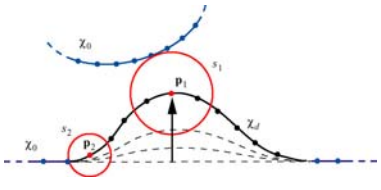


Collision Detection

- Deformations can lead to self-intersections
- Apply boolean inside/outside classification to detect collisions
- Restricted to collisions between deformable region and zero-region to ensure efficient computations

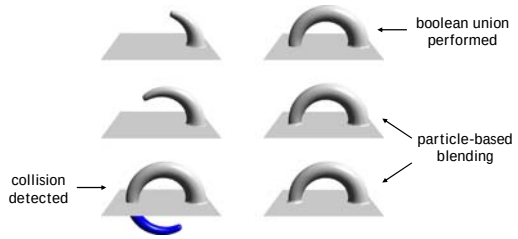
Collision Detection

- Exploiting temporal coherence

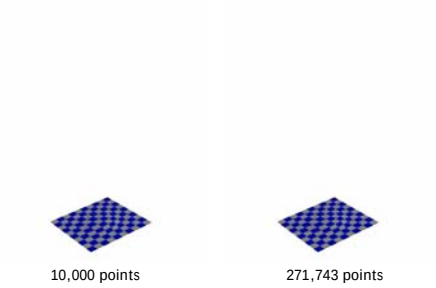


Collision Detection

- Interactive modeling session



Dynamic Sampling

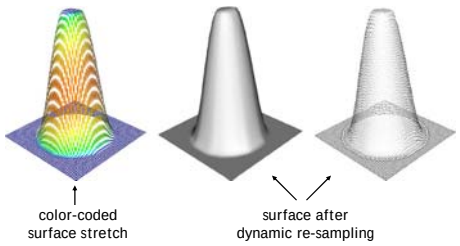


Dynamic Sampling

- Large model deformations can lead to strong surface distortions
- Requires adaptation of the sampling density
- Dynamic insertion and deletion of point samples

Dynamic Sampling

- Surface distortion varies locally

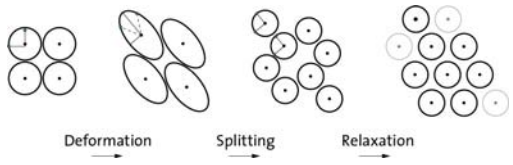


Dynamic Sampling

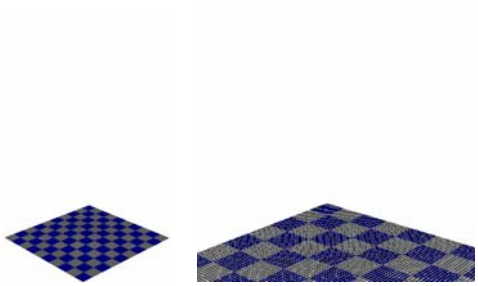
1. Measure local surface stretch from first fundamental form
2. Split samples that exceed stretch threshold
3. Regularize distribution by relaxation
4. Interpolate scalar attributes

Dynamic Sampling

- 2D illustration

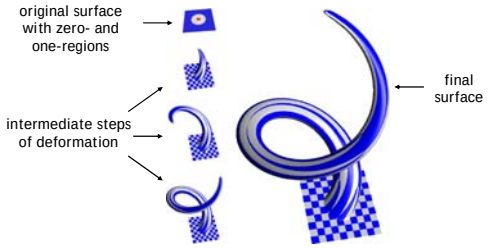


Dynamic Sampling



Free-form Deformation

- Interactive modeling session with dynamic sampling



Results

- 3D shape modeling functionality has been integrated into Pointshop3D to create a complete system for point-based shape and appearance modeling
  - Boolean operations
  - Free-form deformation
  - Painting & texturing
  - Sculpting
  - Filtering
  - Etc.

Results

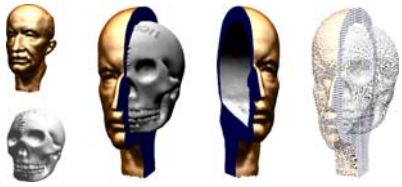
- Ab-initio design of an Octopus
  - Free-form deformation with dynamic sampling from 69,706 to 295,222 points

Results

- Modeling with synthetic and scanned data
  - Combination of free-form deformation with collision detection, boolean operations, particle-based blending, embossing and texturing

Results

- Boolean operations on scanned data
  - Irregular sampling pattern, low resolution models



Results

- Interactive modeling with scanned data
  - noise removal, free-form deformation, cut-and-paste editing, interactive texture mapping

Conclusion

- Points are a versatile shape modeling primitive
  - Combines advantages of implicit and parametric surfaces
  - Integrates boolean operations and free-form deformation
  - Dynamic restructuring
  - Time and space efficient implementations

Conclusion

- Complete and versatile point-based 3D shape and appearance modeling system
  - Directly applicable to scanned data
  - Suitable for low-cost 3D content creation and rapid proto-typing

References

- Pauly: Point Primitives for Interactive Modeling and Processing of 3D Geometry, PhD Thesis, ETH Zurich, 2003
- Pauly, Keiser, Kobbelt, Gross: Shape Modeling with Point-sampled Geometry, SIGGRAPH 03
- Pauly, Kobbelt, Gross: Multiresolution Modeling with Point-sampled Geometry, ETH Technical Report, 2002
- Zwicker, Pauly, Knoll, Gross: Pointshop3D: An Interactive System for Point-based Surface Editing, SIGGRAPH 02
- Adams, Dutre: Boolean Operations on Surfel-Bounded Solids, SIGGRAPH 03
- Szeliski, Tonnesen: Surface Modeling with Oriented Particle Systems, SIGGRAPH 92
- www.pointshop3d.com

SECTION 4

SUPPLEMENTAL MATERIAL

Surface Representation with Point Samples. Marc Alexa

Shape Modeling with Points. Mark Pauly

Supplemental Notes on Surface Representation with Point Samples

Marc Alexa
Discrete Geometric Modeling Group
Department of Computer Science
Darmstadt U of Technology
Fraunhoferstr. 5, 64283 Darmstadt, Germany
phone: +49.6151.155.674, fax: +49.6151.155.669
Email: alexa@informatik.tu-darmstadt.de

April 26, 2004

This document contains supplemental notes on "Surface Representation with Point Samples" as part of the SIGGRAPH course on "Point-based Computer Graphics". It should be used together with the slide set, which contains helpful illustrations.

1 Introduction

Point sets are emerging as a surface representation. The particular appeal of points sets is their generality: every shape can be represented by a set of points on its boundary, where the degree of accuracy typically depends only on the number of points. Point sets do not have a fixed continuity class or are limited to a certain topology as many other surface representations.

To define a manifold from the set of points, the inherent spatial interrelation among the points has to be exploited as implicit connectivity information. A mathematical definition or algorithm attaches a topology and a geometric shape to the set of points. This is non-trivial since it is unclear what spacing of points represents connected respectively disconnected pieces of the surface.

Representing the surface with points is slightly different from the problem of reconstructing a surface from point samples: The basic idea of representation is to use the points as the main source of information about the shape. Efficient algorithms are applied to the points to determine if a certain point in space is inside or outside of the shape, how far it is from the surface, to

project this point onto the surface, or to intersect other primitives with the surface. In contrast, reconstruction is typically concerned with converting the point set into another representation, where these algorithmic goals are easier to perform.

A consequence of this view is that we are interested in local algorithms. Only local algorithms have the premise to be efficient when used to perform certain local operations on very large point sets. Specifically, we'd like to avoid the construction of a global connectivity structure among the points. We admit that doing this can lead to very good reconstruction results, however, it also has some drawbacks, which we won't discuss here. Despite the lack of global structure, we wish that putting all the local computations together would result in a smooth and (where reasonable) manifold surface.

2 Notation & Terms

We assume that the points $\mathscr{P} = \{p_i \in \mathbb{R}^3\}, i \in \{1,\ldots,N\}$, are sampled from an unknown surface $\mathscr{S}$, and that they might contain some noise due to the imperfect sampling process. Some sampling processes additionally provide normal information in each point, which we assume to be represented as $\mathscr{N} = \{\mathbf{n}_i \in \mathbb{R}^3, \|\mathbf{n}_i\| = 1\}$.

We assume that data is *irregular*, i.e. that the points are not sampled from a regular lattice in space.

Our goal is to define computational methods for the interrogation or manipulation of a point $\mathbf{x} \in \mathbb{R}^3$. These computational tools indirectly define a surface $\hat{\mathscr{S}}$ from the points $\mathscr{P}$ (and possibly the normals $\mathscr{N}$). We understand the term *locality* as the extent of space or the number of points that are necessary to perform the computations for $\mathbf{x}$. A *global* method will potentially require all points in $\mathscr{P}$ to perform the computations.

The reconstructed surface is said to be *interpolating* if $\mathscr{P} \in \hat{\mathscr{S}}$, otherwise it is approximating. We will almost exclusively look at the case of approximation. Approximating the points takes into account that the surface is assumed to be not too wiggly and that the points contain some noise. An approximation allows smoothing this noise and providing a reasonably behaved surface.

Before we approach the general surface representation problem, we'll recall some basic methods for the interpolation or approximation of functional data. For this, we assume that each point $\mathbf{p}_i = (\mathbf{q}_i, f_i)$ is composed of a position $\mathbf{q}_i$ in parameter space ($\mathbb{R}^2$ in our setting) and a value f_i at this position.

3 Interpolation and approximation of functional data

For now, our goal is to determine a function f that interpolates or approximates the given constraints $\mathbf{p}_i = (\mathbf{q}_i, f_i)$, i.e. $\hat{f}(\mathbf{q}_i) \approx f_i$. Defining such a function means to describe an algorithm that computes for every $\mathbf{x} \in \mathbb{R}^d$ a function value $\hat{f}(\mathbf{x})$.

We start with a very simple approach: given $\mathbf{x}$, find the closest location for which a function value is defined, i.e. $\min_j \|\mathbf{q}_j - \mathbf{x}\|$. If the minimum is not unique, choose the one with smallest

index j. Then set $\hat{f}(\mathbf{x})$ to f_j. More formally, we define

$$\hat{f}(\mathbf{x}) = \mathbf{q}_j, 0 < j < i \Rightarrow \|q_j - \mathbf{x}\| < \|q_i = \mathbf{x}\|, i < j < N \Rightarrow \|q_j - \mathbf{x}\| \leq \|q_i = \mathbf{x}\| \tag{1}$$

The result is a function that interpolates the points but is not continuous.

The obvious idea to improve the continuity of $\hat{f}$ is to combine the values of several close points. In general, our approach looks like this:

$$\hat{f}(\mathbf{x}) = \sum_i w_i(\mathbf{x}) f_i, \tag{2}$$

where $w_i(\mathbf{x})$ are weight functions appropriate to combine the values of several points in a location $\mathbf{x}$.

Depending on how the set of 'close' points is identified and how the weight functions are compute based on the set of close points, several methods with different properties can be derived. We will take a close look at the following ideas:

Voronoi techniques Identify the regions for which the location of data point $\mathbf{q}_i$ is closest and exploit the adjacency of these regions.

RBF Attach a (radial) function to each data point that describes how it influences space.

Shepard Collect points in a certain radius and weight them based on distance.

MLS Collect points in a certain radius and weight them so that the resulting function would reproduce a given class of functions.

We will first look at these approaches and then broaden the discussion by introducing (adaptive) regular spatial subdivisions based on the last approach.

3.1 Voronoi techniques

We note that a certain set of locations $\mathbf{x}$ has the property $\|\mathbf{q}_j - \mathbf{x}\| \leq \|q_i - \mathbf{x}\|, j \neq i$, i.e. is closest to $\mathbf{q}_j$. This set of points is the Voronoi cell C_j, i.e. formally

$$C_j = \{\mathbf{x}, \|\mathbf{q}_j - \mathbf{x}\| \leq \|\mathbf{q}_i = \mathbf{x}\|, j \neq i\} \tag{3}$$

Note that two Voronoi cells C_i and C_j might have some points in common (points $\mathbf{x}$ for which $\|\mathbf{q}_i - \mathbf{x}\| = \|\mathbf{q}_j - \mathbf{x}\|$. If $C_i \cap C_j \neq \emptyset$ we call C_i and C_j or, equivalently, u and j *neighbors*.

We can use the concept of Voronoi cells and neighboring data points in different ways to improve the continuity of $\hat{f}$. We will only sketch these ideas as it turns out that computing Voronoi cells is a global problem.

Connecting all neighboring locations $\mathbf{q}_i$ and $\mathbf{q}_j$ defines a geometric graph, the so called Delaunay graph. In most cases all faces of this graph are triangles. Triangulating the remaining faces with 4 or more edges generates the Delaunay triangulation. Let $\{\mathcal{T}_{i,j,k}\}$ be the triangles spanned by $\mathbf{q}_i, \mathbf{q}_j, \mathbf{q}_k$. Note that the signed area of a triangle spanned by $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ is

$$A(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) = \frac{1}{2} \det(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) \tag{4}$$

The weights for defining $\hat{f}$ are barycentric coordinates for the triangle that contains $\mathbf{x}$. To compute those, we need to identify (i,j,k) so that $\mathbf{x} \in \mathscr{T}_{i,j,k}$. To compute this, note that

$$\mathbf{x} \in \mathscr{T}_{i,j,k} \Longleftrightarrow \text{sgn}A(\mathbf{x},\mathbf{q}_i,q_j) = \text{sgn}A(\mathbf{x},\mathbf{q}_j,\mathbf{q}_k) = \text{sgn}A(\mathbf{x},\mathbf{q}_k,\mathbf{q}_i). \tag{5}$$

Then, barycentric coordinates w.r.t. the locations $\mathbf{q}_i,\mathbf{q}_j,\mathbf{q}_k$ are defined as

$$w(\mathbf{x})_i = \frac{A(\mathbf{x},\mathbf{q}_j,q_k)}{A(\mathbf{q}_i,\mathbf{q}_j,\mathbf{q}_k)}, w(\mathbf{x})_j = \frac{A(\mathbf{x},\mathbf{q}_k,q_i)}{A(\mathbf{q}_i,\mathbf{q}_j,\mathbf{q}_k)}, w(\mathbf{x})_k = \frac{A(\mathbf{x},\mathbf{q}_i,q_j)}{A(\mathbf{q}_i,\mathbf{q}_j,\mathbf{q}_k)}. \tag{6}$$

In essence, $\hat{f}$ is defined as the Delaunay triangulation of the locations $\{q_i\}$ lifted to the heights $\{f_i\}$. As such, $\hat{f}$ is a continuous function, with discontinuous derivatives along the edges of the triangulation.

The Voronoi cells can be further exploited to achieve an interpolating function $\hat{f}$ with continuous derivatives. To derive the weights for $\mathbf{x}$, compute the Voronoi cell $C_\mathbf{x}$ for $\mathbf{x}$ if it was added to the set of points. Then weights could be defined as

$$w_i(\mathbf{x}) = \frac{|C_\mathbf{x} \cap C_i|}{|C_\mathbf{x}|}, \tag{7}$$

i.e. as the relative area the cell $C_\mathbf{x}$ shares with the original cells $\{C_i\}$. There are alternative definitions for the weights based on this general concept, leading to additional properties for $\hat{f}$.

3.2 Radial Basis Functions

A basic and very general approach is to model the weight functions as radial functions

$$w_i(\mathbf{x}) = \frac{c_i}{f_i}\theta(\|\mathbf{x}-\mathbf{q}_i\|) \Leftrightarrow w_i(\mathbf{x})\,f_i = c_i\theta(\|\mathbf{x}-\mathbf{q}_i\|), \tag{8}$$

where θ is a function that describes the influence of $\mathbf{q}_i$ on $\mathbf{x}$ based on the distance between the two locations.

In this approach, all weight functions w_i are essentially the same and only differ by a linear factor. Note that the method is already fully defined for a fixed function θ in case we ask for interpolation: Requiring $\hat{f}(\mathbf{q}_j) = f_j$ leads to

$$\hat{f}(\mathbf{q}_j) = \sum_i c_i\theta(\|\mathbf{q}_j-\mathbf{q}_i\|) = f_j, \tag{9}$$

which is, in fact, a system of linear equations:

$$\begin{pmatrix} \theta(\|\mathbf{q}_0-\mathbf{q}_0\|) & \theta(\|\mathbf{q}_0-\mathbf{q}_1\|) & \theta(\|\mathbf{q}_0-\mathbf{q}_2\|) & \cdots \\ \theta(\|\mathbf{q}_1-\mathbf{q}_0\|) & \theta(\|\mathbf{q}_1-\mathbf{q}_1\|) & \theta(\|\mathbf{q}_1-\mathbf{q}_2\|) & \cdots \\ \theta(\|\mathbf{q}_2-\mathbf{q}_0\|) & \theta(\|\mathbf{q}_2-\mathbf{q}_1\|) & \theta(\|\mathbf{q}_2-\mathbf{q}_2\|) & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \end{pmatrix} \tag{10}$$

So, before we are able to compute the weights we first need to solve this linear system. This requires that the system has a solution, which means the data points allow being interpolated

with the given functions. As the matrix depends only on values of θ, solvability obviously depends on the choice of θ.

Standard choices for the radial function θ are $\theta(\delta) = \delta^{-\nu}$ or the Gaussian $\theta(\delta) = \exp(\delta^2/h^2)$. However, another concern makes these functions impractical: Each point influences every other point, making the approach global. This can also be recognized from the fact that the a dense linear system has to be solved before $\hat{f}$ could be evaluated in any point.

In an attempt to make the solution local we should use locally supported radial functions. This means, we can choose a distance parameter ε. If two points are further apart then ε the function θ attached to either of them vanishes in the other point, i.e. $\delta > \varepsilon \Rightarrow \theta(\delta) = 0$.

Popular choices with good properties are Wendland's radial functions [Wen95], because they consist of polynomial pieces with low degree (i.e. they are easy to compute) and lead to solvable linear systems. The particular function to be used depends on the space in which the locations $\mathbf{q}_i$ live.

Using these locally supported functions leads to sparse linear systems, which can be solved in almost linear time. Nevertheless, strictly speaking this is a global solution, as the inverse of a sparse matrix is not necessarily sparse. Practically speaking this means moving one point could potentially influence points further away then ε by a cascade of effects on other points.

On the other hand, the sparse linear system has to be solved only once. This defines the linear factors $\{c_i\}$, which in turn define the weight functions $w_i(\mathbf{x})$. Evaluating $\hat{f}(\mathbf{x})$ is typically very cheap, as θ has to be evaluated only for few close points.

3.3 Least squares fitting

I assume that most readers are familiar with the idea of least squares fitting a polynomial to given data. Here, we will rediscover this method in our setting, by introducing the concept of a precision set. I hope this presentation eases the understanding of the following techniques.

As before, we represent $\hat{f}$ at $\mathbf{x}$ as $\sum_i w_i(\mathbf{x}) f_i$. We ask that $\hat{f}$ has a certain precision, which is described by a precision set $\mathscr{G}$: If the pairs $(\mathbf{q}_i, f_i)$ happen to be sampled from a function contained in the precision set (say $g \in \mathscr{G}$), then we wish that $\hat{f}$ results to be exactly that function. We can formalize this requirement as follows: For every $g \in \mathscr{G}$ the weight functions have to satisfy

$$g(\mathbf{x}) = \sum_i w_i(\mathbf{x}) g(\mathbf{q}_i). \tag{11}$$

As a more concrete example, consider the precision set of quadratic polynomials $g(\mathbf{x}) = a + \mathbf{b}^\mathsf{T}\mathbf{x} + \mathbf{x}^\mathsf{T}\mathbf{C}\mathbf{x}$. Look at the following system of equations

$$\begin{aligned} 1 &= \sum_i w_i(\mathbf{x}) 1 \\ x_0 &= \sum_i w_i(\mathbf{x}) q_{i_0} \\ &\vdots \\ x_0^2 &= \sum_i w_i(\mathbf{x}) q_{i_0}^2 \\ &\vdots \end{aligned} \tag{12}$$

and note that the set of linear combinations of these equations

$$a + \mathbf{b}^\mathsf{T}\mathbf{x} + \mathbf{x}^\mathsf{T}\mathbf{C}\mathbf{x} = \sum_i w_i(\mathbf{x}) \left(a + \mathbf{b}^\mathsf{T}\mathbf{q}_i + \mathbf{q}_i^\mathsf{T}\mathbf{C}\mathbf{q}_i \right). \tag{13}$$

is, in fact, the requirement of reproducing any function from the precision set of quadratic polynomials.

We can write the system of equations in matrix form as

$$\mathbf{Q}\mathbf{w}(\mathbf{x}) = \mathbf{z}. \tag{14}$$

Typically, we will have more points than dimensions in the space of polynomials, i.e. the system is underdetermined. We need to restrict the weights further. A common way to do this would be to ask that sum of squared weights is minimal, i.e.

$$\min_{\{w_i(\mathbf{x})\}} \sum_i w_i(\mathbf{x}) = \min_{\mathbf{w}(\mathbf{x})} \mathbf{w}(\mathbf{x})^\mathsf{T}\mathbf{w}(\mathbf{x}). \tag{15}$$

How could we find this minimum, subject to the linear constraints given in Eq. 14? Assume we know the solution vector $\mathbf{w}(\mathbf{x})$. Now look at the polynomial $(a, b_0, \ldots)\mathbf{Q}\mathbf{w}(\mathbf{x})$. We can certainly choose the polynomial coefficients $(a, b_0, \ldots)$ so that this polynomial attains a minimum or a maximum for the given weight vector $\mathbf{w}(\mathbf{x})$. So instead of minimizing only squared weights, we try to minimize

$$\mathbf{w}(\mathbf{x})^\mathsf{T}\mathbf{w}(\mathbf{x}) - (a, b_0, \ldots)\mathbf{Q}\mathbf{w}(\mathbf{x}), \tag{16}$$

where we have the polynomial coefficients as additional degrees of freedom. This approach helps to include the linear constraints in the minimization, at the cost of additional variables to solve for. A necessary condition for the minimum is that all partial derivatives are identical zero. Taking all partial derivatives w.r.t. the weights and setting to zero leads to

$$\mathbf{w}(\mathbf{x})^\mathsf{T} - (a, b_0, \ldots)\mathbf{Q} = 0 \Longleftrightarrow \mathbf{w}(\mathbf{x}) = \mathbf{Q}^\mathsf{T}(a, b_0, \ldots)^\mathsf{T} \tag{17}$$

Using that in Eq. 14 yields

$$\mathbf{Q}\mathbf{Q}^\mathsf{T}(a, b_0, \ldots)^\mathsf{T} = \mathbf{z}, \tag{18}$$

which is identical to the normal equation for least squares fitting a polynomial and also shows that the solution is independent of the location $\mathbf{x}$. Once the polynomial coefficients are determined one could indeed solve for the weights at $\mathbf{x}$, however, in this case it is easier to compute $\hat{f}$ using the representation as a polynomial.

Notice that the solution we have presented works for any precision set that could be represented as finite linear space.

3.4 Moving Least Squares

We will follow the basic ideas of the last section. The only modification is that we localize weights. We do this by incorporating a separation measure into the minimization of squared weights:

$$\min_{\{w_i(\mathbf{x})\}} \sum_i w_i^2(\mathbf{x})\eta(\|\mathbf{q}_i - \mathbf{x}\|) = \min_{\mathbf{w}(\mathbf{x})} \mathbf{w}(\mathbf{x})^\mathsf{T}\mathbf{E}(\mathbf{x})\mathbf{w}(\mathbf{x}) \tag{19}$$

The separation measure $\eta(\|\mathbf{q}_i - \mathbf{x}\|)$ penalizes the influence of points at $\mathbf{q}_i$ far away from $\mathbf{x}$, i.e. the function increases with the distance between $\mathbf{q}_i$ and $\mathbf{x}$.

The solution to this constrained minimization is similar to the uniform situation. Now one has to solve

$$\mathbf{w}(\mathbf{x})^\mathsf{T}\mathbf{E}(\mathbf{x}) - (a, b_0, \ldots)\mathbf{Q} = 0 \tag{20}$$

which leads to

$$\mathbf{w}(\mathbf{x}) = \mathbf{E}(\mathbf{x})^{-1}\mathbf{Q}^\mathsf{T}(a, b_0, \ldots)^\mathsf{T}. \tag{21}$$

This can be inserted into the constraint equation $\mathbf{Q}\mathbf{w}(\mathbf{x}) = \mathbf{z}$ to get the polynomial coefficients:

$$\left(\mathbf{Q}\mathbf{E}(\mathbf{x})^{-1}\mathbf{Q}^\mathsf{T}\right)(a, b_0, \ldots)^\mathsf{T} = \mathbf{z} \tag{22}$$

We see that the polynomial coefficients result from a weighted least squares system. The weighting comes from the η^{-1}, which we call θ for convenience. It depends on the location $\mathbf{x}$, because η depends on $\mathbf{x}$. The resulting approach could also be interpreted like this: In each location $\mathbf{x}$ determine a locally weighted least squares fitting polynomial and use the value of this polynomial at $\mathbf{x}$ (a 'moving' least squares approximation [LS98, Lev98]). In this interpretation it seems the approximating values of $\hat{f}(\mathbf{x})$ are computed independently for different locations, so it might not be immediately clear that $\hat{f}$ is a continuously differentiable function. Our derivation of $\hat{f}$, however, reveals that it is, if η (or, better, $\mathbf{E}(\mathbf{x})$) is continuously differentiable.

If θ is locally supported (i.e. vanishes for large distances between $\mathbf{x}$ and $\mathbf{q}_i$) the computations for $\mathbf{x}$ are also local, as they depend only on the data points that are within the support. For $\eta(0) = 0$ (i.e. $\theta(0) = \infty$) the resulting function $\hat{f}$ interpolates the points. Notice that the statements about the continuity of $\hat{f}$ hold also for the case of local support and/or interpolation.

The coefficients of the polynomial could be used to find the weights as

$$\mathbf{w}(\mathbf{x}) = \mathbf{E}(\mathbf{x})^{-1}\mathbf{Q}\left(\mathbf{Q}\mathbf{E}(\mathbf{x})^{-1}\mathbf{Q}^\mathsf{T}\right)^{-1}\mathbf{z}. \tag{23}$$

Now we take a closer look at the special case of asking only for constant precision. Then, $\mathbf{Q}$ is a row vector containing only ones and $\mathbf{z} = 1$. Then $\mathbf{E}(\mathbf{x})^{-1}\mathbf{Q}$ is a row vector containing the terms $\theta(\|\mathbf{q}_i - \mathbf{x}\|)$ and $\mathbf{Q}\mathbf{E}(\mathbf{x})^{-1}\mathbf{Q}^\mathsf{T}$ is the sum of these terms. This means we get the following weights for location $\mathbf{x}$ when asking only for constant precision:

$$w_j(\mathbf{x}) = \frac{\theta(\|\mathbf{q}_j - \mathbf{x}\|)}{\sum_i \theta(\|\mathbf{q}_i - \mathbf{x}\|)} \tag{24}$$

This type of weight is commonly called a *Partition of Unity*, because the weights sum up to one everywhere. Using $\theta(\delta) = \delta^{-r}, r > 0$ we rediscover a particular and well-known instance of Partition of Unity: Shepard's interpolation method [She68, FN80].

3.5 Adaptive spatial subdivisions

The MLS method associates a local approximation with each location in the parameter domain. For the evaluation of $\hat{f}(\mathbf{x})$ this means one has to compute the polynomial coefficients (or, the weights w_i) for each location $\mathbf{x}$. An approach to reduce the amount of computations is to divide

the parameter space into cells and use only one approximating function per cell. Of course, to achieve an overall continuous approximation we make the cells slightly overlapping and blend the local approximations in the areas of overlap. For this blending, we use a Partition of Unity, however, now derived from the cells rather than the points. The benefit of computing the local approximations and the weights for the cells is simply that we can construct the cells so that we have much less cells than points. Because of blend function that sum up to one are a natural choice this approach is called Partition of Unity approach in some other communities [BM97, GS00, GS02a, GS02b].

To be more concrete, let the cells be denoted Ω_i. Together they form an overlapping covering of the parameter domain, i.e. the intersection of neighboring cells has non-zero area. With each cell Ω_i we associate a weight function w_i and a local (polynomial) approximation $\hat{f}_i$ of the points within Ω_i. Then the global approximating function is given as

$$\hat{f}(\mathbf{x}) = \sum_i w_i(\mathbf{x})\,\hat{f}_i \tag{25}$$

How do we compute the local approximations $\hat{f}_i$ and the weights $w_i(\mathbf{x})$? The local approximations could be simply low degree polynomials derived from least squares fitting to the points contained in Ω_i. The weights could be computed by starting from a blend function θ_i that has support only inside the cell Ω_i, i.e. vanishes outside the cell and then deriving $w_j(\mathbf{x}) = \theta(\mathbf{x})/\sum_i \theta_i(\mathbf{x})$, very similar to the MLS situation.

A good choice of θ depends on the shape of the cell. For spherical cells with radius R_i centered at $\mathbf{c}_i$ the quadratic B-spline $b(t)$ yields a reasonable low degree approximation of a Gaussian as $\theta_i(\mathbf{x}) = b(\|\mathbf{c}_i - \mathbf{x}\|/R_i)$. For box-shaped cells, the quadratic B-spline could be used in a tensor product. For example, given a box with center $\mathbf{c}_i$ and edge lengths $e_0, e_1, \ldots$ this yields $\theta_i(\mathbf{x}) = b(2\|c_{i_0} - x_0\|/e_0) \cdot b(2\|c_{i_1} - x_0\|/e_1) \ldots$. Of course, other choices are possible.

Having these ingredients together, it remains to find a reasonable decomposition of the parameter space into cells. here, reasonable means that each cell contains not too many points, so that the local approximation is accurate, but also not too few, because than the number of cells is large (and, the local approximation could become ambiguous). For irregular data this could be very difficult.

A much better strategy is to use adaptive subdivisions: We start by subdividing the parameter domain into few cells. For example, we divide the space into half along each dimension and then slightly increase the size of the cells in all dimensions to generate the necessary overlap. For example, in the plane this generates a 2 by 2 grid of rectangular cells. In each of the cells we compute a local (low degree polynomial) approximation. Then we compute the max-norm error of these local approximations. If this error exceeds a certain threshold, this cell is subdivided again, and the procedure is applied to the sub-cells. In the plane, the result is a quadtree (though with slightly larger cells to provide the overlap), where the error of the local approximation is bounded by a given threshold in each cell. Once this is achieved, weight functions are determined for the cells, and $\hat{f}$ could be evaluated.

In fact, we don't need to build all approximations before starting to evaluate $\hat{f}$ – this can be done on the fly only where needed: For the evaluation of $\hat{f}(\mathbf{x})$ we determine all cells containing $\mathbf{x}$ and compute local approximations. Then the error is determined and the cells are subdivided if necessary. This procedure is repeated for the sub-cells and so on, until the error is bounded.

For the cells with bounded error that contain $\mathbf{x}$, $\hat{f}_i(\mathbf{x})$ and $w_i(\mathbf{x})$ and determined to assemble $\hat{f}$. A pseudo-code for the evaluation of $\hat{f}_i$ is given below. For evaluating $\hat{f}$ this function has to be called with a cell that contains all points in $\mathscr{P}$:

```
Evaluate f̂(Ω_i, x, ε)
    if (f̂_i is not created yet) then
        Create f̂_i and compute e = max_{q_j ∈ Ω_i} ‖ f_j − f̂_i(q_i) ‖;
        if (e > ε) then
            Create subcells {ω_k};
            for each ω_k
                Evaluate f̂(ω_k, x, ε);
            end for
        else (error is below bound)
        f̂(x) = f̂(x) + w_i(x) * f̂_i(x);
        σ_w = σ_w + w_i(x);
    end if
f̂(x) = f̂(x)/σ_w;
```

This adaptive approach has several nice properties: The approximation error is bounded, because the max-norm error of $\hat{f}$ cannot be larger than the max-norm error in any of the cells. This is a result of the weight functions summing up to one. All computations are purely local, resulting a an almost linear run time behavior. In fact, for practical distributions of points the algorithm scales linearly with the number of points. Another consequence of data locality is that out-of-core or parallel implementations are straightforward.

4 Normals

So far we have considered functional data. Now we turn to the problem of approximating a surface represented by points $\mathscr{P}$ in space. In this setting, we don't have a suitable parameter domain to directly apply the techniques explained above. It turns out, that almost all approaches compensate for this lack of information by using or approximating tangent planes or normals on or close to the point set.

Normals might be part of the data, or not. If normals are missing, we can try to estimate them as follows: Assume we want to compute the normal $\mathbf{n}$ in a location $\mathbf{q}$ in space. The points close to $\mathbf{q}$ describe the surface around $\mathbf{q}$. A tangent in $\mathbf{q}$ should be as close as possible to these close points.

Determining a tangent plane around $\mathbf{q}$ can be formulated as a least squares problem. We search a plane $H(\mathbf{x}) : \mathbf{n}^\mathsf{T}\mathbf{q} = \mathbf{n}^\mathsf{T}\mathbf{p}_i, \|\mathbf{n}\| = 1$ passing through $\mathbf{q}$ that minimizes the squares $(\mathbf{n}^\mathsf{T}(\mathbf{q}-\mathbf{p}_i))^2$. However, we want to consider only few points $\mathbf{p}_i$ close to $\mathbf{q}$. We could do this by either using only the k-nearest neighbors of $\mathbf{q}$, or by weighting close points with a locally supported weight function θ. Because the k-nearest neighbor approach could be simulated by using a hat function with appropriate radius for θ, we will only detail the locally weighted version. Then, $\mathbf{n}$ is defined

by the following minimization problem:

$$\min_{\|\mathbf{n}\|=1} \sum_i \Big(\mathbf{n}^\mathsf{T}\mathbf{p}_i - \mathbf{q})\Big)^2 \theta(\|\mathbf{p}_i - \mathbf{p}_q\|) \tag{26}$$

This is a non-linear optimization problem, because of the quadratic constraint $\|\mathbf{n}\| = 1$. To arrive at a computable solution, we use the outer product matrices $(\mathbf{p}_i - \mathbf{q})(\mathbf{p}_i - \mathbf{q})^\mathsf{T}$ and rewrite the functional to be minimized as

$$m(\mathbf{n}) = \mathbf{n}^\mathsf{T} \left(\sum_i (\mathbf{p}_i - \mathbf{q})(\mathbf{p}_i - \mathbf{q})^\mathsf{T} \theta(\|\mathbf{p}_i - \mathbf{q}\|) \right) \mathbf{n}, \tag{27}$$

and inspect the eigenvalue/eigenvector decomposition of the sum of outer products

$$\sum_i (\mathbf{p}_i - \mathbf{q})(\mathbf{p}_i - \mathbf{q})^\mathsf{T} = \mathbf{E}\mathrm{diag}(\lambda_0, \lambda_1, \ldots)\mathbf{E}^\mathsf{T}. \tag{28}$$

Using this decomposition we see that in transformed coordinates $\mathbf{E}^\mathsf{T}\mathbf{n}$ the functional $m(\mathbf{n}) = \mathbf{n}^\mathsf{T}\mathbf{E}\mathrm{diag}(\lambda_0, \lambda_1, \ldots)\mathbf{E}^\mathsf{T}\mathbf{n}$ has only pure quadratic terms, and each of these quadratic terms has an eigenvalue as coefficient. Let $\lambda_0 \leq \lambda_1, \ldots$, then $m(\mathbf{n})$ clearly attains its minimum among all unit-length vectors for $mvE^\mathsf{T}\mathbf{n} = (1, 0, 0, \ldots)^\mathsf{T}$. This means, $\mathbf{n}$ is the eigenvector corresponding to the smallest eigenvalue.

Fitting a tangent plane will only yield a normal direction, not an orientation. We assume that the correct orientation can be derived from inside/outside information generated using scanning the object.

5 Implicit surfaces from points and offset points

The basic idea of approaches based on implicit surfaces is to assume that all points on the surface have zero value, i.e. the surface is implicitly defined by

$$\mathscr{S} = \{\mathbf{x} | \hat{f}(\mathbf{x}) = 0\}. \tag{29}$$

In this setting, the point set delivers a set of constraints of the form

$$\hat{f}(\mathbf{p}_i) = 0. \tag{30}$$

Now, our approach is to apply the techniques for local function estimation presented in the preceding section. However, all of these methods would result in $\hat{f} = 0$, as this perfectly satisfies all constraints. We obviously need additional non-zero constraints. These additional constraints have to be generated based on the given point data.

The normals can be used to generate additional point constraints for $\hat{f}$. A standard trick is this: Move a small step (say δ) from $\mathbf{p}_i$ in normal direction outwards from the surface. This point is $\mathbf{p}_i + \delta\mathbf{n}_i$. Require $\hat{f}$ to be δ at this point. The surface could be additionally supported by also moving to the inside and requiring that the value at $\mathbf{p}_i - \delta\mathbf{n}_i$ is $-\delta$.

A potential danger of this approach is that the $\mathbf{p}_i + \delta\mathbf{n}_i$ is not really δ away from the surface, because we the step is so large that we have moved towards some other part of the surface. A

good strategy to check and avoid this is to compute the closest point to $\mathbf{p}_i + \delta\mathbf{n}_i$. If this is no $\mathbf{p}_i$, the step size δ has to be reduced until this holds true.

If a spatial subdivision is used to organize the points this is another good way to add non-zero constraints. In each of the corners and centers of a cell the distance to the surface is approximated as the smallest distance to any point of the set. A sign for the distance can be computed from inside/outside information. For small distances, the distance to the closest point becomes less reliable. We propose to rather compute the distances to the k-nearest points ($k = 3$) and check that they all have the same sign.

The result of either procedure is a set of additional constraints of the form

$$\hat{f}(\mathbf{p}_{N+i}) = d_i. \tag{31}$$

Together with the constraints $\hat{f}(\mathbf{p}_i) = 0$ they can be used to approximate a function $\hat{f}$ using any of the techniques for approximating functions as described in the last section.

Several works discuss the details of using RBF for approximating the implicit function [Mur91, MYR*01, CBC*01, DST01, TO02, DTS02, OBS03], spatial subdivisions have been used together with Partition of Unity weights, either RBF approximations in a k-d tree [TRS04], or local polynomial and specific sharp edge functions in the cells of an octree [OBA*03].

6 Implict surface from points and tangent frames

Rather than generating additional point constraints to make the standard function approximation techniques applicable we could also try to adapt them to the more general setting. An underlying idea of several approaches in this direction is to estimate local tangent frames and then use a standard technique in this local tangent frame.

6.1 Hoppe's and related approaches

A simple approach to generate an implicit function $\hat{f}$ based on the normals is due to Hoppe [HDD*92]: For a point in space $\mathbf{x}$ compute the closest point $\mathbf{p}_i$ in $\mathscr{P}$. Then set $\hat{f}(\mathbf{x})$ to $\mathbf{n}_i(\mathbf{p}_i - \mathbf{x})$, i.e. the signed distance to the tangent plane through $\mathbf{p}_i$. This yields a piecewise linear approximation of signed distances to the surface. The set of points in space associated to the same point $\mathbf{p}_i$ form the Voronoi cell around $\mathbf{p}_i$. So, another viewpoint on this is that we use local linear approximations, however, for Voronoi cells we use a local frame based on the tangent plane.

One can compute a smoother surface approximation by exploiting the Voronoi cells around the points and using Voronoi interpolation as explained in the preceding section. This has been exploited by Boissonnat et al. [BC00].

6.2 MLS surfaces

The MLS surface S_P of $\mathscr{P}$ is defined implicitly by a projection operator. The basic idea for projecting a point $\mathbf{r}$ onto S_P is based on two steps: First, a locally tangent reference domain H is computed. Then, a local bivariate polynomial is fitted over H to the point set.

However, to compensate for points with some distance to the surface, we don't restrict the tangent plane to pass through the point $\mathbf{r}$. Yet, we still want to weight the influence of points based on the distance to the origin of the tangent frame. This leads to a more complex minimization problem, however, yields the desired projection property.

Specifically, the local reference domain $H = \{\mathbf{x} | \langle \mathbf{n}, \mathbf{x} \rangle - D = 0, \|\mathbf{n}\| = 1$ is determined by minimizing

$$\sum_{i=1}^{N} \left(\langle \mathbf{n}, \mathbf{p_i} - \mathbf{r} - t\mathbf{n} \rangle \right)^2 \theta(\|\mathbf{p_i} - \mathbf{r} - t\mathbf{n}\|) \tag{32}$$

among all normal directions $\mathbf{n}$ and offsets t. Let $\mathbf{q_i}$ be the projection of $\mathbf{p_i}$ onto H, and f_i the height of $\mathbf{p_i}$ over H, i.e $f_i = \mathbf{n} \cdot (\mathbf{p_i} - \mathbf{q})$. The polynomial approximation g is computed by minimzing the weighted least squares error

$$\sum_{i=1}^{N} \left(g(x_i, y_i) - f_i \right)^2 \theta(\|\mathbf{p_i} - \mathbf{r} - t\mathbf{n}\|) \tag{33}$$

The projection of $\mathbf{r}$ is given by

$$MLS(\mathbf{r}) = \mathbf{r} + (t + g(0,0))\mathbf{n} \tag{34}$$

Formally, the surface S_P is the set of points that project onto themselves. We can also define the surface in the standard notation using

$$\hat{f}(\mathbf{x}) = \|(t + g(0,0))\mathbf{n}(\mathbf{x})\| \tag{35}$$

The projection procedure itself has turned out to be a useful computational method for computing points on the surface. We point the reader to [Lee00, ABCO*01, PGK02, PKG02, ABCO*03, FCOAS03, Lev03, PKKG03] for details on the properties, extensions, and implementation of this approach.

6.3 Surfaces from normals and weighted averages

Inspired by MLS surfaces, we can also define the surface implicitly based on normal directions and weighted averages. The implicit function $f : \mathbb{R}^3 \to \mathbb{R}$ describes the distance of a point $\mathbf{x}$ to the weighted average $\mathbf{a}(\mathbf{x})$ projected along the normal direction $\mathbf{n}(\mathbf{x})$:

$$\hat{f}(\mathbf{x}) = \mathbf{n}(\mathbf{x}) \cdot (\mathbf{a}(\mathbf{x}) - \mathbf{x}) \tag{36}$$

where the weighted average of points at a location $\mathbf{s}$ in space is

$$\mathbf{a}(\mathbf{s}) = \frac{\sum_{i=0}^{N-1} \theta(\|\mathbf{s} - \mathbf{p}_i\|)\mathbf{p}_i}{\sum_{i=0}^{N-1} \theta(\|\mathbf{s} - \mathbf{p}_i\|)}. \tag{37}$$

If θ is locally supported one has to make sure to compute $\hat{f}$ only in the support of the weights. Computing the weighted average and the local tangent frame also allows to define boundaries of the surface in a natural way: We inspect the relative location of the weighted average $\mathbf{a}(\mathbf{x})$ in the points. For points far away from the point set the distance $||\mathbf{x} - \mathbf{a}(\mathbf{x})||$ increases, while we

expect this distance to be rather small for locations close to (or "inside") the points. The main idea for defining a boundary is to require $||\mathbf{x} - \mathbf{a}(\mathbf{x})||$ to be less than a user-specified threshold.

For computing points on the surface, several efficient curve-surface intersection and projection operators can be implemented. These intersections can be computed independent of normal orientation. More details can be found in [AA03b, AA03a, AA04]

Acknowledgements

Numerous people have contributed to this work. I gratefully acknowledge the help of Nina Amenta, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, Markus Gross, Leonidas Guibas, David Levin, Jörg Peters, Ulrich Reif, Claudio Silva, Denis Zorin, and several anonymous reviewers.

This work was supported by a grants from the European Union, the German Ministry of Science and Education (BMBF), the Israeli Ministry of Science, the German Israeli Foundation (GIF), and the Israeli Academy of Sciences (center of excellence).

References

[AA03a] ADAMSON A., ALEXA M.: Approximating and intersecting surfaces from points. In *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing* (2003), Eurographics Association, pp. 230–239.

[AA03b] ADAMSON A., ALEXA M.: Ray tracing point set surfaces. In *Proceedings of Shape Modeling International* (2003).

[AA04] ADAMSON A., ALEXA M.: Approximating bounded, non-orientable surfaces from points. In *Proceedings of Shape Modeling International 2004* (2004), IEEE Computer Society. accepted for publication.

[ABCO*01] ALEXA M., BEHR J., COHEN-OR D., FLEISHMAN S., LEVIN D., SILVA C. T.: Point set surfaces. In *Proceedings of the conference on Visualization '01* (2001).

[ABCO*03] ALEXA M., BEHR J., COHEN-OR D., FLEISHMAN S., LEVIN D., SILVA C. T.: Computing and rendering point set surfaces. *IEEE Transactions on Computer Graphics and Visualization 9*, 1 (2003), 3–15.

[BC00] BOISSONNAT J.-D., CAZALS F.: Smooth shape reconstruction via natural neighbor interpolation of distance functions. In *Proceedings of the 16th Annual Symposium on Computational Geometry (SCG-00)* (N. Y., June 12–14 2000), ACM Press, pp. 223–232.

[BM97] BABUŠKA I., MELENK J. M.: The partition of unity method. *International Journal of Numerical Methods in Engineering 40* (1997), 727–758.

[CBC*01] CARR J. C., BEATSON R. K., CHERRIE J. B., MITCHELL T. J., FRIGHT W. R., MCCALLUM B. C., EVANS T. R.: Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of ACM SIGGRAPH 2001* (2001), Computer Graphics Proceedings, Annual Conference Series, pp. 67–76.

[DST01] DINH H. Q., SLABAUGH G., TURK G.: Reconstructing surfaces using anisotropic basis functions. In *International Conference on Computer Vision (ICCV) 2001* (Vancouver, Canada, July 2001), pp. 606–613.

[DTS02] DINH H. Q., TURK G., SLABAUGH G.: Reconstructing surfaces by volumetric regularization. *IEEE Transactions on Pattern Analysis and Machine Intelligence 24*, 10 (October 2002), 1358–1371.

[FCOAS03] FLEISHMAN S., COHEN-OR D., ALEXA M., SILVA C. T.: Progressive point set surfaces. *ACM Transactions on Graphics (TOG) 22*, 4 (2003), 997–1011.

[FN80] FRANKE R., NIELSON G.: Smooth interpolation of large sets of scattered data. *International Journal of Numerical Methods in Engineering 15* (1980), 1691–1704.

[GS00] GRIEBEL M., SCHWEITZER M. A.: A Particle-Partition of Unity Method for the solution of Elliptic, Parabolic and Hyperbolic PDE. *SIAM J. Sci. Comp. 22*, 3 (2000), 853–890.

[GS02a] GRIEBEL M., SCHWEITZER M. A.: A Particle-Partition of Unity Method—Part II: Efficient Cover Construction and Reliable Integration. *SIAM J. Sci. Comp. 23*, 5 (2002), 1655–1682.

[GS02b] GRIEBEL M., SCHWEITZER M. A.: A Particle-Partition of Unity Method—Part III: A Multilevel Solver. *SIAM J. Sci. Comp. 24*, 2 (2002), 377–409.

[HDD*92] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Surface reconstruction from unorganized points. *Computer Graphics (Proceedings of SIGGRAPH 92) 26*, 2 (July 1992), 71–78.

[Lee00] LEE I.-K.: Curve reconstruction from unorganized points. *Computer Aided Geometric Design 17*, 2 (February 2000), 161–177.

[Lev98] LEVIN D.: The approximation power of moving least-squares. *Math. Comp. 67* (1998), 1517–1531.

[Lev03] LEVIN D.: Mesh-independent surface interpolation. *Geometric Modeling for Scientific Visualization* (2003).

[LS98] LANCASTER P., SALKAUSKAS K.: Surfaces generated by moving least squares methods. *Mathematics of Computation 37* (1998), 141–158.

[Mur91] MURAKI S.: Volumetric shape description of range data using "Blobby Model". *Computer Graphics 25*, 4 (July 1991), 227–235. Proceedings of ACM SIGGRAPH 1991.

[MYR*01] MORSE B. S., YOO T. S., RHEINGANS P., CHEN D. T., SUBRAMANIAN K. R.: Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions. In *Shape Modeling International 2001* (Genova, Italy, May 2001), pp. 89–98.

[OBA*03] OHTAKE Y., BELYAEV A., ALEXA M., TURK G., SEIDEL H.-P.: Multi-level partition of unity implicits. *ACM Transactions on Computer Graphics (SIGGRAPH 2003 Proceedings) 22*, 3 (2003), 463–470.

[OBS03] OHTAKE Y., BELYAEV A., SEIDEL H.-P.: A multi-scale approach to 3d scattered data interplation with compactly supported basis functions. In *Proceedings of Shape Modeling International 2003* (May 2003).

[PGK02] PAULY M., GROSS M., KOBBELT L. P.: Efficient simplification of point-sampled surfaces. In *Proceedings of the conference on Visualization '02* (2002), pp. 163–170.

[PKG02] PAULY M., KOBBELT L., GROSS M.: *Multiresolution Modeling of Point-Sampled Geometry*. Tech. rep., ETH Zurich, Department of Computer Science, 2002.

[PKKG03] PAULY M., KEISER R., KOBBELT L. P., GROSS M.: Shape modeling with point-sampled geometry. *ACM Transactions on Graphics (TOG) 22*, 3 (2003), 641–650.

[She68] SHEPARD D.: A two dimensional interpolation function for irregular spaced data. In *Proceedings of 23rd ACM National Conference* (1968), pp. 517–524.

[TO02] TURK G., O'BRIEN J.: Modelling with implicit surfaces that interpolate. *ACM Transactions on Graphics 21*, 4 (October 2002), 855–873.

[TRS04] TOBOR I., REUTER P., SCHLICK C.: Multiresolution reconstruction of implicit surfaces with attributes from large unorganized point sets. In *Proceedings of Shape Modeling International (SMI 2004)* (2004). in press.

[Wen95] WENDLAND H.: Piecewise polynomial, positive definite and compactly supported radial basis functions of minimal degree. *Advances in Computational Mathematics 4* (1995), 389–396.

Shape Modeling with Points
Supplemental Course Notes
SIGGRAPH 2004

Mark Pauly

May 12, 2004

Chapter 1

Local Surface Analysis

Mark Pauly

This part of the course notes describes some of the fundamental techniques that are used to define more sophisticated processing algorithms for point-based surfaces. In particular, efficient methods for estimating local surface properties, such as normal vector and curvature, are described. These computations are based on local neighborhoods of point samples, which are defined as subsets of a given point cloud P that satisfy some local neighborhood relation. Also, the moving least squares (MLS) surface model is discussed, which defines a smooth manifold surface from the point cloud and allows the evaluation of the signed distance function.

The concept of a 2-manifold surface embedded in 3-space is crucial for the algorithms described in these notes. The following definition of a manifold surface is taken from [dC76]:
Definition: A subset $S \subset \mathbb{R}^3$ is a *2-manifold surface*, if for each $\mathbf{x} \in S$, there exists a neighborhood V in $\mathbb{R}^3$ and a map $X : U \rightarrow V \cap S$ of an open set $U \subset \mathbb{R}^2$ onto $V \cap S \subset \mathbb{R}^3$ such that

- X is differentiable, i.e., if $X(u,v) = (x(u,v), y(u,v), z(u,v))$ for $(u,v) \in U$ then the functions x, y, and z have continuous partial derivatives of all orders in U,
- X is a homeomorphism, i.e., the inverse $X^{-1} : V \cap S \rightarrow U$ exists and is continuous,
- for each $\mathbf{u} \in U$, the differential $dX_{\mathbf{u}}$ is one-to-one.

Note that the definition of a surface is based on local (possibly infinitesimal) neighborhoods. Intrinsic properties of the surface, such as tangent plane or Gaussian curvature, are defined with respect to such neighborhoods (see [dC76] for details).

1.1 Local Neighborhoods

In the discrete setting, a local neighborhood can be defined through spatial relations of the sample points. Given a point $\mathbf{p} \in P$, a local neighborhood is defined as an index set N_p, such that each $\mathbf{p}_i, i \in N_p$ satisfies a certain neighborhood condition. This condition should be set in such a way that the points of N_p adequately represent a small, local surface patch around the point $\mathbf{p}$. For the computations described below it is important that local neighborhoods only depend on the geometric locations of the point samples in space, not on some global combinatorial structure associated with the point cloud. If the point cloud is locally uniform, the k-nearest neighbors relation is most commonly used to define local neighborhoods. Other relations, e.g., local Delaunay neighbors [FR01], or Linsen's method [Lin01] can be used for non-uniformly sampled models.

1.1.1 Local Sampling Density

An important measure for analyzing point-sampled surfaces is the local density of the point samples, i.e., the number of samples per unit area. The local sampling density ρ_i at a sample point $\mathbf{p}_i \in P$ can be estimated by computing the sphere with minimum radius r_i centered at $\mathbf{p}_i$ that contains the k-nearest neighbors to $\mathbf{p}_i$. By approximating the intersection of the underlying surface and this sphere with a disc, ρ_i can be defined as $\rho_i = k/(\pi r_i^2)$. This discrete estimation of local sampling density can be extended to a continuous function $\rho : \mathbb{R}^3 \rightarrow \mathbb{R}_+$ using scattered data approximation. A continuous density function $\rho(\mathbf{x})$ is useful because it provides for each point $\mathbf{x}$ on the surface S an estimate of the sampling density of a small patch around $\mathbf{x}$. Similarly, an estimate for the local sample spacing can be derived as $\eta(\mathbf{x}) = 1/\sqrt{\rho(\mathbf{x})}$. η measures the average distance of sample points within the sphere defined by the k-nearest neighbors.

1.2 Covariance Analysis

Based on the k-neighborhood relation described above, local surface properties at a point $\mathbf{p} \in P$ can be estimated using a statistical analysis of the neighboring samples. In particular, eigenanalysis of the covariance matrix of the point positions and normals of a local neighborhood yields efficient algorithms for estimating normal vectors, and surface and normal variation (to be defined below). Let $\bar{\mathbf{p}}$ be the centroid of the neighborhood of $\mathbf{p}$, i.e.,

$$\bar{\mathbf{p}} = \frac{1}{|N_p|} \sum_{i \in N_p} \mathbf{p}_i$$

The 3×3 covariance matrix $\mathbf{C}$ for the sample point $\mathbf{p}$ is then given as

$$\mathbf{C} = \begin{bmatrix} \mathbf{p}_{i_1} - \bar{\mathbf{p}} \\ \cdots \\ \mathbf{p}_{i_k} - \bar{\mathbf{p}} \end{bmatrix}^T \cdot \begin{bmatrix} \mathbf{p}_{i_1} - \bar{\mathbf{p}} \\ \cdots \\ \mathbf{p}_{i_k} - \bar{\mathbf{p}} \end{bmatrix}, i_j \in N_p \tag{1.1}$$

$\mathbf{C}$ describes the statistical properties of the distribution of the sample points in the neighborhood of point $\mathbf{p}$ by accumulating the squared distances of these points from the centroid $\bar{\mathbf{p}}$. Consider the eigenvector problem $\mathbf{C} \cdot \mathbf{v}_l = \lambda_l \cdot \mathbf{v}_l, l \in \{0, 1, 2\}$. Since $\mathbf{C}$ is symmetric and positive semi-definite, all eigenvalues λ_l are real-valued and the eigenvectors $\mathbf{v}_l$ form an orthogonal frame, corresponding to the principal components of the point set defined by N_p [Jol86]. The λ_l measure the *variation* of the $\mathbf{p}_i, i \in N_p$, along the direction of the corresponding eigenvectors. The total variation, i.e., the sum of squared distances of the from the centroid is given by

$$\sum_{i \in N_p} \|\mathbf{p}_i - \bar{\mathbf{p}}\|^2 = \lambda_0 + \lambda_1 + \lambda_2.$$

Normal Estimation

Assuming $\lambda_0 \leq \lambda_1 \leq \lambda_2$, it follows that the plane $T(\mathbf{x}) : (\mathbf{x} - \bar{\mathbf{p}}) \cdot \mathbf{v}_0 = 0$ through $\bar{\mathbf{p}}$ minimizes the sum of squared distances to the neighbors of $\mathbf{p}$ [Jol86]. Thus $\mathbf{v}_0$ approximates the surface normal $\mathbf{n}_p$ at $\mathbf{p}$, or in other words, $\mathbf{v}_1$ and $\mathbf{v}_2$ span the tangent plane at $\mathbf{p}$ (see Figure 1.1 (a)). A straightforward extension to this scheme uses weighted covariances, typically applying a Gaussian or polynomial weight function that drops with increasing distance to $\mathbf{p}$ [AA03a].

Normal Orientation

A consistent orientation of the normal vectors can be computed using a method based on the Euclidean minimum spanning tree of the point cloud, as described in [HDD*94]. The algorithm starts with an

extremal point, e.g., the sample with largest z-coordinate, and orients its normal to point away from the centroid of the point cloud. The normal vector of each adjacent point in the minimum spanning tree can then be oriented based on the assumption that the angle of the normal vectors of adjacent points is less than $\pi/2$ (see Figure 1.1 (b)). If the underlying surface is orientable (note that some surface are nonorientable, e.g., the Moebius strip) and the sampling distribution is sufficiently dense, then a consistent orientation of the normals will be obtained after all points of the point cloud have been visited.

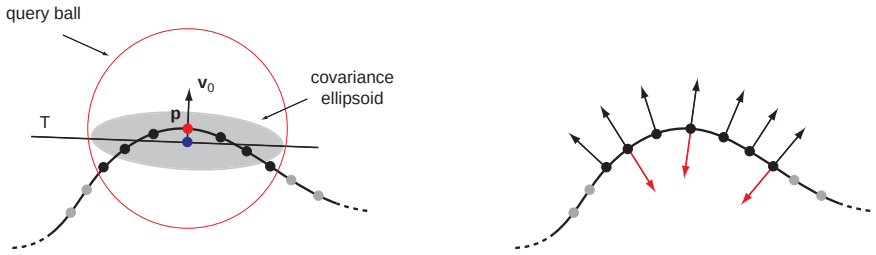


Figure 1.1: Normal estimation (2D for illustration). (a) Computing the tangent plane using covariance analysis, (b) normal orientation using a minimum spanning tree, where the red normals have been flipped, since the angle to the next adjacent oriented normal is larger then $\pi/2$.

Surface Variation

λ_0 quantitatively describes the variation along the surface normal, i.e., estimates how strongly the points deviate from the tangent plane. The *surface variation* at point $\mathbf{p}$ in a neighborhood of size n is defined as

$$\sigma_n(\mathbf{p}) = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2}.$$

The maximum surface variation $\sigma_n(\mathbf{p}) = 1/3$ is assumed for completely isotropically distributed points, while the minimum value $\sigma_n(\mathbf{p}) = 0$ indicates that all points lie in a plane. Note also that λ_1 and λ_2 describe the variation of the sampling distribution in the tangent plane and can thus be used to estimate local anisotropy [PGK02].

Normal Variation

A similar method for local surface analysis considers the covariance matrix of the surface normals, i.e.,

$$\mathbf{C}' = \sum_{i \in N_p} \mathbf{n}_i^T \cdot \mathbf{n}_i.$$

Let $\lambda'_0 \leq \lambda'_1 \leq \lambda'_2$ be the eigenvalues of $\mathbf{C}'$ with corresponding eigenvectors $\mathbf{v}'_0, \mathbf{v}'_1$, and $\mathbf{v}'_2$. As Garland discusses in [Gar99], λ'_2 measures the variation of the surface normals in the direction of the mean normal, while λ'_1 measures the maximum variation on the Gauss sphere. Thus the normal variation can be defined as $\sigma'_n(\mathbf{p}) = \lambda'_1$. Garland also analyzes the covariance matrix of the normal vectors in the context of differential geometry. He shows that under mild conditions on the smoothness of the surface, the eigenvectors $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$ converge to the direction of minimum curvature, maximum curvature, and mean normal, respectively, when sampling density goes to infinity. Figure 1.2 compares surface variation and normal variation for the Max Planck model consisting of 139,486 points. With increasing neighborhood size, a smoothing of the variation estimates can be observed.

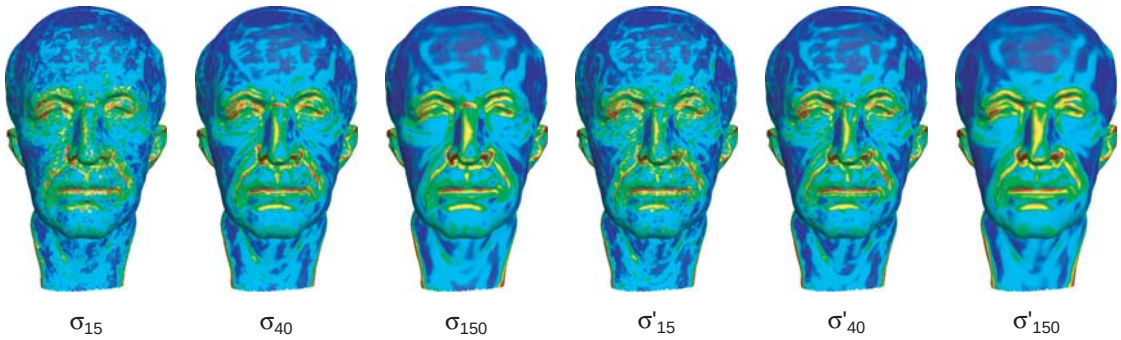


Figure 1.2: Comparison of surface variation (left) and normal variation (right) for increasing size of the local neighborhoods.

1.3 Moving Least Squares Surfaces

Given a point cloud P the goal of a point-based surface model is to define a surface S that approximates or interpolates the sample points $\mathbf{p}_i \in P$. For the algorithms defined in these notes the surface model should satisfy the following requirements:

- Smoothness: The surface should be smooth and differentiable, preferably in a controllable manner. This means that there should be some mechanism to adjust the smoothness depending on the intended application.
- Locality: The evaluation of the surface model should be local, i.e., only points within a certain distance should influence the definition of the surface at a particular location. Locality is desirable to enable local modifications without affecting the global shape of the surface. It also increases the efficiency of the computations.
- Stability: The evaluation of the surface model should be stable and robust, even for non-uniform sampling distributions. This means that the surface model needs to be adaptive, i.e dependent on the local sampling density.

Methods for interpolating or approximating functions in $\mathbb{R}^d$ from a discrete set of scattered data values have been studied extensively in the past. Examples include reconstruction using finite elements, radial basis functions, and moving least squares (MLS) approximation [LS86]. The latter two are advantageous because they are *mesh-less*, i.e., do not require a consistent tessellation of the function domain.

Recently, Levin has introduced an extension of the moving least squares approximation to surfaces [Lev03]. This method can be used for computing local approximations of the surface represented by a point cloud, and to evaluate the corresponding signed distance function. The idea is to locally approximate the surface by polynomials that minimize a weighted least squares error at the data points. Since the method is solely based on Euclidean distance between sample points, no additional combinatorial structure on the point cloud is required.

Given a point set P, the MLS surface $S(P)$ is defined as the stationary set of a projection operator Ψ_P, i.e., $S(P) = \{\mathbf{x} \in \mathbb{R}^3 | \Psi_P(\mathbf{x}) = \mathbf{x}\}$. The operator Ψ_P is defined by a two-step procedure:

- Compute a local reference plane $H = \{\mathbf{y} \in \mathbb{R}^3 | \mathbf{y} \cdot \mathbf{n} - D = 0\}$ by minimizing the weighted sum of squared distances

$$\sum_i (\mathbf{p}_i \cdot \mathbf{n} - D)^2 \phi(\|\mathbf{p}_i - \mathbf{q}\|), \tag{1.2}$$

where $\mathbf{q}$ is the orthogonal projection of $\mathbf{x}$ onto H. The reference plane defines a local coordinate system with $\mathbf{q}$ at the origin. Let (u_i, v_i, f_i) be the coordinates of the point $\mathbf{p}_i$ in this coordinate system, i.e., (u_i, v_i) are the parameter values in H and f_i is the height of $\mathbf{p}_i$ over H.

- Compute a bivariate polynomial $\tilde{p}(u, v)$ that minimizes

$$\sum_i (p(u_i, v_i) - f_i)^2 \phi(\|\mathbf{p}_i - \mathbf{q}\|) \tag{1.3}$$

among all $p \in \Pi_m^2$.

The projection of $\mathbf{x}$ onto $S(P)$ is then given as $\Psi_P(\mathbf{x}) = \mathbf{q} + \tilde{p}(0,0) \cdot \mathbf{n}$. Figure 1.3 illustrates the MLS projection for a curve example in 2D. The left image shows a single projection of a point $\mathbf{x}$ onto the MLS curve $S(P)$ depicted on the right. In [Lev98] Levin analyzes the smoothness and convergence rate,

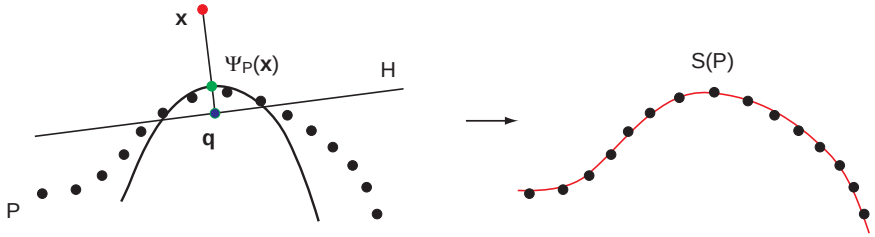


Figure 1.3: 2D illustration of the MLS projection.

which leads him to the conjecture that the smoothness of $S(P)$ directly depends on the smoothness of ϕ, i.e., if $\phi \in C^k$ then $S(P) \in C^k$. The kernel function ϕ thus controls the shape of the surface $S(P)$. A suitable choice for ϕ is the Gaussian, i.e., $\phi(x) = e^{-x^2/h^2}$, where h is a global scale factor that determines the kernel width. The Gaussian kernel has been used successfully in different applications (see also [ABCO*01, AA03b, FCOAS03]) and will be used in most of the algorithms discussed in these notes.

The scale factor h can be understood as the characteristic scale of $S(P)$, i.e., features that are smaller than h will be smoothed out. In this sense the MLS projection operator implements a low-pass filter, whose filter width can be controlled by the parameter h. Figure 1.4 shows an example an MLS surface computed with different scale factors.

Figure 1.4: Smoothing effect of the MLS projection. The Gaussian kernel width has been doubled for each image from left to right.

Signed Distance Function

An important tool for local surface analysis is the signed distance function $d_+ : \mathbb{R}^3 \rightarrow \mathbb{R}$ that measures for each point $\mathbf{x} \in \mathbb{R}^3$ the signed distance to the surface S. Using the MLS projection an approximate distance function can be defined as $d(\mathbf{x}) = \|\mathbf{x} - \Psi_P(\mathbf{x})\|$. If S is closed and the normals on S are consistently oriented, e.g., always pointing outward of the surface (see Section 1.2), then the signed distance function can be formulated as $d_+(\mathbf{x}) = (\mathbf{x} - \Psi_P(\mathbf{x})) \cdot \mathbf{n}$, where $\mathbf{n}$ is the surface normal at $\Psi_P(\mathbf{x})$ with $\|\mathbf{n}\| = 1$.

Computing the MLS Projection

The MLS projection method defined above consists of two steps: Computation of the local reference plane H and computation of the least-squares polynomial $\tilde{p}$ with respect to that reference plane. The former requires a non-linear optimization, since the distances used in the kernel function depend on the projection $\mathbf{q}$ of the point of interest $\mathbf{x}$ onto the unknown reference plane. Alexa et al. [ABCO*01] use Powell iteration to compute D and $\mathbf{n}$ in Equation 1.2. A different approach is to try and estimate directly, which determines $\mathbf{n} = (\mathbf{x}-\mathbf{q})/\|\mathbf{x}-\mathbf{q}\|$ and $D = \mathbf{q}\cdot\mathbf{n}$. This can be achieved using a Newton-type iteration based on a geometric gradient estimation as described in [PKG02].

Given the reference plane H, all points in P are projected into a local reference frame defined by H. The computation of the polynomial $\tilde{p}$ is now a linear weighted least squares approximation, which can be computed using normal equations. For example, a cubic polynomial leads to a linear system with 10 unknowns, which can be solved using a standard linear solver. Both computation of the reference plane and the fitting polynomial require order $O(n)$ computations, since they involve the entire point cloud. However, since the value of the weight function drops quickly with distance, efficient approximation schemes can be applied to significantly reduce the complexity of the computations. For example, Alexa et al. [ABCO*01] use a multi-pole scheme, where they cluster groups of points that are far away from the point of interest into a single point and use this representative in the optimization. A different approach is to only collect points within a sphere s_r of radius r centered at $\mathbf{x}$, such that the weight for each point outside this sphere is so small that the influence in the least-squares optimization is negligible. For example, given an MLS scale factor h, r can be set to $3h$ to yield an MLS weight of less then 0.001 in Equations 1.2 and 1.3 for all points outside of s_r.

Adaptive MLS Surfaces

So far the MLS surface approximation has been defined with a fixed Gaussian kernel width h. Finding a suitable h can be difficult for non-uniformly sampled point clouds, however. A large scale factor will cause excessive smoothing in regions of high sampling density. Even worse, if the filter kernel is too small, only very few points will contribute significantly to Equations 1.2 and 1.3 due to the exponential fall-off of the weight function. This can cause instabilities in the optimization because of limited precision of the computations, which lead to wrong surface approximations.

Figure 1.5 shows an example of a locally uniformly sampled surface that cannot be approximated adequately using a global scale factor. The surface is defined using a concentric sine wave whose wavelength and amplitude gradually increases towards the rim of the surface (see Figure 1.5 (c)). Similarly, the sampling density decreases towards the boundary as illustrated in Figures 1.5 (b) and (d). Thus the surface detail is coupled with the sampling density, i.e., in regions of high sampling density the surface exhibits high-frequency detail, whereas low-frequency detail is present where the surface is sampled less densely. Figure 1.5 (e) to (g) shows reconstructions of the central section of this surface using a regular sampling grid of 100×100 sample points. The Gaussian used in Figure 1.5 (e) causes substantial smoothing and leads to a significant loss of geometric detail in the central area. In Figures 1.5 (f) and (g) the kernel width has been successively halved, which improves the approximation in the central region but leads to increasing instability towards the boundary. To cope with this problem, the MLS approximation needs to adapt to the local sampling density. In regions of high

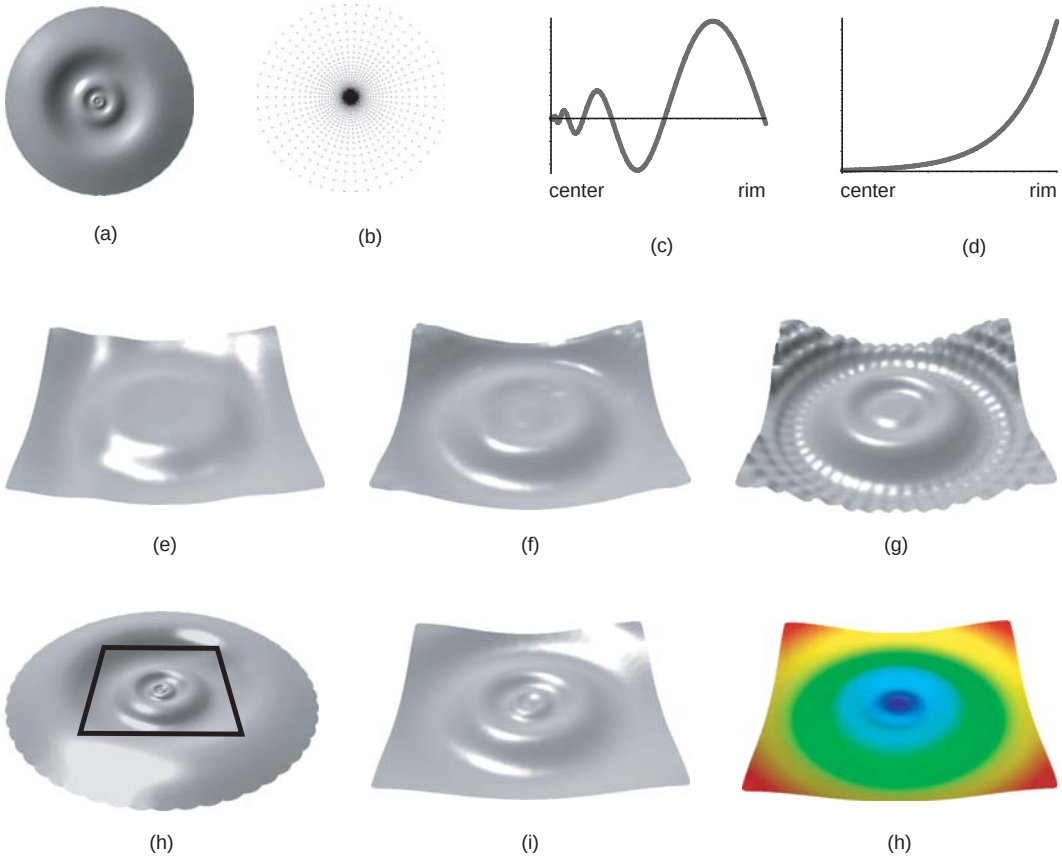


Figure 1.5: A surface that cannot be reconstructed using a fixed kernel width. (a) surface, (b) sampling pattern, (c) vertical cross-section of surface, (d), sample spacing, (e) - (g), MLS approximation of the region shown in (h) using 10k sample points for different kernel widths, (i) adaptive MLS approximation, (j) estimate of the local sample spacing, where blue corresponds to low values, while red indicates high values.

sampling density a small Gaussian kernel should be applied to accurately reconstruct potential high geometric detail. If the sampling density is low, the kernel width needs to be bigger to improve the stability of the approximation.

Given the continuous local sampling density estimate ρ of Section 1.1.1, the MLS approximation can be extended in the following way: Instead of using a fixed scale factor h for all approximation, a dynamically varying scale factor can be defined as $h(\mathbf{x}) = h/\rho(\mathbf{x})$, where $\mathbf{x}$ is the point that is to be projected onto the MLS surface. Figure 1.5 (i) shows an example of an adaptive MLS surface using the same input point cloud and sampling pattern as in (e) to (g). Observe that the high-frequency detail is faithfully recovered within the limits of the resolution of the sampling grid and that no instabilities occur at the surface boundary.

Chapter 2

Surface Simplification

Mark Pauly

Point-sampled surfaces often describe complex geometric objects using millions or even billions of sample points (see for example [LPC*00]). Reducing the complexity of such data sets is one of the key processing techniques for the design of scalable modeling and visualization algorithms. Surface simplification provides a means to generate the required approximations of a given surface that use fewer sample points than the original point model. These approximations should of course resemble the original surface as closely as possible. This part of the notes describes various surface simplification techniques for point-sampled models.

Formally, the goal of point-based surface simplification can be stated as follows: Let S be a manifold surface defined by a point cloud P. Given a target sampling rate $n < |P|$, find a point cloud P' with $|P'| = n$ such that the distance $\epsilon = d(S, S')$ of the corresponding surface S' to the original surface S is minimal. Alternatively, a target distance ϵ can be specified and the goal is to find the point cloud P' such that $d(S, S') \leq \epsilon$ and $|P'|$ is minimal.

These objectives require the definition of a metric d that measures the geometric distance between the original and the simplified surface. As will be described in Section 2.4.1 a discrete surface distance metric can be defined using the MLS projection operator.

In practice, finding the global optimum to the above problems is intractable [AS94]. Most existing surface simplification techniques therefore use different heuristics based on local error measures. In the following, three different approaches will be described and analyzed [PGK02]:

- Clustering methods split the point cloud into a number of disjoint subsets, each of which is replaced by one representative sample (see Section 2.1).
- Iterative simplification successively contracts point pairs in a point cloud according to a quadric error metric (Section 2.2).
- Particle simulation computes new sampling positions by moving particles on the point-sampled surface according to inter-particle repulsion forces (Section 2.3).

These methods are extensions and generalizations of mesh simplification algorithms to point clouds, targeted towards densely-sampled organic shapes stemming from 3D acquisition, iso-surface extraction or sampling of implicit functions. They are less suited for surfaces that have been carefully designed in a particular surface representation, such as low-resolution polygonal CAD data.

Furthermore, the goal is to design algorithms that are general in the sense that they do not require any knowledge of the specific source of the data. For certain applications this additional knowledge could be exploited to design more effective simplification algorithms, but this would also limit the applicability of these methods.

The algorithms described in this chapter differ in a number of aspects such as quality of the generated surfaces, computational efficiency and memory overhead. These features are discussed in a comparative analysis in Section 2.4. The purpose of this analysis is to give potential users of point-based surface simplification suitable guidance for choosing the right method for their specific application. Real-time applications, for instance, will put particular emphasis on efficiency and low memory footprint. Methods for creating surface hierarchies favor specific sampling patterns (e.g., [Tur01]), while visualization applications require accurate preservation of appearance attributes, such as color or material properties.

Earlier methods for simplification of point-sampled models have been introduced by Alexa et al. [ABCO*01] and Linsen [Lin01]. These algorithms create a simplified point cloud that is a true subset of the original point set by ordering iterative point removal operations according to a surface error metric. While both papers report good results for reducing redundancy in point sets, pure subsampling unnecessarily restricts potential sampling positions, which can lead to aliasing artifacts and uneven sampling distributions. To alleviate these problems, the algorithms described here re-sample the input surface and implicitly apply a low-pass filter (e.g., clustering methods perform a local averaging step to compute the cluster's centroid). In [PG01], Pauly and Gross introduced a re-sampling strategy based on Fourier theory. They split the model surface into a set of patches that are re-sampled individually using a spectral decomposition. This method directly applies signal processing theory to point-sampled geometry, yielding a fast and versatile point cloud decimation method. Potential problems arise due to the dependency on the specific patch layout and difficulties in controlling the target model size by specifying spectral error bounds.

2.1 Clustering

Clustering methods have been used in many computer graphics applications to reduce the complexity of 3D objects. Rossignac and Borrel, for example, used vertex clustering to obtain multi-resolution approximations of complex polygonal models for fast rendering [RB93]. The standard strategy is to subdivide the model's bounding box into grid cells and replace all sample points that fall into the same cell by a common representative. This volumetric approach has some drawbacks, however. By using a grid of fixed size this method cannot adapt to non-uniformities in the sampling distribution. Furthermore, volumetric clustering easily joins unconnected parts of a surface, if the grid cells are too large. To alleviate these shortcomings, surface-based clustering techniques can be applied, where clusters are build by collecting neighboring samples while regarding local sampling density. Two general approaches for building clusters will be described in these notes. An incremental method, where clusters are created by region-growing, and a hierarchical approach that splits the point cloud into smaller subsets in a top-down manner [SG01]. Both methods create a set of clusters $\{C_i\}$, such that for every point $\mathbf{p}_j \in P$ there exists a unique cluster C_i with $\mathbf{p}_j \in C_i$. The simplified point cloud is then obtained by replacing each cluster C_i by a representative sample, typically its centroid given as

$$\bar{\mathbf{p}}_i = \frac{1}{|C_i|} \sum_{j \in C_i} \mathbf{p}_j. \tag{2.1}$$

2.1.1 Incremental Clustering

Starting from a random seed point $\mathbf{p}_j \in P$, a cluster C_0 is built by successively adding nearest neighbors. This incremental region-growing is terminated when the size of the cluster reaches a maximum bound. Additionally, the maximum allowed variation σ_n of each cluster can be restricted (see Section 1.2). This results in a curvature-adaptive clustering method, where more and smaller clusters are created in regions of high surface variation. The next cluster C_1 is then build by starting the incremental growth with a new seed chosen from the neighbors of C_0 and excluding all points of from

C_0 the region-growing. This process is terminated when all sample points of P have been assigned to a cluster in $\{C_i\}$.

Due to fragmentation, this method creates many clusters that did not reach the maximum size or variation bound, but whose incremental growth was restricted by adjacent clusters. To obtain a more even distribution of clusters, sample points of all clusters that did not reach a minimum size and variation bound (typically half the values of the corresponding maximum bounds) are distributed to neighboring clusters (see Figure 2.1). Note that this potentially increases the size and variation of the clusters beyond the user-specified maxima. Figure 2.2 illustrates incremental clustering, where oriented circular splats are used to indicate the sampling distribution.

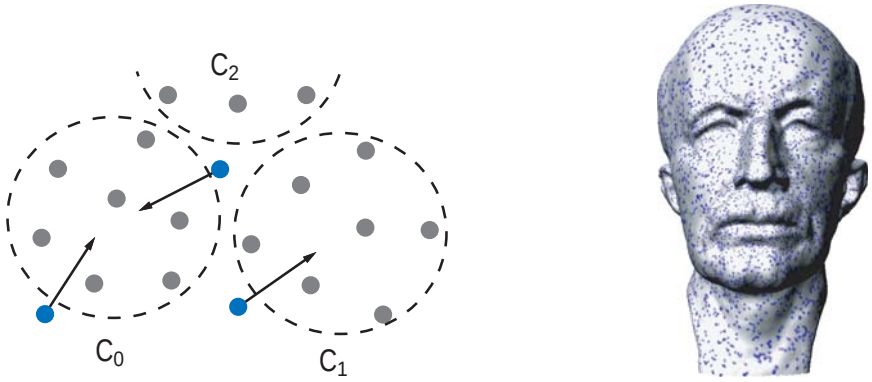


Figure 2.1: Fragmentation of incremental clustering. Gray dots correspond to sample points that are assigned to clusters that reached the necessary size and variation bounds. All other points are stray samples (blue dots) and will be attached to the cluster with closest centroid.

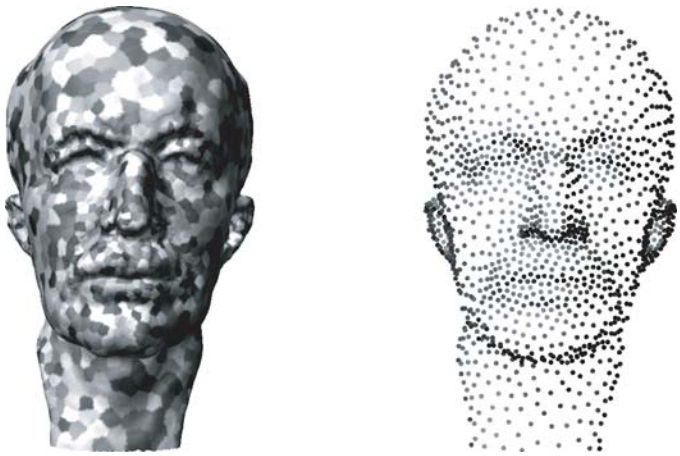


Figure 2.2: Uniform incremental clustering: The left image illustrates the corresponding clusters on the original point set (296,850 points), while the right image shows the simplified point set (2,413 points).

2.1.2 Hierarchical Clustering

An alternative method for computing the set of clusters recursively splits the point cloud using a binary space partition. The point cloud P is split, if

- the size $|P|$ is larger than the user specified maximum cluster size n_{max}, or
- the variation $\sigma_n(P)$ is above a maximum threshold σ_{max}.

The split plane is defined by the centroid of P and the eigenvector $\mathbf{v}_2$ of the covariance matrix of with largest corresponding eigenvector (see also Figure 1.1). Hence the point cloud is always split along the direction of greatest variation [SG01]. If the splitting criterion is not fulfilled, the point cloud P becomes a cluster C_i. As shown in Figure 2.3, hierarchical clustering builds a binary tree, where each leaf of the tree corresponds to a cluster. A straightforward extension to the recursive scheme uses a priority queue to order the splitting operations [BW00, SG01]. While this leads to a significant increase in computation time, it allows direct control over the number of generated samples, which is difficult to achieve by specifying $n_{max}$and σ_{max} only. Figure 2.4 illustrates adaptive hierarchical clustering.

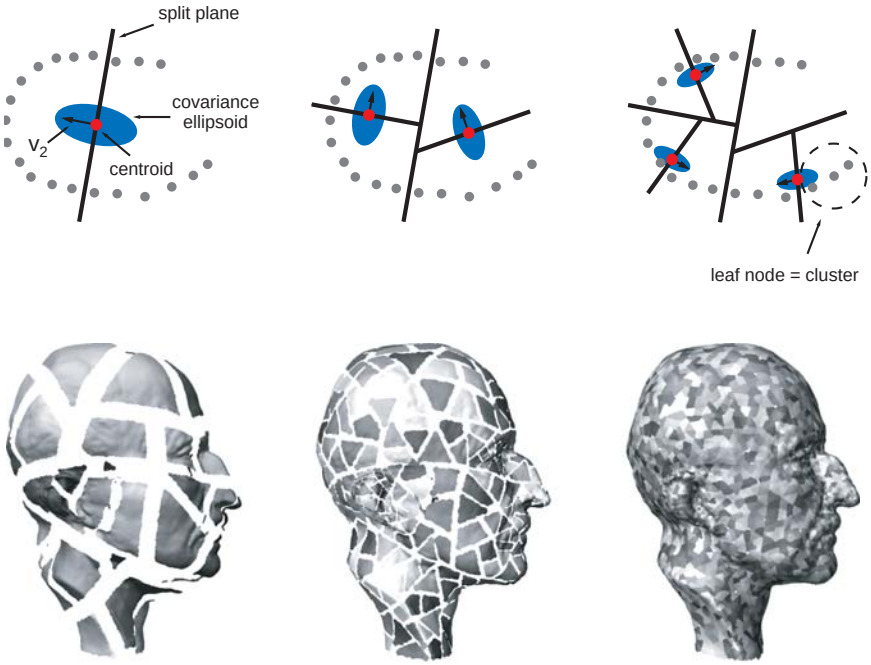


Figure 2.3: Three intermediate steps of the hierarchical clustering algorithm. (a) 2D sketch, (b) uniform hierarchical clustering for the Max Planck model.

2.2 Iterative Simplification

A different strategy for point-based surface simplification iteratively reduces the number of points using an atomic decimation operator. This approach is very similar to mesh-based simplification methods for creating progressive meshes [Hop96]. Decimation operations are usually arranged in a priority queue according to an error metric that quantifies the error caused by the decimation. The iteration is then performed in such a way that the decimation operation causing the smallest error is applied

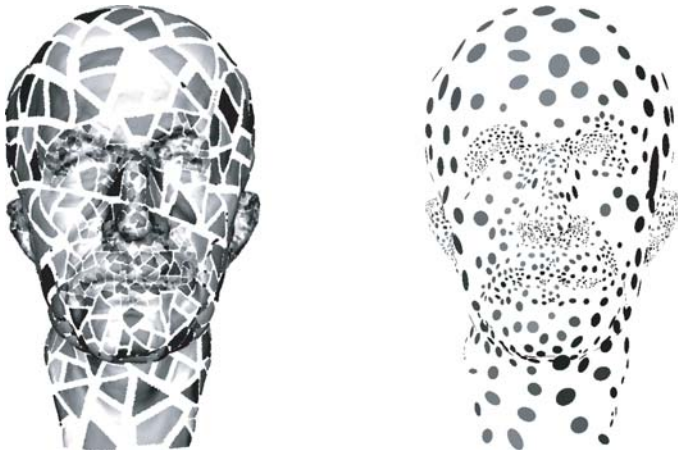


Figure 2.4: Adaptive hierarchical clustering: The left image illustrates the clusters on the original point set, while the right image shows the simplified point set (1,831 points). The size of the splats on the right image is proportional to the corresponding cluster size.

first. Earlier work [ABCO*01, Lin01] uses simple point removal, i.e., points are iteratively removed from the point cloud, resulting in a simplified point cloud that is a true subset of the original point set. As discussed above, this can lead to undesirable artifacts, which can be avoided by using point-pair contraction instead of point removal. This extension of the common edge collapse operator replaces two points $\mathbf{p}_1$ and $\mathbf{p}_2$ by a new point $\bar{\mathbf{p}}$ implicitly applying a low-pass filter by computing a weighted average of the contracted point pair.

To rate the cost of a contraction operation, an adaptation of the quadric error metric is used as presented for polygonal meshes in [GH97]. The idea there is to approximate the surface locally by a set of tangent planes and to estimate the geometric deviation of a mesh vertex $\mathbf{v}$ from the surface by the sum of the squared distances to these planes. The error quadrics for each vertex $\mathbf{v}$ are initialized with a set of planes defined by the triangles around that vertex and can be represented by a symmetric 4×4 matrix $Q_{\mathbf{v}}$. The quality of the collapse $(\mathbf{v}_1, \mathbf{v}_2) \rightarrow \bar{\mathbf{v}}$ is then rated according to the minimum of the error functional $Q_{\bar{\mathbf{v}}} = Q_{\mathbf{v}_1} + Q_{\mathbf{v}_2}$.

In order to adapt this technique to the decimation of unstructured point clouds, manifold surface connectivity is replaced by the k-nearest neighbor relation (see Section 1.1). The error quadrics for every point sample $\mathbf{p}$ are initialized by estimating a tangent plane E_i for every *edge* that connects $\mathbf{p}$ with one of its neighbors $\mathbf{p}_i$. This tangent plane is spanned by the vectors $\mathbf{e}_i = \mathbf{p} - \mathbf{p}_i$ and $\mathbf{b}_i = \mathbf{e}_i \times \mathbf{n}$, where $\mathbf{n}$ is the estimated normal vector at $\mathbf{p}$. After this initialization the point cloud decimation works exactly like mesh decimation with the point $\bar{\mathbf{p}}$ inheriting the neighborhoods of its ancestors and and being assigned the error functional $Q_{\bar{\mathbf{p}}} = Q_{\mathbf{p}_1} + Q_{\mathbf{p}_2}$. Figure 2.5 shows an example of a simplified point cloud created by iterative point-pair contraction.

2.3 Particle Simulation

In [Tur92], Turk introduced a method for re-sampling polygonal surfaces using particle simulation. The desired number of particles is randomly spread across the surface and their position is equalized using a point repulsion algorithm. Point movement is restricted to the surface defined by the individual polygons to ensure an accurate approximation of the original surface. Turk also included a curvature estimation method to concentrate more samples in regions of high curvature. Finally, the new vertices

Figure 2.5: Iterative simplification of the Max Planck model from 296,850 (left) to 2,000 sample points (middle). The right image shows all remaining potential point-pair contractions indicated as an edge between two points. Note that these edges do not necessarily form a consistent triangulation of the surface.

are re-triangulated yielding the re-sampled triangle mesh. This scheme can easily be adapted to point-sampled geometry.

2.3.1 Spreading Particles

Turk initializes the particle simulation by randomly distributing points on the surface. Since a uniform initial distribution is crucial for fast convergence, this random choice is weighted according to the area of the polygons. For point-based models, this area measure can be replaced by a density estimate (see Section 1.1.1). Thus by placing more samples in regions of lower sampling density (which correspond to large triangles in the polygonal setting), uniformity of the initial sample distribution can ensured.

2.3.2 Repulsion

For repulsion the same linear force term is used as in [Tur92], because its radius of influence is finite, i.e., the force vectors can be computed very efficiently as,

$$F_i(\mathbf{p}) = k(r - \|\mathbf{p} - \mathbf{p}_i\|) \cdot (\mathbf{p} - \mathbf{p}_i), \tag{2.2}$$

where $F_i(\mathbf{p})$ is the force exerted on particle $\mathbf{p}$ due to particle $\mathbf{p}_i$, k is a force constant and r is the repulsion radius. The total force exerted on $\mathbf{p}$ is then given as

$$F(\mathbf{p}) = \sum_{i \in N_p} F_i(\mathbf{p}), \tag{2.3}$$

where N_p is the neighborhood of $\mathbf{p}$ with radius r. Using a 3D grid data structure, this neighborhood can be computed efficiently in constant time.

2.3.3 Projection

In Turk's method, displaced particles are projected onto the closest triangle to prevent the particles from drifting away from the surface. Since no explicit surface representation is available, the MLS projection operator (see Section 1.3) is applied to keep the particles on the surface. However, applying

this projection every time a particle position is altered is computationally too expensive. Therefore, a different approach has been used: A particle $\mathbf{p}$ is kept close to the surface by simply projecting it onto the tangent plane of the point $\mathbf{p}'$ of the original point cloud that is closest to $\mathbf{p}$. The full moving least squares projection is only applied at the end of the simulation, which alters the particle positions only slightly and does not change the sampling distribution noticeably.

2.3.4 Adaptive Simulation

Using the variation estimate of Section 1.2, more points can be concentrated in regions of high curvature by scaling their repulsion radius with the inverse of the variation σ_n. It is also important to adapt the initial spreading of particles accordingly to ensure fast convergence. This can be done by replacing the density estimate ρ by $\rho \cdot \sigma_n$. Figure 2.6 gives an example of an adaptive particle simulation.

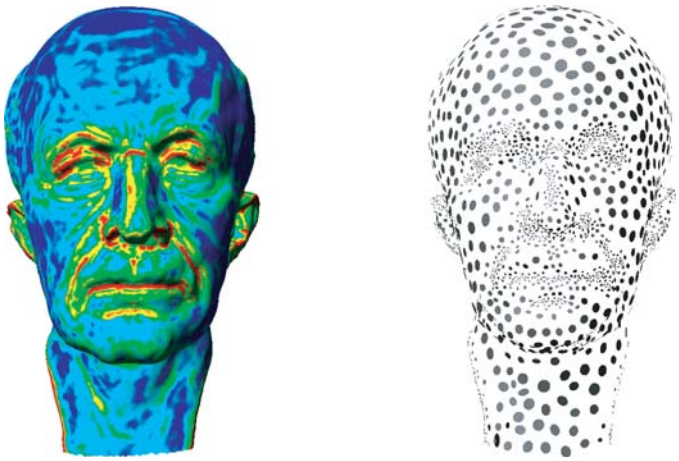


Figure 2.6: Simplification by adaptive particle simulation. The left image shows the repulsion radius on the original model determined from the surface variation estimate. Blue indicates a large radius, while red indicates small radius. On the right, a simplified model consisting of 3,000 points is shown, where the size of the splats is proportional to the repelling force of the corresponding particle.

Scaling of the repulsion radius also provides an easy means for user-controlled surface re-sampling. For this purpose a painting tool can be used (see also Chapter 4) that allows the user to directly paint the desired repulsion radius onto the surface and thus control the sampling distribution of the re-sampled surface. As illustrated in Figure 2.7, particle simulation automatically creates smooth transitions between areas of different sampling density.

2.4 Comparison

The previous sections have introduced different algorithms for point-based surface simplification. As mentioned before, none of these methods attempts to find an optimal point distribution with respect to a global distance metric, but rather uses some built-in heuristics to approximate such an optimum:

- Clustering methods try to partition the point cloud into clusters of equal size and/or surface variation, assuming that each cluster describes an equally important part of the surface.

- Iterative simplification optimizes locally according to the quadric error metric.

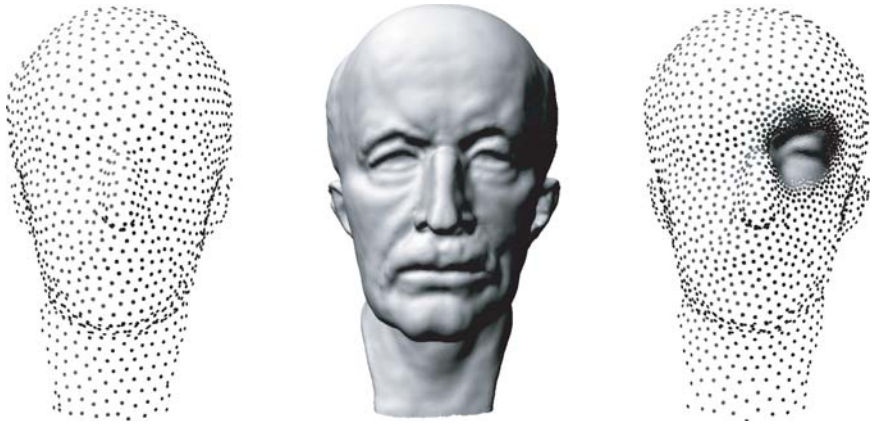


Figure 2.7: Uniform and user-controlled particle simulation. The model in the middle has been simplified to 2,000 points in the left image using uniform repulsion forces. On the right, the repulsion radius has been scaled down to 10% around the eye, leading to a higher concentration of samples in this region.

- Particle simulation is based on the assumption that a minimum of the potential energy of the particles minimizes the distance between original and re-sampled surface.

Since these heuristics are fundamentally different, the surfaces generated by these algorithms will differ significantly in general. Thus, to evaluate and compare the quality of the simplified surfaces, some generic technique for measuring the geometric distance between two point-sampled surfaces is required. This distance measure should be general in the sense that no additional knowledge about the specific method used to generate the simplified surface should be required. Further aspects in the evaluation of the various simplification algorithms include sampling distribution of the simplified model, time and space efficiency, and implementation issues.

2.4.1 Surface Error

Assume that two point clouds P and P' are given, which represent two surfaces S and S', respectively. The distance, or error, between these two surfaces is measured using a sampling approach similar to the method applied in the Metro tool [CRS].

Let Q be a set of points on S and let $d(\mathbf{q}, S') = min_{\mathbf{x}\in S'} d(\mathbf{q}, \mathbf{x})$ be the minimum distance of a point $\mathbf{q} \in Q$ to the surface S'. Then two error measures can be defined:

- Maximum error:

$$\Delta_{max}(S, S') = max_{\mathbf{q}\in Q} d(\mathbf{q}, S') \tag{2.4}$$

  The maximum error approximates the two-sided Hausdorff distance of the two surfaces. Note that the surface-based approach is crucial for meaningful error estimates, as the Hausdorff distance of the two point sets P and P' does not adequately measure the distance between S and S'.

- Average error:

$$\Delta_{avg}(S, S') = \frac{1}{|Q|} \sum_{\mathbf{q}\in Q} d(\mathbf{q}, S') \tag{2.5}$$

  The average error approximates the area-weighted integral of the point-to-surfaces distances.

The point set Q is created using the uniform particle simulation of Section 2.3. This allows the user to control the accuracy of the estimates 2.4 and 2.5 by specifying the number of points in Q. To obtain a visual error estimate, the sample points of Q can be color-coded according to the point-to-surface distance and rendered using a standard point rendering technique (see Figures 2.9 and 2.10).

$d(\mathbf{q}, S')$ is calculated using the MLS projection operator Ψ with linear basis functions (see Section 1.3). Effectively, Ψ approximates the closest point $\mathbf{q}' \in S$ such that $\mathbf{q} = \mathbf{q}' + d \cdot \mathbf{n}$ for a $\mathbf{q} \in Q$, where $\mathbf{n}$ is the surface normal at $\mathbf{q}'$ and d is the distance between $\mathbf{q}$ and $\mathbf{q}'$ (see Figure 2.8). Thus the point-to-surface distance $d(\mathbf{q}, S')$ is given as $d = \|\mathbf{q} - \mathbf{q}'\|$.

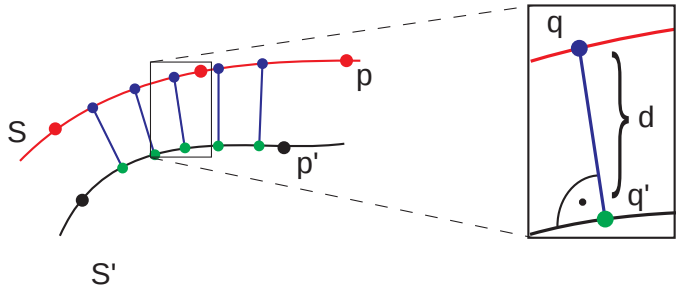


Figure 2.8: Measuring the distance between two surfaces S(red curve) and S' (black curve) represented by two point sets P (red dots) and P' (black dots). P is up-sampled to Q (blue dots) and for each $\mathbf{q} \in Q$ a base point $\mathbf{q}' \in S'$ is found (green dots). The point-to-surface distance $d(\mathbf{q}, S')$ is then equal to $d = \|\mathbf{q} - \mathbf{q}'\|$.

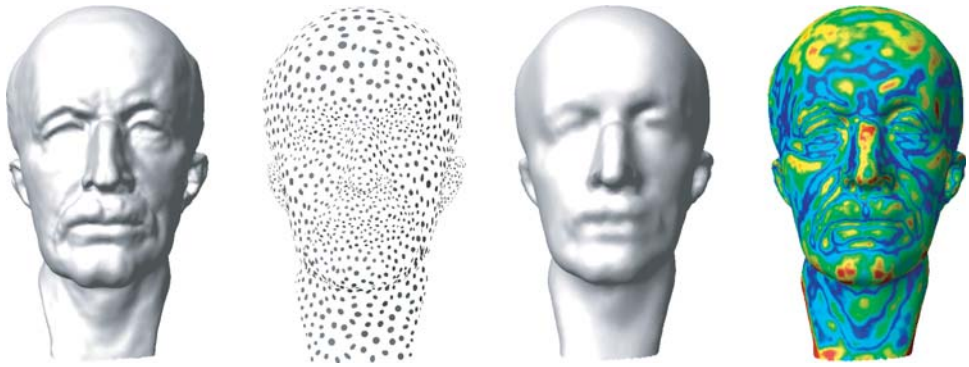


Figure 2.9: Measuring surface error. From left to right: original surface, simplified point cloud, surface obtained by up-sampling the simplified point cloud, color-coded error, where blue corresponds to a small error, while red indicates a large error.

Figure 2.10 shows visual and quantitative error estimates (scaled according to the objects bounding box diagonal) for the David model that has been simplified from 2,000,606 points to 5,000 points. Uniform incremental clustering has the highest average error. Since all clusters consist of roughly the same number of sample points, most of the error is concentrated in regions of high curvature. Adaptive hierarchical clustering performs slightly better, in particular in the geometrically complex regions of the hair. Iterative simplification and particle simulation provide lower average error and distribute the error more evenly across the surface. In general the iterative simplification method using quadric error metrics has been found to produce the lowest average surface error. Clustering methods

perform worse with respect to surface error, because they do not have the fine-grain adaptivity of the iterative simplification and particle simulation methods. For example, consider the hierarchical clustering algorithm as illustrated in Figure 2.3. The split planes generated in the earlier stages of the recursion are propagated down to all subsequent levels. This means that once a top-level split plane has been chosen, the method cannot adapt to local variations across that split plane, which leads to increased surface error.

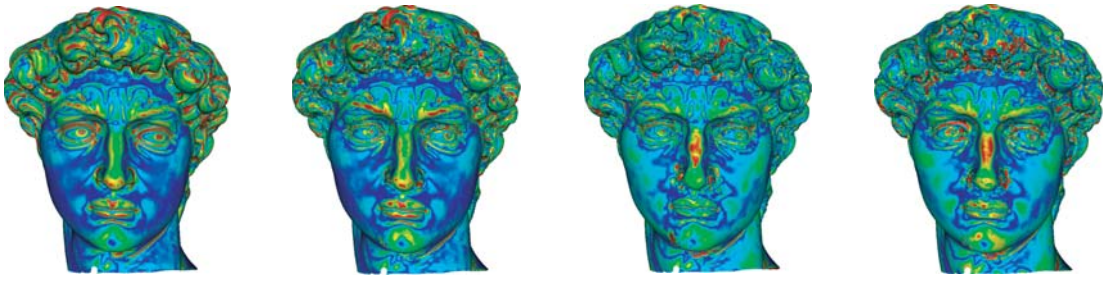


Figure 2.10: Surface Error for Michelangelos David simplified from 2,000,606 points to 5,000 points.

2.4.2 Sampling Distribution

Apart from the geometric error, the distribution of samples within the surface can be an important aspect for certain applications. As mentioned before, all simplification algorithms described here create a point cloud that is in general not a subset of the original sample set. Where this is required, methods such as those presented by Alexa et al. [ABCO*01] or Linsen [Lin01] are preferable.

For clustering methods the sampling distribution in the final model is closely linked to the sampling distribution of the input model. In some applications this might be desirable, e.g., where the initial sampling pattern carries some semantic information, such as in geological models. Other applications, e.g., pyramid algorithms for multi-level smoothing [KCVS98] or texture synthesis [Tur01], require uniform sampling distributions, even for highly non-uniformly sampled input models. Here non-adaptive particle simulation is most suitable, as it distributes sample points uniformly and independently of the sampling distribution of the underlying surface. As illustrated in Figure 2.7, particle simulation also provides a very easy mechanism for locally controlling the sampling density by scaling the repulsion radius accordingly. While similar effects can be achieved for iterative simplification by penalizing certain point-pair contractions, particle simulation offers much more intuitive control. Note that none of the surface simplification methods described above gives any guarantees that the resulting point cloud satisfies specific sampling criteria (see eg [ABK98]). It is rather left to the application to specify a suitable target sampling rate.

2.4.3 Computational Effort

Figure 2.11 shows computation times for the different simplification methods both as a function of target model size and input model size. Due to the simple algorithmic structure, clustering methods are by far the fastest simplification techniques discussed in these notes. Iterative simplification has a relatively long pre-computing phase, where initial contraction candidates and corresponding error quadrics are determined and the priority queue is set up. The simple additive update rule of the quadric metric (see Section 2.2) make the simplification itself very efficient, however. In the current

implementation particle simulation is the slowest simplification technique for large target model sizes, mainly due to slow convergence of the relaxation step. A possible improvement is the hierarchical approach introduced in [WH94]]. The algorithm would start with a small number of particles and relax until the particle positions have reached equilibrium. Then particles are split, their repulsion radius is adapted and relaxation continues. This scheme can be repeated until the desired number of particles is obtained.

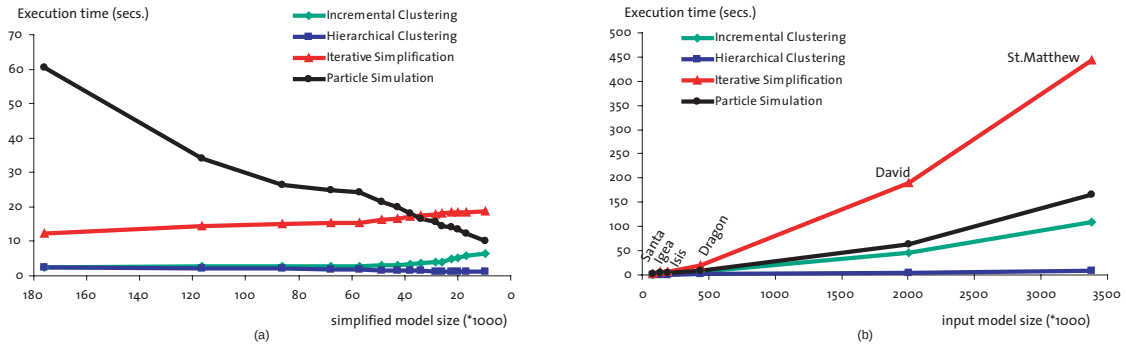


Figure 2.11: Execution times for simplification, measured on a Pentium 4 (1.8GHz) with 1Gb of main memory: (a) as a function of target model size for the dragon model (435,545 points), (b) as a function of input model size for a simplification to 1%.

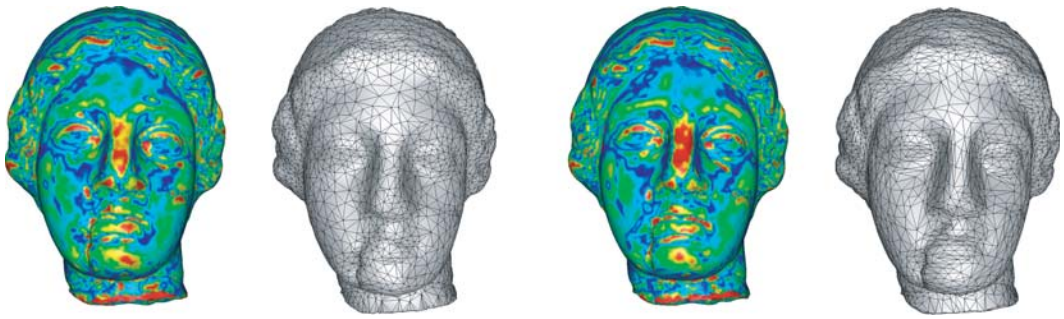
It is interesting to note that for incremental clustering and iterative simplification the execution time increases with decreasing target model size, while hierarchical clustering and particle simulation are more efficient the smaller the target models. Thus the latter are more suitable for real-time applications where the fast creation of coarse model approximations is crucial.

2.4.4 Memory Requirements and Data Structures

Currently all simplification methods discussed in these notes have been implemented in-core, i.e., require the complete input model as well as the simplified point cloud to reside in main memory. For incremental clustering a balanced kd-tree is used for fast nearest-neighbor queries, which can be implemented efficiently as an array [Sed98], requiring $4 \cdot n$ bytes, where n is the size of the input model. Hierarchical clustering builds a BSP tree, where each leaf node corresponds to a cluster. Since the tree is built by re-ordering the sample points, each node only needs to store the start and end index in the array of sample points and no additional pointers are required. Thus this maximum number of additional bytes is $2 \cdot 2 \cdot 4 \cdot m$, where m is the size of the simplified model. Iterative simplification requires 96 bytes per point contraction candidate, 80 of which are used for storing the error quadric (floating point numbers are stored in double precision, since single precision floats lead to numerical instabilities). Assuming six initial neighbors for each sample point, this amounts to $6/2 \cdot 96 \cdot n$ bytes. Particle simulation uses a 3D grid data structure with bucketing to accelerate the nearest neighbor queries, since a static kd-tree is not suitable for dynamically changing particle positions. This requires a maximum of $4 \cdot (n + k)$ bytes, where k is the resolution of the grid. Thus incremental clustering, iterative simplification and particle simulation need additional storage that is linearly proportional to the number of input points, while the storage overhead for hierarchical clustering depends only on the target model size.

2.4.5 Comparison to Mesh Simplification

Figure 2.12 shows a comparison of point-based simplification and simplification for polygonal meshes. In (a), the initial point cloud is simplified from 134,345 to 5,000 points using the iterative simplification method of Section 2.2. The resulting point cloud has then been triangulated using the surface reconstruction method of [GJ02]. In (b), the input point cloud has first been triangulated and the resulting polygonal surface has then been simplified using the mesh simplification tool QSlim [GH97]. Both methods produce similar results in terms of surface error and both simplification processes take approximately the same time (approx. 3.5 seconds). However, creating the triangle mesh from the simplified point cloud took 2.45 seconds in (a), while in (b) reconstruction time for the input point cloud was 112.8 seconds. Thus when given a large unstructured point cloud, it is much more efficient to first do the simplification on the point data and then reconstruct a mesh (if desired) than to first apply a reconstruction method and then simplify the triangulated surface. This illustrates that point-based simplification methods can be very useful when dealing with large geometric models stemming from 3D acquisition.



(a) point-based simplification (b) mesh-based simplification

Figure 2.12: Comparison of point-based (a) and mesh-based (b) surface simplification. Error images are shown on the left, final triangulated surfaces on the right.

Chapter 3

Shape Modeling

Mark Pauly

Modeling the shape of 3D objects is one of the fundamental techniques in digital geometry processing. In this chapter two fundamental modeling approaches for point-sampled geometry will be discussed: Boolean operations and free-form deformation [PKKG03]. While the former are concerned with building complex objects by combining simpler shapes, the latter defines a continuous deformation field in space to smoothly deform a given surface. Boolean operations are most easily defined on implicit surface representations, since the required inside-outside classification can be directly evaluated on the underlying scalar field. On the other hand, free-form deformation is a very intuitive modeling paradigm for explicit surface representations. For example, mesh vertices or NURBS control points can be directly displaced according to the deformation field. For point-based representations, the hybrid structure of the MLS surface model can be exploited to integrate these two modeling approaches into a unified shape modeling framework. Boolean operations can utilize the approximated signed distance function defined by the MLS projection (see Section 1.3) for inside-outside classification, while free-form deformations operate directly on the point samples.

3.1 Boolean Operations

A common approach in geometric modeling is to build complex objects by combining simpler shapes using boolean operations (see Figure 3.1). In constructive solid geometry (CSG) objects are defined using a binary tree, where each node corresponds to a union, intersection, or difference operation and each leaf stores a base shape. Operations such as ray-tracing, for example, are then implemented by traversing this tree structure. More commonly, surfaces are defined as boundary representations (BReps) of solids. Here boolean operations have to be evaluated explicitly, which requires an algorithm for intersecting two surfaces. Computing such a surface-surface intersection can be quite involved, however, in particular for higher order surfaces (see for example [KM97]).

As will be demonstrated below, the MLS projection operator can be used both for inside/outside classification as well as for explicitly sampling the intersection curve. The goal is to perform a boolean operation on two orientable, closed surfaces S_1 and S_2 that are represented by two point clouds P_1 and P_2, to obtain a new point cloud P_3 that defines the resulting surface S_3. P_3 consists of two subsets $Q_1 \subseteq P_1$ and $Q_2 \subseteq P_2$ plus a set of newly generated sample points that explicitly represent the intersection curves. Thus in order to perform a boolean operation for point-sampled geometry, the following techniques are required:

- a classification predicate to determine the two sets Q_1 and Q_2,
- an algorithm to find samples on the intersection curve, and

- a rendering method that allows to display crisp features curves using point primitives.

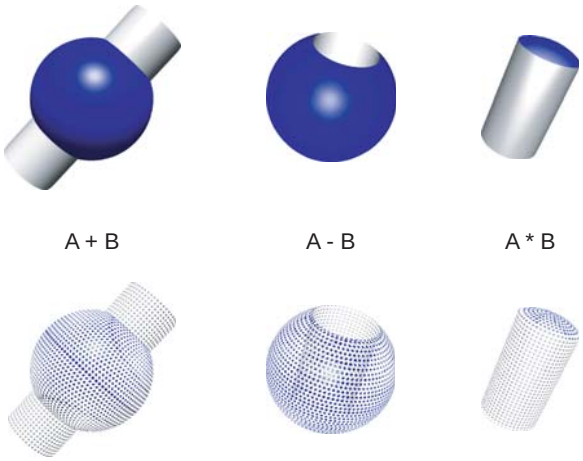


Figure 3.1: Boolean operations of a sphere A and a cylinder B. The bottom row illustrates the sampling distribution.

3.1.1 Classification

The goal of the classification stage is to determine which $\mathbf{p} \in P_1$ are inside or outside the volume enclosed by the surface S_2 and vice versa. For this purpose a classification predicate Ω_P is defined such that for $\mathbf{x} \in \mathbb{R}^3$

$$\Omega_P(\mathbf{x}) = \begin{cases} 1 & \mathbf{x} \in V \\ 0 & \mathbf{x} \notin V, \end{cases} \tag{3.1}$$

where V is the volume bounded by the MLS surface S represented by the point cloud P. Let $\mathbf{y} \in S$ be the closest point on S from $\mathbf{x}$. It is well-known from differential geometry that, if S is continuous and twice differentiable, the vector $\mathbf{x} - \mathbf{y}$ is aligned with the surface normal $\mathbf{n}_y$ at $\mathbf{y}$ [dC76]. If surface normals are consistently oriented to point outwards of the surface, then $(\mathbf{x} - \mathbf{y}) \cdot \mathbf{n}_y > 0$, if and only if $\mathbf{x} \notin V$. Since only a discrete sample P of the surface is given, the closest point $\mathbf{y}$ on S is replaced by the closest point $\mathbf{p} \in P$. Thus $\mathbf{x}$ is classified as outside,if $(\mathbf{x} - \mathbf{p}) \cdot \mathbf{n}_p > 0$, i.e., if the angle between $\mathbf{x} - \mathbf{p}$ and the normal $\mathbf{n}_p$ at is less than $\pi/2$ (see Figure 3.2, left image). This discrete test yields the correct inside/outside classification of the point $\mathbf{x}$, if the distance $\|\mathbf{x} - \mathbf{p}\|$ is larger than the local sample spacing η_p at $\mathbf{p}$. If $\mathbf{x}$ is extremely close to the surface, the classification could fail, as illustrated in the right image of Figure 3.2. In this case the exact closest point $\mathbf{y} \in S$ is approximated using the MLS projection. Since for classification only an inside/outside test is of interest, the performance can be significantly improved by exploiting local coherence: $\Omega_P(\mathbf{x}) = \Omega_P(\mathbf{x}')$ for all points $\mathbf{x}'$ that lie in the sphere around $\mathbf{x}$ with radius $\|\mathbf{x} - \mathbf{p}\| - \eta_p$. Thus the number of closest point queries and MLS projections can be reduced drastically, in practice up to 90 percent.

Given the classification predicate Ω, the subsets Q_1 and Q_2 can be computed as shown in Table 3.1. As Figure 3.3 illustrates, the resulting inside/outside classification is very robust and easily handles complex, non-convex surfaces. Observe that boolean operations can create a large number of disconnected components, i.e., can lead to a significant change in genus.

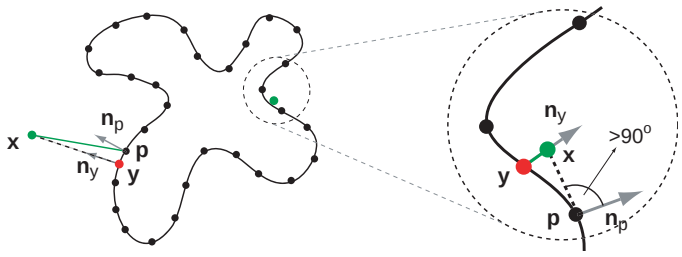


Figure 3.2: Inside/outside test. For $\mathbf{x}$ very close to the surface, the closest point $\mathbf{p} \in P$ can yield a false classification (right image). In this case, $\mathbf{x}$ is classified according to its MLS projection $\mathbf{y}$.

	Q_1	Q_2
$S_1 \cup S_2$	$\{\mathbf{p} \in P_1 \| \Omega_{P_2}(\mathbf{p}) = 0\}$	$\{\mathbf{p} \in P_2 \| \Omega_{P_1}(\mathbf{p}) = 0\}$
$S_1 \cap S_2$	$\{\mathbf{p} \in P_1 \| \Omega_{P_2}(\mathbf{p}) = 1\}$	$\{\mathbf{p} \in P_2 \| \Omega_{P_1}(\mathbf{p}) = 1\}$
$S_1 - S_2$	$\{\mathbf{p} \in P_1 \| \Omega_{P_2}(\mathbf{p}) = 0\}$	$\{\mathbf{p} \in P_2 \| \Omega_{P_1}(\mathbf{p}) = 1\}$
$S_2 - S_1$	$\{\mathbf{p} \in P_1 \| \Omega_{P_2}(\mathbf{p}) = 1\}$	$\{\mathbf{p} \in P_2 \| \Omega_{P_1}(\mathbf{p}) = 0\}$

Table 3.1: Classification for Boolean operations.

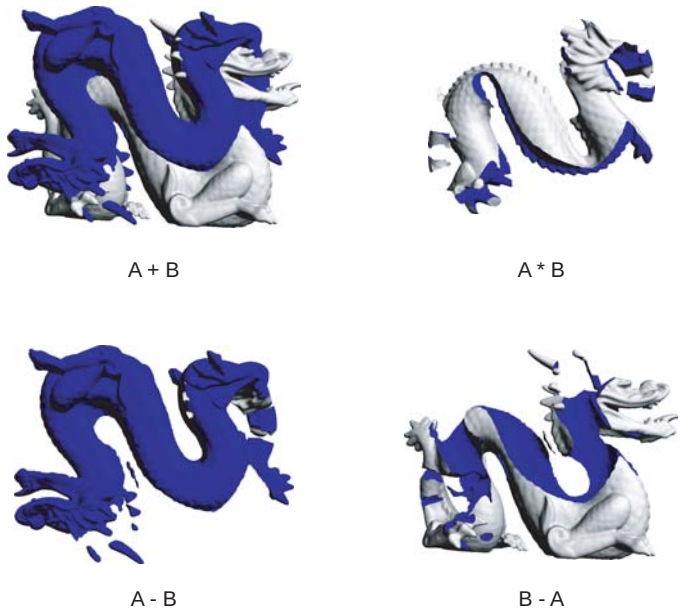


Figure 3.3: Boolean operations of a blue dragon (A) and a white dragon (B).

3.1.2 Intersection Curves

Taking the union of Q_1 and Q_2 will typically not produce a point cloud that accurately describes the surface S_3, since the intersection curve of the two MLS surfaces S_1 and S_2 is not represented adequately. Therefore, a set of sample points that lie on the intersection curve is explicitly computed and added to $Q_1 \cup Q_2$, to obtain the point cloud P_3. First, all points in Q_1 and Q_2 are found that are close to the

intersection curve by evaluating the distance function induced by the MLS projection operator. From all closest pairs ($\mathbf{q}_1 \in Q_1, \mathbf{q}_2 \in Q_2$) of these points a point $\mathbf{q}$ on the intersection curve is computed using a Newton-type iteration. This is done as follows (see Figure 3.4 (a-d)): Let $\mathbf{r}$ be the point on the intersection line of the two tangent planes of $\mathbf{q}_1$ and $\mathbf{q}_2$ that is closest to both points, i.e., that minimizes the distance $\|\mathbf{r}-\mathbf{q}_1\|+\|\mathbf{r}-\mathbf{q}_2\|$. $\mathbf{r}$ is the first approximation of $\mathbf{q}$ and can now be projected onto S_1 and S_2 to obtain two new starting points $\mathbf{q}_1'$ and $\mathbf{q}_2'$ for the iteration. This procedure can be repeated iteratively until the points $\mathbf{q}_1$ and $\mathbf{q}_2$ converge to a point $\mathbf{q}$ on the intersection curve. Due to the quadratic convergence of the Newton iteration, this typically requires less than three iterations. The sampling density estimation of Section 1.1 is used to detect whether the sampling resolution of

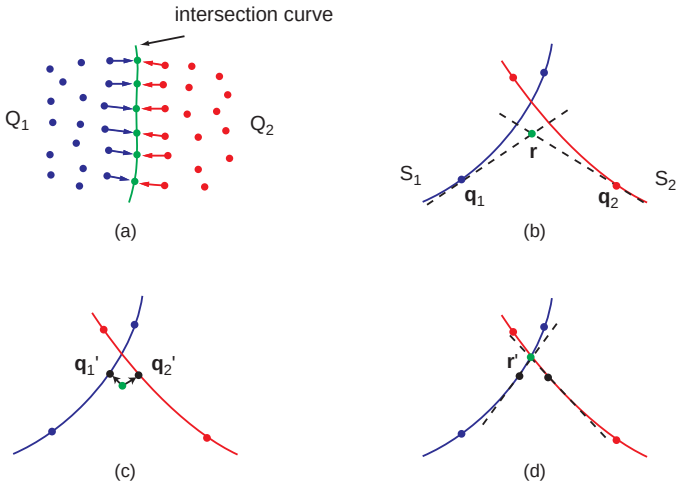


Figure 3.4: Sampling the intersection curve. (a) closest pairs of points in Q_1 and Q_2, (b) first estimate $\mathbf{r}$, (c) re-projection, (d) second estimate $\mathbf{r}'$.

the two input surfaces differs significantly in the vicinity of the intersection curve. To avoid a sharp discontinuity in sampling density, the coarser model is up-sampled in this area to match the sampling density of the finer model, using the dynamic sampling method of Section 3.2.2.

Note that the above Newton scheme also provides an easy mechanism for adaptively refining the intersection curve. A simple subdivision rule is evaluated to create a new starting point for the Newton iteration, e.g., the average of two adjacent points on the curve. Applying the iteration then yields a new sample on the intersection curve (see Figure 3.5).

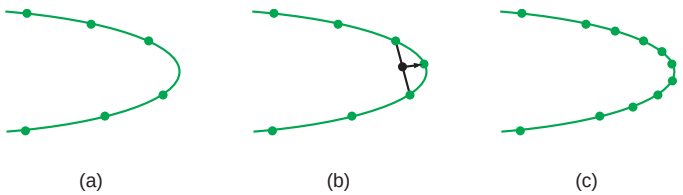


Figure 3.5: Adaptive refinement. (a) original intersection curve, (b) new point inserted in region of high curvature, (c) final, adaptively sampled intersection curve.

3.1.3 Rendering Sharp Creases

The accurate display of the intersection curves requires a rendering technique that can handle sharp creases and corners. For this purpose an extension of the surface splatting technique presented in [ZPvG01] is used [ZRB*04]. In this method, each sample point is represented by a *surfel*, an oriented elliptical splat that is projected onto the screen to reconstruct the surface in image space. A point on the intersection curve can now be represented by two surfels that share the same center, but whose normals stem from either one of the two input surfaces. During scan-conversion, each of these surfels is then clipped against the plane defined by the other to obtain a piecewise linear approximation of the intersection curve in screen space (see Figure 3.6). This concept can easily be generalized to handle corners as shown in Figure 3.6 (e).

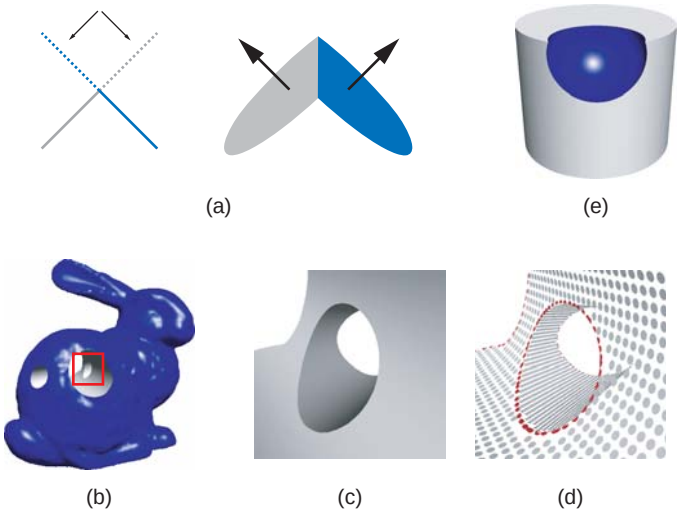


Figure 3.6: Rendering the intersection curve. (a) mutual clipping of two surfels on the intersection curve, (b) boolean differences on the bunny model, (c) zoom of the intersection curves, (d) sampling distribution, where samples on the intersection curve are rendered using two red half ellipses, (e) an example of a corner.

Figure 3.7 shows an example of a difficult boolean operation of two identical cylinders that creates two singularities. While the classification and intersection curve sampling work fine, the rendering method produces artifacts. This is due to numerical instabilities, since the clipping planes of two corresponding surfels are almost parallel. However, such cases are rare in typical computer graphics applications, e.g., digital character design. As such, the algorithms for boolean operations are less suited for industrial manufacturing applications, where robust handling of degenerated cases is of primary concern.

3.1.4 Particle-Based Blending

As illustrated in Figure 3.1, boolean operations typically produce sharp intersections. In some applications it is more desirable to create a smooth blend between the two combined surface parts. To smooth out the sharp creases created by boolean operations an adaptation of oriented particles [ST92] can be used. The idea is to define inter-particle potentials in such a way that the minimum of the global potential function yields a smooth surface that minimizes curvature. Summing up these potentials yields a particle's total potential energy. From this potential energy one can derive the positional and rotational forces that are exerted on each particle and compute its path of motion under these forces.

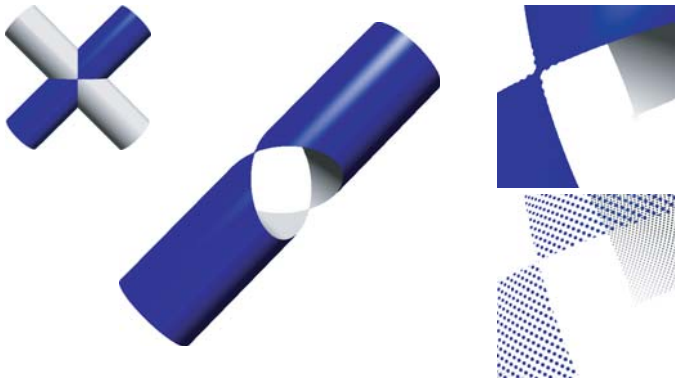


Figure 3.7: A difficult boolean difference operation that creates two singularities.

Additionally, an inter-particle repulsion force is applied to equalize the particle distribution (see also Section 2.3). All forces are scaled with a smooth fall-off function that measures the distance to the intersection curve to confine the particle simulation to a small area around the intersection curve without affecting other parts of the surface.

Figure 3.8 shows the particle-based blending for the intersection of two planes, where the degree of smoothness can be controlled by the number of iterations of the simulation. In Figure 3.9, a more complex blending operation is shown. A union operation of three tori has created numerous sharp intersection curves as shown in (a). These can be blended simultaneously as illustrated in (b) using the particle simulation described above. The same blending technique can of course also be applied to the intersection and difference operations described in Section 3.1.

3.2 Free-Form Deformation

Apart from composition of surfaces using boolean operations, many shape design applications require the capability to modify objects using smooth deformations [Bar84, SP86]. These include bending, twisting, stretching, and compressing of the model surface. For this purpose a point-based free-form deformation tool is described that allows the user to interactively deform a surface by specifying a smooth deformation field [PKKG03].

The user first defines a deformable region $\chi_d \subset S$ on the model surface and marks parts of this region as a control handle. The surface can then be modified by pushing, pulling or twisting this handle. These user interactions are translated into a continuous tensor-field, which for each point in the deformable region defines a translatory and rotational motion under which the surface deforms. The tensor-field is based on a continuously varying scale parameter $t \in [0, 1]$ that measures the relative distance of a point from the handle. The closer a point is to the handle, the stronger will the deformation be for that point. More precisely, let $\chi_1 \subset \chi_d$ be the handle, also called *one-region*, and $\chi_0 = S - \chi_d$ the *zero-region*, i.e., all points that are not part of the deformable region. For both zero- and one-region distance measures d_0 and d_1,respectively, are defined as

$$d_j(\mathbf{p}) = \begin{cases} 0 & \mathbf{p} \in \chi_j \\ \min_{\mathbf{q} \in \chi_j} \|\mathbf{p} - \mathbf{q}\| & \mathbf{p} \notin \chi_j \end{cases} \tag{3.2}$$

for $j = 0, 1$. From these distance measures the scale parameter t is computed as $t = \beta(d_0(\mathbf{p})/(d_0(\mathbf{p}) + d_1(\mathbf{p})))$, where $\beta : [0, 1] \to [0, 1]$ is a continuous blending function with $\beta(0) = 0$ and $\beta(1) = 1$. Thus $t = 0$ for $\mathbf{p} \in \chi_0$ and $t = 1$ for $\mathbf{p} \in \chi_1$. Using this scale parameter, the position of a point $\mathbf{p} \in \chi_d$ after the

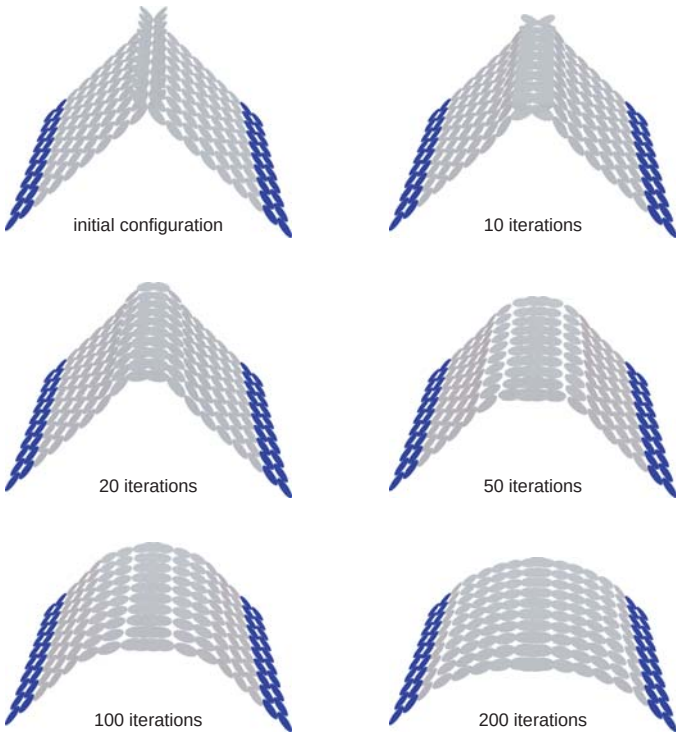


Figure 3.8: Particle simulation to blend two intersecting planes. Gray particles participate in the simulation, blue points indicate the fixed boundary.

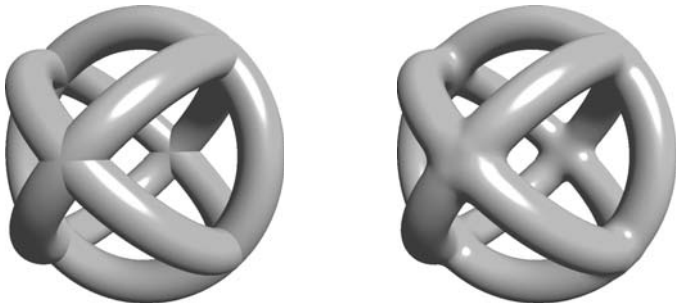


Figure 3.9: Boolean union of three tori. Left: Reconstruction with sharp feature curves, right: Feature curves have been blended using particle simulation.

deformation is determined as $\mathbf{p}' = F(\mathbf{p}, t)$, where F is a deformation function composed of a translatory and a rotational part. The deformation function can be written as $F(\mathbf{p}, t) = F_T(\mathbf{p}, t) + F_R(\mathbf{p}, t)$, where

- $F_T(\mathbf{p}, t) = \mathbf{p} + t \cdot \mathbf{v}$ with $\mathbf{v}$ a translation vector and
- $F_R(\mathbf{p}, t) = R(\mathbf{a}, t \cdot \alpha) \cdot \mathbf{p}$, where $R(\mathbf{a}, \alpha)$ is the matrix that specifies a rotation around axis $\mathbf{a}$ with angle α.

Figure 3.10 shows a translatory deformation of a plane where the translation vector $\mathbf{v}$ is equal to the plane normal. This figure also illustrates the effect of different choices of the blending function

β. In Figure 3.11, two rotational deformations of a cylinder are shown, while a combination of both translatory and rotational deformations is illustrated in Figure 3.16.

To perform a free-form deformation the user only needs to select the zero- and one-regions and choose an appropriate blending function. She can then interactively deform the surface by displacing the handle with a mouse or trackball device, similar to [KCVS98]. This gives the method great flexibility for handling a wide class of free-form deformations, while still providing a simple and intuitive user interface. The deformable region and the handle can be specified using a simple paint tool that allows the user to mark points on the surface by drawing lines, circles, rectangles, etc. and applying flood filling and erasure. The system also supports pre-defined and user-editable selection stencils, which can be used to create embossing effects (Figure 3.12).

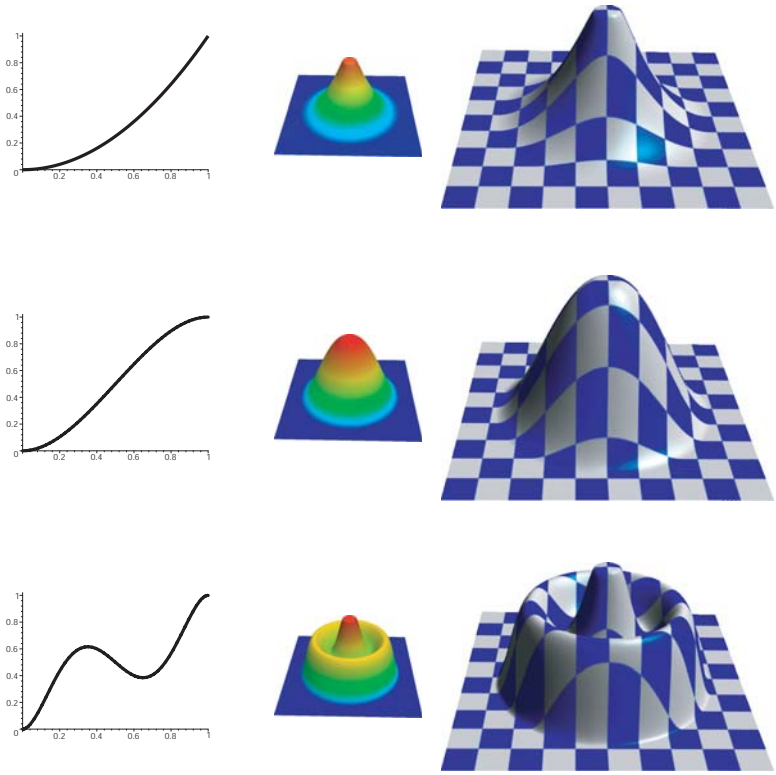


Figure 3.10: Deformations of a plane for three different blending functions. Left: Blending function, middle: Color-coded scale parameter, where blue indicates the zero region ($t = 0$) and red the one-region ($t = 1$), right: Final textured surface.

3.2.1 Topology Control

An important issue in shape design using free-form deformation is the handling of self-intersections. During deformation, parts of the deformable region can intersect other parts of the surface, which leads to an inconsistent surface representation. A solution to this problem requires a method for detecting and resolving such collisions.

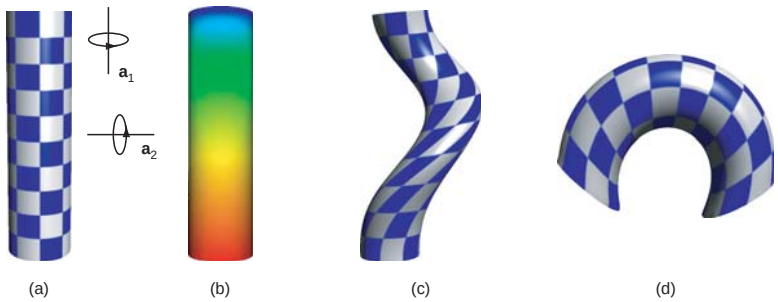


Figure 3.11: Rotational deformations of a cylinder. (a) original, (b) color-coded scale parameter, (c) rotation around axis $\mathbf{a}_1$, (d) rotation around axis $\mathbf{a}_2$.

Figure 3.12: Embossing effect. The Siggraph label on the top left has been converted to a selection stencil using simple image processing tools. This stencil can then be mapped to a surface, where blue color corresponds to the zero-region and red to the one-region. Subsequent deformation yields an embossing effect as shown on the right.

Collision Detection

Similar to boolean operations, this requires an inside/outside classification to determine which parts of the surface have penetrated others. Thus the classification predicate defined in Section 3.1 can be used for this purpose. First, the closest point $\mathbf{p} \in \chi_0$ to each sample point $\mathbf{p} \in \chi_d$ is computed. This defines an empty sphere s_p around $\mathbf{p}$ with radius $\|\mathbf{p} - \mathbf{q}\|$. If the point $\mathbf{p}$ only moves within this sphere during deformation, it is guaranteed not to intersect with the zero-region (see Figure 3.13). So additionally to exploiting spatial coherence as for boolean classification, this approach also exploits the temporal coherence induced by the smooth deformation field. The classification predicate Ω has to be re-evaluated only when $\mathbf{p}$ leaves s_p, which at the same time provides a new estimate for the updated sphere s_p.

Collision Handling

There are different ways to respond to a detected collision. The simplest solution is to undo the last deformation step and recover the surface geometry prior to the collision. Alternatively, the penetrating parts of the surface can be joined using a boolean union operation to maintain the validity of the surface.

Figure 3.14 shows an editing session, where a deformation causes a self-intersection. After performing a boolean union, a sharp intersection curve is created as shown in (d). In the context of free-form deformation it is often more desirable to create a smooth transition between the two combined surface parts. Thus the particle simulation described in Section 3.1.4 can be used to blend the intersection region.

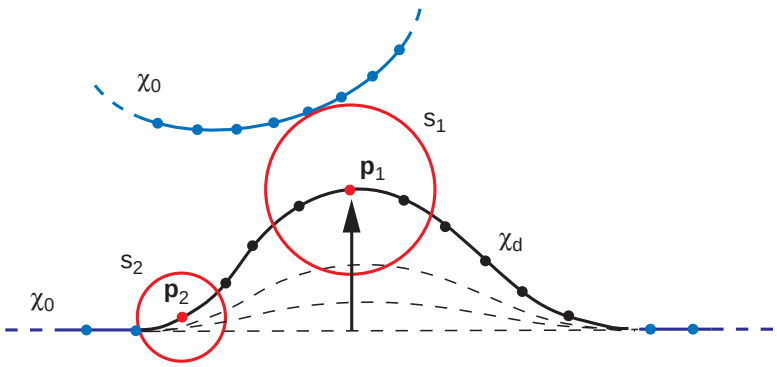


Figure 3.13: Temporal coherence for collision detection during deformation. The points $\mathbf{p}_1$ and $\mathbf{p}_2$ can move with the spheres s_1 and s_2, resp., without intersecting the zero-region.

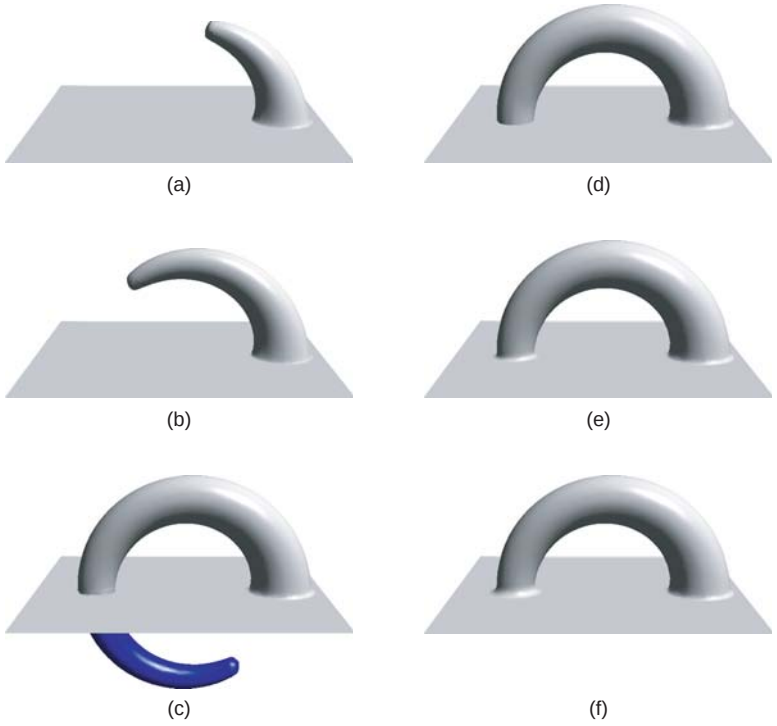


Figure 3.14: Interactive modeling session with collision detection. (a - b) intermediate steps of the deformation, (c) collision detection, where the blue part has been detected as self-intersecting, (d), boolean union with sharp intersection curve, (e - f), particle-based blending with different fall-off functions.

3.2.2 Dynamic Sampling

Large deformations may cause strong distortions in the distribution of sample points on the surface that can lead to an insufficient local sampling density. To prevent the point cloud from ripping apart and maintain a high surface quality, new samples have to be included where the sampling density

becomes too low. This requires a method for measuring the surface stretch to detect regions of insufficient sampling density. Then new sample points have to be inserted and their position on the surface determined. Additionally, scalar attributes, e.g., color values or texture coordinates, have to be preserved or interpolated.

Measuring Surface Stretch

The first fundamental form known from differential geometry [dC76] can be used to measure the local distortion of a surface under deformation. Let $\mathbf{u}$ and $\mathbf{v}$ be two orthogonal tangent vectors of unit length at a sample point $\mathbf{p}$. The first fundamental form at $\mathbf{p}$ is defined by the 2×2 matrix

$$\begin{bmatrix} \mathbf{u}^2 & \mathbf{u} \cdot \mathbf{v} \\ \mathbf{u} \cdot \mathbf{v} & \mathbf{v}^2 \end{bmatrix} \tag{3.3}$$

The eigenvalues of this matrix yield the minimum and maximum stretch factors and the corresponding eigenvectors define the principal directions into which this stretching occurs. When applying a deformation, the point $\mathbf{p}$ is shifted to a new position $\mathbf{p}'$ and the two tangent vectors are mapped to new vectors $\mathbf{u}'$ and $\mathbf{v}'$. Local stretching implies that $\mathbf{u}'$ and $\mathbf{v}'$ might no longer be orthogonal to each other nor do they preserve their unit length. The amount of this distortion can be measured by taking the ratio of the two eigenvalues of Equation 3.3 (local anisotropy) or by taking their product (local change of surface area). When the local distortion becomes too strong, new samples have to be inserted to re-establish the prescribed sampling density. Since Equation 3.3 defines an ellipse in the tangent plane centered at $\mathbf{p}$ with the principal axes defined by the eigenvectors and eigenvalues, $\mathbf{p}$ can be replaced by two new samples $\mathbf{p}_1$ and $\mathbf{p}_2$ that are positioned on the main axis of the ellipse (see Figure 3.15).

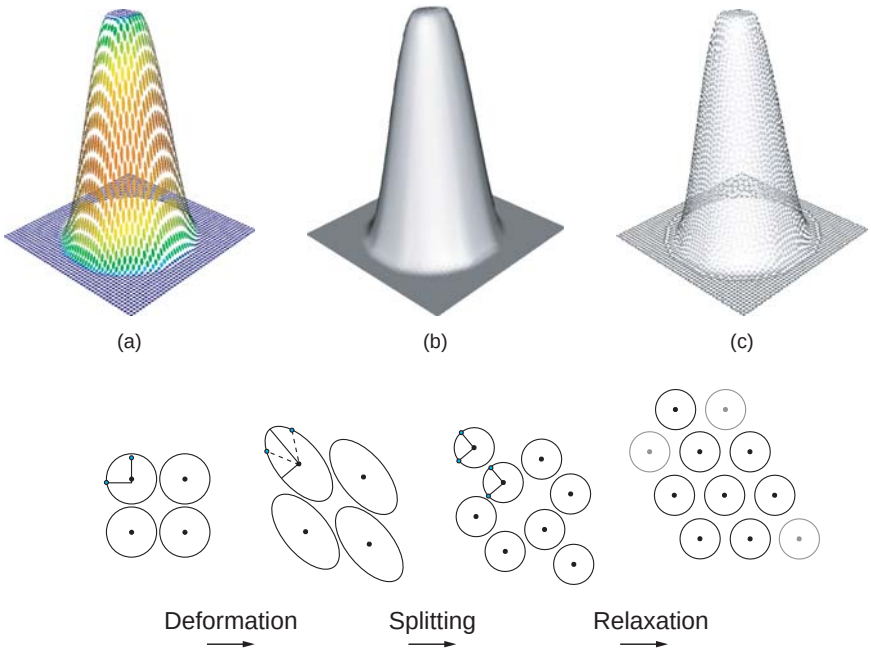


Figure 3.15: Dynamic sampling. Top row: Deformation of a plane. (a) local stretching: blue corresponds to zero stretch, while red indicates maximum stretch, (b) surface after re-sampling, (c) sampling distribution. Bottom row: illustration of point insertion.

3.2.3 Filter Operations

Whenever a splitting operation is applied, both the geometric position and the scalar function values for the newly generated sample points have to be determined. Both these operations can be described as the application of a filtering operator: A *relaxation filter* determines the sample positions while an *interpolation filter* is applied to obtain the function values.

Relaxation

Introducing new sample points through a splitting operation creates local imbalances in the sampling distribution. To obtain a more uniform sampling pattern, a relaxation operator is applied that moves the sample points within the surface (see Figure 3.15). Similar to [Tur92] (see also Section 2.3) a simple point repulsion scheme is used with a repulsion force that drops linearly with distance. This confines the radius of influence of each sample point to its local neighborhood, which allows very efficient computation of the relaxation forces. The resulting displacement vector is then projected into the points tangent plane to keep the samples on the surface.

Interpolation

Once the position of a new sample point $\mathbf{p}$ is fixed using the relaxation filter, the associated function values need to be determined. This can be achieved using an interpolation filter by computing a local average of the function values of neighboring sample points. The relaxation filter potentially moves all points of the neighborhood of $\mathbf{p}$. This tangential drift leads to distortions in the associated scalar functions. To deal with this problem a copy of each point that carries scalar attributes is created and its position is fixed during relaxation. In particular, for each sample that is split a copy is maintained with its original data. These points will only be used for interpolating scalar values, they are not part of the current geometry description. Since these samples are dead but their function values still live, they are called *zombies*. Zombies will undergo the same transformation during a deformation operation as living points, but their positions will not be altered during relaxation. Thus zombies accurately describe the scalar attributes without distortions. Therefore, zombies are only used for interpolation, while for relaxation only living points are considered. After an editing operation is completed, all zombies will be deleted from the representation.

Figure 3.16 illustrates this dynamic re-sampling method for a very large deformation that leads to a substantial increase in the number of sample points. While the initial plane consists of 40,000 points, the final model contains 432,812 points, clearly demonstrating the robustness and scalability of the method in regions of extreme surface stretch.

3.2.4 Down-Sampling

Apart from lower sampling density caused by surface stretching, deformations can also lead to an increase in sampling density, where the surface is squeezed. It might be desirable to eliminate samples in such regions while editing, to keep the overall sampling distribution uniform. However, dynamically removing samples also has some drawbacks. Consider a surface that is first squeezed and then stretched back to its original shape. If samples get removed during squeezing, surface information such as color will be lost, which leads to increased blurring when the surface is re-stretched. Thus instead of dynamic sample deletion, an optional garbage collection is performed at the end of the editing operation. To reduce the sampling density, any of the simplification methods of Chapter 2 can be used.

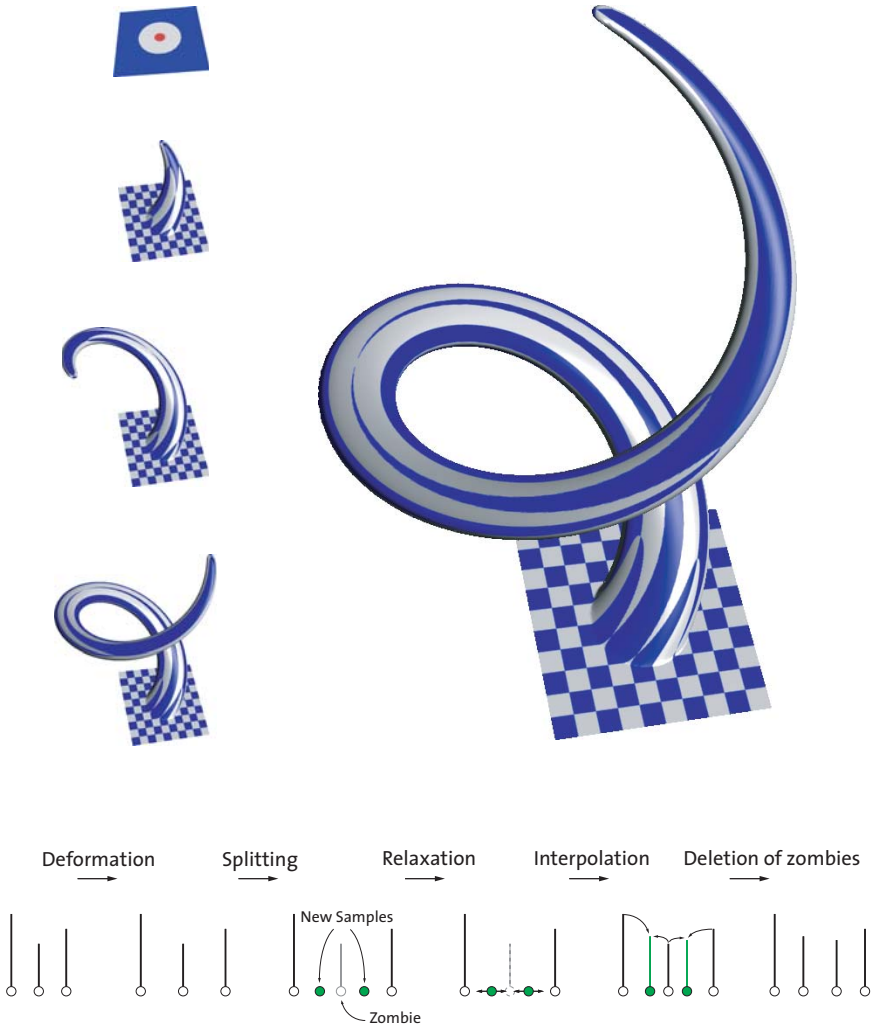


Figure 3.16: A very large deformation using a combination of translatory and rotational motion. The left column shows intermediate steps with the top image indicating zero- and one-regions. Each point of the surface carries texture coordinates, which are interpolated during re-sampling and used for texturing the surface with a checkerboard pattern. The bottom row illustrates this interpolation process, where the function values are indicated by vertical lines.

Chapter 4

Appearance Modeling

Mark Pauly

The previous chapter has discussed boolean operations and free-form deformation for point-based shape modeling. Apart from the mere geometric shape, which is described by the position of the sample points in space, 3D objects also carry a number of additional attributes, such as color or material properties, that determine the overall appearance of the surface. In this chapter, a number of tools and algorithms for interactively editing surface appearance attributes will be discussed [ZPKG02]. These methods can be understood as a generalization of common photo editing techniques from 2D images to 3D point-sampled surfaces.

4.1 Overview

This section gives an overview of the appearance modeling functionality for point-sampled models implemented in Pointshop3D (www.pointshop3d.com) by defining a surface editing operation on an abstract level. A brief analysis of 2D photo editing identifies three fundamental building blocks of an interactive editing operation: Parameterization, re-sampling, and editing. It will be shown that these concepts can be extended from 2D photo editing to 3D surface editing. For this purpose an operator notation will be introduced that allows a wide variety of editing operations to be defined in a unified and concise way. The fundamental differences between functional images and manifold surfaces lead to different implementations of these operators, however.

4.1.1 2D Photo Editing

A 2D image I can be considered as a discrete sample of a continuous image function containing image attributes such as color or transparency. Implicitly, the discrete image always represents a continuous image, yet image editing operations are typically performed directly on the discrete samples. A general image editing operation can be described as a function of a given image I and a brush image B. The brush image is used as a generic tool to modify the image I. Depending on the considered operation it may be interpreted as a paint brush or a discrete filter, for example.

The editing operation involves the following steps: First, a parameter mapping Φ has to be found that aligns the image I with the brush B. For example, Φ can be defined as the translation that maps the pixel at the current mouse position to the center of B. Next, a common sampling grid for I and B has to be established, such that there is a one-to-one correspondence between the discrete samples. This requires a re-sampling operator Ψ that first reconstructs the continuous image function and then samples this function on the common grid. Finally, the editing operator Ω combines the image samples with the brush samples using the one-to-one correspondence established before. The

resulting discrete image I' is then obtained using a concatenation of the operators described above as $I' = \Omega(\Psi(\Phi(I)), \Psi(B))$. The goal is to generalize this operator framework to irregular point-sampled surfaces, as illustrated in Figure 4.1. Formally, this can be done by replacing the discrete image I by a point cloud P that represents a surface S. The discrete sample points can be considered as a direct extension of image pixels, carrying the attributes shown in Figure 4.4 (a). This motivates the term *surfel*, short for *surface element*, similar to *pixel*, which stands for *picture element* (see also [ZPvG01]). The transition from image to surface has the following effects on the individual terms of the operator notation introduced above:

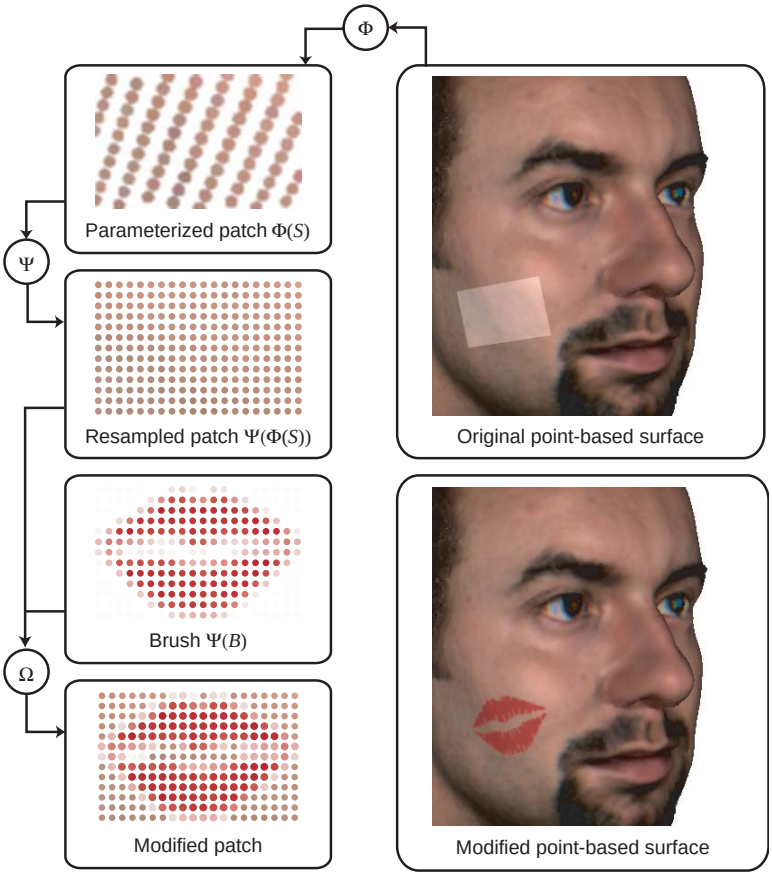


Figure 4.1: Overview of the operator framework for point-based surface editing.

Parameterization Φ

For photo editing, the parameter mapping Φ is usually specified by a simple, global 2D to 2D affine mapping, i.e., a combination of translation, scaling, and rotation. Mapping a manifold surface onto a 2D domain is much more involved, however. Therefore, the user interactively selects subsets, or *patches*, of S that are parameterized individually. In general, such a mapping leads to distortions that cannot be avoided completely. Section 4.2 will describe two algorithms for computing a parameterization that correspond to two different interaction schemes: Parameterization by orthogonal projection for interactive brush painting, and a method to compute a constrained minimum distortion parameterization. The latter allows the user to control the mapping in an intuitive manner by setting corresponding

feature points both on the parameter plane and the surface, respectively.

Re-sampling Ψ

Images are usually sampled on a regular grid, hence signal processing methods can be directly applied for re-sampling. However, the sampling distribution of surfaces is in general irregular, requiring alternative methods for reconstruction and sampling. For this purpose a parameterized scattered data approximation can be used that reconstructs a continuous function from the samples (see [ZPKG02]). This continuous function can then be evaluated at the desired sampling positions.

Editing Ω

Once the parameterization is established and re-sampling has been performed, all computations take place on the discrete samples in the 2D parameter domain. Hence the full functionality of photo editing systems can be applied for texturing and texture filtering. Additionally, operations that modify the geometry, e.g., sculpting or geometry filtering, can easily be incorporated into the system. As will be described in Section 4.1.3, all of these tools are based on the same simple interface that specifies an editing tool by a set of bitmaps and few additional parameters. For example, a sculpting tool is defined by a 2D displacement map, an alpha mask and an intrusion depth.

4.1.2 Interaction Modes

The appearance attributes of a point-sampled model can be manipulated using two different interaction schemes:

Brush Interaction

In this interaction mode the user moves a brush device over the surface and continuously triggers editing events, e.g., painting operations (see Figure 4.2). The brush is positioned using the mouse cursor and aligned with the surface normal at the current interaction point. This means that the parameterization is continuously and automatically re-computed and re-sampling is performed for each editing event. A complete editing operation is then performed using a fixed brush image.

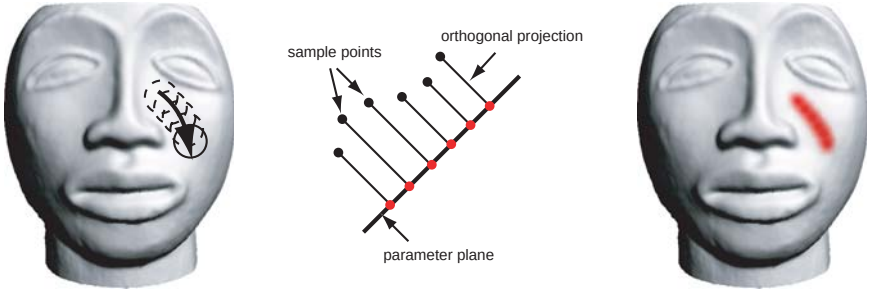


Figure 4.2: Brush interaction: Left: Cursor movement, middle: Parameterization by orthogonal projection onto the brush plane (2D for illustration), right: Painted surface.

Selection Interaction

Here the user first selects a subset of the surface called a patch and defines the parameter mapping interactively by imposing point constraints (see Figure 4.3). Based on this fixed parameterization,

various editing operations can be applied. Hence parameterization and re-sampling operators are evaluated once, while different editing operators can be applied successively.

Figure 4.3: Selection Interaction: Left: Feature points on parameter plane, middle: Feature points on surface, right: Final texture-mapped surface.

4.1.3 Brush Interface

All appearance modeling operations are based on a generic brush interface (see also Figure 4.1). A brush is defined as a $M \times N$ grid B, where each grid point $\mathbf{b}_{mn}$ stores all the surface appearance attributes shown in Figure 4.4 (a). Each individual bitmap, e.g., the diffuse color image $\{\mathbf{c}_{mn}\}$, defines a *brush channel* that represents a corresponding continuous attribute function, similar to the surface samples P that represent the continuous surface S. Additionally, each brush channel carries an alpha mask that can be used for blending the brush coefficients with the surface coefficients as described in Section 4.4. The channel $\{d_{mn}\}$ defines a bitmap of displacement coefficients that can be used for sculpting operations, such as normal displacement or carving. Figure 4.4 shows a typical brush that combines texture, geometry and material properties to support flexible editing operations.

4.2 Parameterization

This section describes two different methods to compute a parameterization for a point-sampled surface that correspond to the two interaction schemes defined above. For brush interaction the parameter mapping will be computed by a simple orthogonal projection, while an optimization method is applied for computing a constrained minimum distortion parameterization for selection interactions (see also [ZPKG02]). To define the parameterization Φ, the user first selects a subset S' of the surface S, described by a point cloud $P' \subseteq P$. A mapping $\Phi : P' \rightarrow [0,1] \times [0,1]$ is then computed that assigns parameter coordinates $\mathbf{u}_i$ to each point $\mathbf{p}_i \in P'$.

4.2.1 Orthogonal Projection

A simple method for computing a parameter mapping is dimension reduction. 2D parameter values for a point $\mathbf{p}_i \in P'$ can be obtained by simply discarding one of the three spatial coordinates. With an appropriate prior affine transformation, this amounts to an orthogonal projection of the sample points onto a plane. This plane can either be specified by the user, or computed automatically according to the distribution of the sample points, e.g., as a least-squares fit. Using covariance analysis (see Section 1.2), the normal vector of the parameter plane would then be chosen as the eigenvector of the covariance matrix with smallest corresponding eigenvalue. Figure 4.5 shows examples of texture-mapping operations using a parameterization obtained by orthogonal projection. In general, such a mapping will exhibit strong distortions and discontinuities, leading to inferior editing results. However,

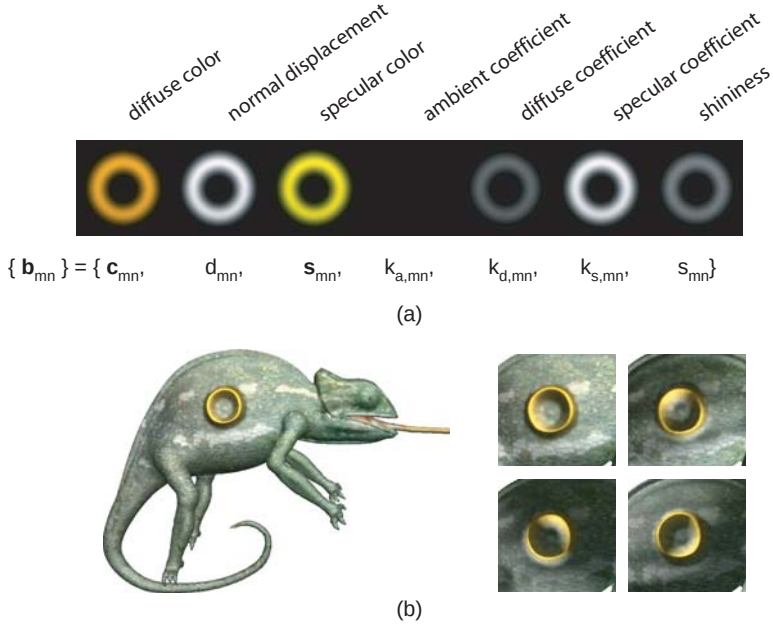


Figure 4.4: Brush interface: (a) a brush specified by a set of bitmaps, (b) the brush applied to a surface. The zooms on the right show the surface under different illumination to illustrate how the reflectance properties have been modified.

if the surface patch is sufficiently small, distortions will be small too and no discontinuities will occur. Thus orthogonal projection is a suitable parameterization method for brush interactions, where the parameter plane is defined by the surface normal at the tool cursor and the surface patch is defined by the projection of the brush onto the surface, as shown in Figure 4.2.

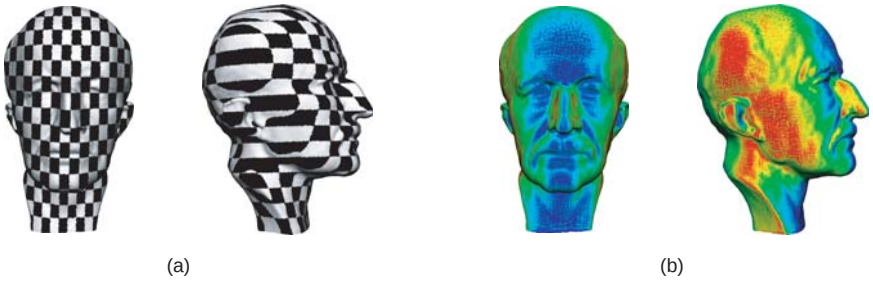


Figure 4.5: The Max Planck model parameterized by orthogonal projection from the front. (a) texture mapped surface, (b) the color-coded first derivative of the parameterization measures the stretch of the mapping, where blue corresponds to minimum, red to maximum stretch.

4.2.2 Constrained Minimum Distortion Parameterization

As Figure 4.5 illustrates, orthogonal projection leads to strong distortions in the parameterization. Furthermore, it provides little support for interactive control of the mapping by the user. Consider the

typical texture mapping operation shown in Figure 4.3. The goal is to map a 2D image of a human face onto a laser-range scan of a different face. It is certainly desirable to minimize the distortion of the mapping. Equally important, however, is a good correspondence of feature points, e.g., the tip of the nose in the image should be mapped onto the tip of the nose on the surface. Thus some mechanism is needed that allows the user to define corresponding feature points both in the image and on the surface. These point-to-point correspondences are then incorporated as constraints into an optimization that computes the mapping [Lev01]. First, an objective function for a continuous surface patch is defined that penalizes high distortion as well as the approximation error of the feature point constraints. A suitable discretization then yields a system of linear equations that can be solved using conjugate gradient methods.

Objective Function

A continuous parameterized surface S_X can be defined by a mapping

$$X : [0,1] \times [0,1] \rightarrow S_X \subset \mathbb{R}^3,$$

which for each parameter value $\mathbf{u} = (u, v) \in [0,1] \times [0,1]$ determines a point $\mathbf{x} = X(\mathbf{u}) \in S_X$ on the surface S_X. The mapping X defines a parameterization of the surface S_X. Let $U = X^{-1}$ be the inverse mapping, i.e., a function that assigns parameter coordinates $\mathbf{u}$ to each point $\mathbf{x} \in S_X$. The distortion of the parameter mapping can be measured using the cost function

$$C_{dist}(X) = \int_H \gamma(\mathbf{u}) d\mathbf{u},$$

where $H = [0,1] \times [0,1]$,

$$\gamma(\mathbf{u}) = \int_\theta \left(\frac{\partial^2}{\partial r^2} X_{\mathbf{u}}(\theta, r) \right)^2 d\theta,$$

and

$$X_{\mathbf{u}}(\theta, r) = X\left(\mathbf{u} + r \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \right).$$

$\gamma(\mathbf{u})$ is defined as the integral of the squared second derivative of the parameterization in each radial direction at a parameter value $\mathbf{u}$ using a local polar re-parameterization $X_{\mathbf{u}}(\theta, r)$. If $\gamma(\mathbf{u})$ vanishes, the parameterization is arc length preserving, i.e., defines a polar geodesic map at $\mathbf{u}$. Additionally, a set M of point-to-point correspondences can be specified such that a point $\mathbf{p}_j$ of the point cloud corresponds to a point $\mathbf{u}_j$ in the parameter domain. These point pairs serve as constraints that are approximated in a least-squares sense using the cost function

$$C_{fit}(X) = \sum_j (X(\mathbf{u}_j) - \mathbf{p}_j)^2.$$

The two cost functions C_{dist} and C_{fit} can be combined into the objective function $C(X) = C_{fit}(X) + \beta \cdot C_{dist}(X)$, where β is an additional parameter that allows to control the relative weight of the fitting error and distortion measure. This derivation of the objective function follows Levy's method for triangle meshes [Lev01]. By replacing the mesh connectivity with a point neighborhood relation as defined in Section 1.1, a discrete formulation of the objective function can be derived for point-sampled surfaces. This requires a discretization of the directional derivatives in $\gamma(\mathbf{u})$, which can be obtained using divided differences on a discrete set of normal sections. For a complete derivation, the reader is referred to [ZPKG02]. When substituting X for U, the discrete objective function finally has the form

$$\tilde{C}(U) = \sum_j \left(\mathbf{b}_j - \sum_i a_{j,i} \mathbf{u}_i \right)^2 = \|\mathbf{b} - A\mathbf{u}\|^2, \tag{4.1}$$

where $\mathbf{u}$ is the vector of all unknowns $\mathbf{u}_i = (u_i, v_i)^T$. The coefficients $a_{j,i}$ result from the discretization of the second derivatives and the $\mathbf{b}_j$ are derived from the fitting constraints. Using normal equations, the linear least squares problem of Equation 4.1 can be transformed to a sparse system of linear equations, which can be solved with conjugate gradient methods.

Nested Iteration

A standard approach for improving the convergence of iterative conjugate gradient solvers is nested iteration. The system is first solved on a coarse representation and this solution is propagated to the finer representation. Since each unknown in Equation 4.1 corresponds to a sample of the model point cloud, a coarser representation can be obtained using the clustering methods described in Section 2.1. This scheme can be recursively extended to a hierarchy of nested approximations as illustrated in Figure 4.6. Special care has to be taken to define the constraints on the coarser levels, as the original point pair correspondences were defined on the finest level. A simple solution is to just propagate the constraint to the centroid of the cluster it belongs to. If a point carries more than one constraint at a certain level, all but one of the constraints are removed on this level to maintain the injectivity of the mapping.

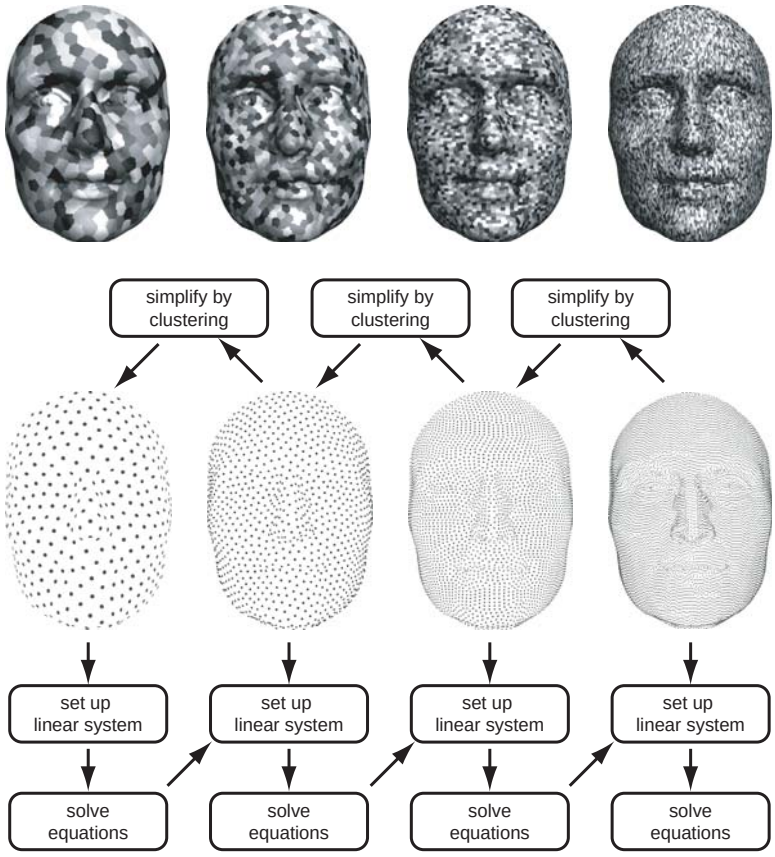


Figure 4.6: Hierarchy used for nested iteration. The top-row shows the clusters color-coded on the original point cloud.

Discussion

Figure 4.7 shows the influence of the parameter β in the objective function $C(X)$. As the images illustrate, it allows the user to define a trade-off between the approximation of the fitting constraints and the smoothness of the mapping. Floater has presented a different approach to compute a parameterization for point-sampled surfaces using shape preserving weights [FR01]. In his method the system of equations is obtained by expressing each unknown as a convex combination of its neighbors. This means that the boundary of the surface has to be mapped onto a convex region in parameter space. In the above formulation no such constraints have been imposed, which allows the method to be used as an extrapolator. Note that at least three point-pair correspondences are required to define the mapping. Figure 4.8 shows two examples of a texture mapping operation using the minimum distortion

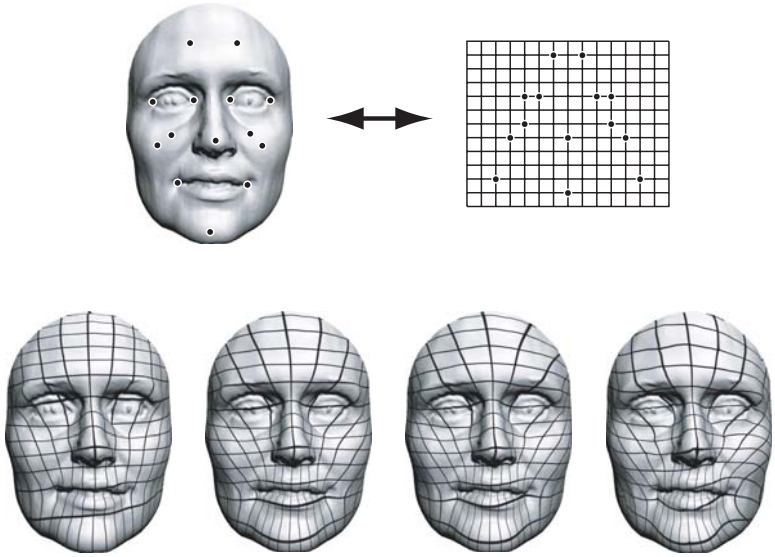


Figure 4.7: Influence of the weighting parameter β of the objective function. The top images illustrate the feature point correspondences, the bottom row show the resulting mapping for different values of β.

parameterization. As described in Section 4.1.2, this type of operation requires the user to interactively define corresponding feature points both on the 2D image and on the 3D point-sampled surface, as shown on the left. Figure 4.9 illustrates the distortion of these parameterizations by mapping a regular square grid texture onto the surface.

4.3 Re-Sampling

The mapping function Φ defines a parameter value $\mathbf{u}_i$ for each $\mathbf{p}_i \in P'$ of the selected surface patch S'. To create a one-to-one correspondence between the surface samples $\mathbf{p}_i \in P'$ and the brush samples $\mathbf{b}_{mn} \in B$, either the surface or the brush needs to be re-sampled. Note that the brush samples are implicitly parameterized, i.e., each $\mathbf{b}_{mn}$ has parameter coordinates $(m/M, n/N) \in [0, 1]$.

4.3.1 Re-sampling the Brush

Re-sampling the brush is performed by evaluating the continuous brush function represented by the discrete brush bitmaps at the parameter values of the sample points of the surface patch. The most

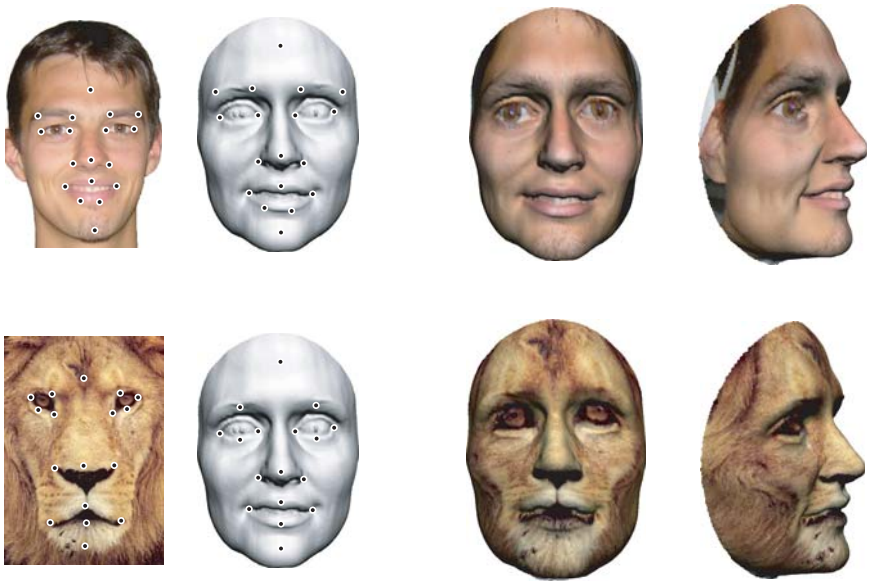


Figure 4.8: Texture mapping using the minimum distortion parameterization. The images on the left show the corresponding feature points. On the right, the final textured surface is shown.

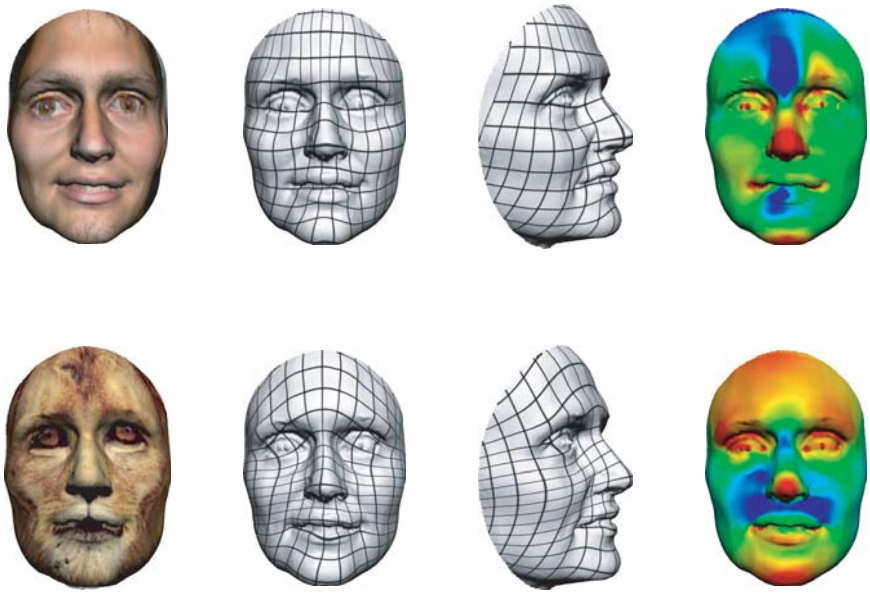


Figure 4.9: Illustration of local stretching for the texture mapping examples of Figure 4.8. The image on the right shows the color-coded first derivative of the parameterization, where blue denotes minimum absolute value and red denotes maximum absolute value.

simple reconstruction uses piecewise constant basis functions which amounts to nearest neighbor sampling. As Figure 4.10 (a) illustrates, this can lead to severe aliasing artifacts, which can be reduced by applying a Gaussian low-pass filter to the brush function prior to sampling. This filtering is analogous

to the elliptical weighted average (EWA) filtering used in point rendering and the reader is referred to [ZPvG01] for further details. Note that if the model surface is sampled substantially less densely than the brush, Gaussian filtering will lead to significant blurring. If the brush contains high-frequency detail, this information cannot be accurately mapped onto the surface without introducing new samples into the model.

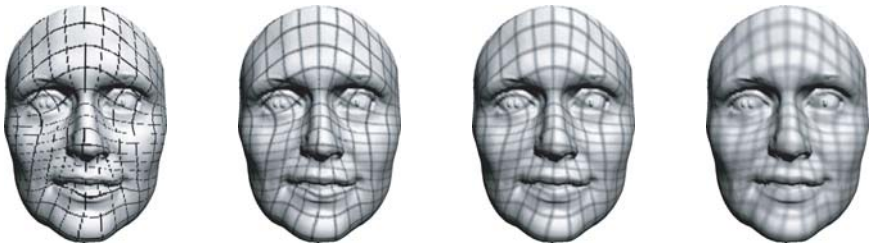


Figure 4.10: Re-sampling the brush on a model consisting of 40,880 points. (a) nearest-neighbor sampling, (b) - (d) Gaussian filtering with increasing filter width.

4.3.2 Re-sampling the Surface

To overcome the problem of information loss when re-sampling the brush, an alternative sampling method re-samples the surface to exactly map the $M \times N$ sampling grid of the brush. For this purpose a parameterized scattered data approximation is used. This method reconstructs a continuous surface from the discrete sample points, which can then be sampled at the parameter values of the brush samples. The idea is to compute local polynomial fitting functions at each $\mathbf{p}_i \in P'$ that are blended using a mapping from the local parameter plane of the fitting functions into the global parameter space of the surface (see [ZPKG02] for details). Figure 4.11 shows various examples of editing operations where the surface has been re-sampled according to the sampling grid of the brush. If the sampling resolution of the brush decreases, surface detail is lost. This is complementary to the situation described above, where brush information was lost due to insufficient sampling density of the model surface.

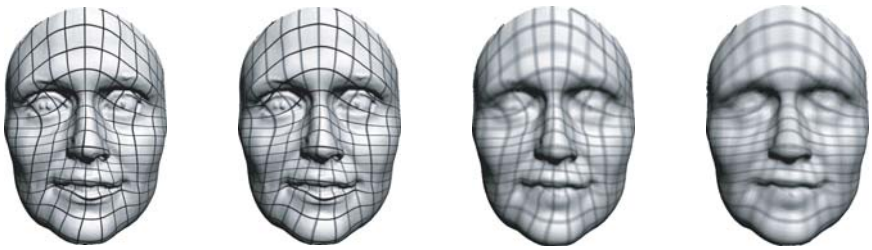


Figure 4.11: Re-sampling the surface. The sampling resolution of the brush varies from 500×500 pixels in (a), 250×250 pixels in (b), 100×100 pixels in (c), 50×50 to pixels in (d).

4.4 Surface Editing

The re-sampling method of Section 4.3 provides samples of the surface $S_\Psi = \Psi(\Phi(S))$ and of the brush $B_\Psi = \Psi(B)$ with identical sampling distribution. Thus the two can be combined by applying an editing operator directly on the discrete coefficients. Note that both S_Ψ and B_Ψ represents all the

sample attributes shown in Figure 4.4 (a). Depending on the intended functionality, an editing operator will then manipulate a subset of these surface attributes, such as diffuse color or spectral coefficient. In the following some of the editing operators will be described that have been implemented in the Pointshop3D system. A prime will denote the manipulated attributes, e.g., $\mathbf{x}_i'$ describes the position of a sample of the edited surface. Quantities that stem from the brush are marked with a bar, e.g., $\bar{\mathbf{c}}_i$ is the diffuse color of a brush sample. All other variables are part of the surface function.

4.4.1 Painting

Painting operations modify surface attributes by alpha-blending corresponding surface and brush coefficients. For example, the diffuse color can be altered by applying the painting operator on the color values, i.e., $\mathbf{c}_i' = \bar{\alpha}_i \cdot \mathbf{c}_i + (1 - \bar{\alpha}_i) \cdot \bar{\mathbf{c}}_i$, where $\bar{\alpha}_i$ is an alpha value specified in the brush function (see Figure 4.12 (a)). Similarly, painting can be applied to other attributes.

(a) (b)

Figure 4.12: Editing operations on the Chameleon (101,685 points): (a) texture painting, where the flowers shown on the left have been alpha-blended onto the surface, (b) texture filtering, where an oil-paint filter has been applied to the left half of the model.

4.4.2 Sculpting

The system supports two variations of sculpting operations that modify the geometry of the surface. The first option is to apply normal displacements to the sample positions, i.e., $\mathbf{x}_i' = \mathbf{x}_i + \bar{d}_i \cdot \mathbf{n}_i$, where $\bar{d}_i$ is a displacement coefficient given in the brush function. As illustrated in Figure 4.14 (a), this type of editing operation is particularly suitable for embossing or engraving. On the other hand, a carving operation is motivated by the way artists work when sculpting with clay or stone. It implements a "chisel stroke" that removes parts of the surface in the fashion of a boolean intersection (see also Section 3.1). The editing tool is defined using a least squares plane fitted at the touching point of the tool cursor and an intrusion depth. The new sample position is then given by

$$\mathbf{x}_i' = \begin{cases} \bar{\mathbf{b}}_i + \bar{d}_i \cdot \bar{\mathbf{n}} & \|\mathbf{x}_i - \bar{\mathbf{b}}_i < \bar{d}_i \\ \mathbf{x}_i & \text{otherwise,} \end{cases}$$

where $\bar{\mathbf{b}}_i$ is the base point on the reference plane and $\bar{\mathbf{n}}$ is the plane normal (see Figure 4.13). Carving operations can also be applied to rough surfaces (see Figure 4.14 (b)), where normal displacements fail due to the strong variations of the surface normals.

4.4.3 Filtering

Filtering is a special kind of editing operation that modifies the samples of the original model using a user-specified filter function f. First, the filter function is applied to S_Ψ yielding $S_\Psi^f = f(S_\Psi)$. Then filtered and original attributes are combined using the brush function for alpha blending. As an example, consider texture filtering, i.e., $\mathbf{c}_i' = \bar{\alpha} \cdot \mathbf{c}_i^f + (1 - \bar{\alpha}) \cdot \mathbf{c}_i$, where $\mathbf{c}_i^f$ is the filtered color value (illustrated in Figure 4.12 (b)). The filter function is usually implemented as a discrete convolution. Therefore, arbitrary discrete linear filters can be implemented by simply choosing the appropriate kernel grid. Filters can be applied to any attribute associated with a sample, e.g., color, normal, or distance from the reference plane for geometric offset filtering. Note that filtering with large kernels can be implemented efficiently in the spectral domain, similar to [PG01].

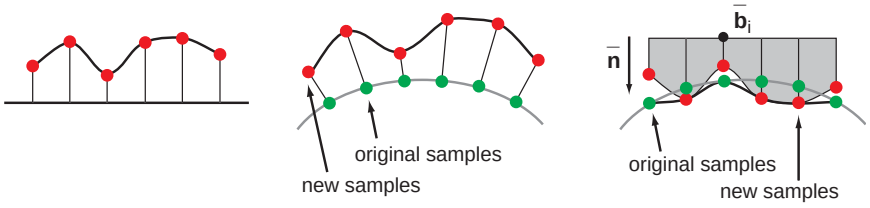


Figure 4.13: Normal displacements vs. carving. (a) brush image, (b) normal displacement, (c) carving.

Figure 4.14: Editing operations. Normal displacements (left) and carving on a rough surface (right).

Chapter 5

Pointshop3D

Mark Pauly

This chapter describes Pointshop3D, a software platform for point-based shape and appearance modeling [ZPKG02, PKKG03]. Most of the algorithms presented in this course have been implemented and analyzed within this framework. A brief description of the main components is followed by a more detailed discussion on data structures for spatial queries.

5.1 System Overview

Pointshop3D is designed as a modular software platform and implementation test-bed for point-based graphics applications. It provides a set of kernel functions for loading, storing, modifying, and rendering point-sampled surfaces. Most of the functionality is implemented in various tools and plug-ins that are briefly discussed below. A more detailed description of the software architecture and user interface can be found in the online documentation available at www.pointshop3d.com.

5.1.1 Tools and Plug-ins

The interaction metaphor of Pointshop3D is similar to common photo editing tools. Figure 5.1 shows a screen shot of the main user interface, where the most important interaction tools are annotated:

- The color picker tool allows to extract sample attributes from the model.
- The selection tool is used to select parts of the surface, e.g., to create a patch for selection interactions.
- The navigation tool controls the camera position and the relative orientation of different objects with respect to the world coordinate system.
- The brush tool implements brush interactions.
- The deformation tool supports free-form deformation as described in Section 3.2.
- The eraser tool allows to remove parts of the surface.
- The filter tool implements various filters (see Section 4.4.3).
- The lighting tool controls the position of the light source.
- The parameterization tool allows the specification of point constraints for computing a minimum distortion parameterization (see Section 4.2.2).

- The template tool is a place-holder that supports easy extensions of the system by adding new tools.
- The What's this? tool provides an online help.
- The color chooser tool allows to select an rgb color from a palette for brush painting.
- The brush builder is used to create new brushes (see Figure 4.4) for brush interactions.
- The brush chooser stores a list of pre- or user-defined brushes used for brush interactions.
- The tool settings viewer displays additional parameters of the currently active tool.
- The preview renderer icon activates the preview renderer.
- The object chooser selects a certain object within the scene.

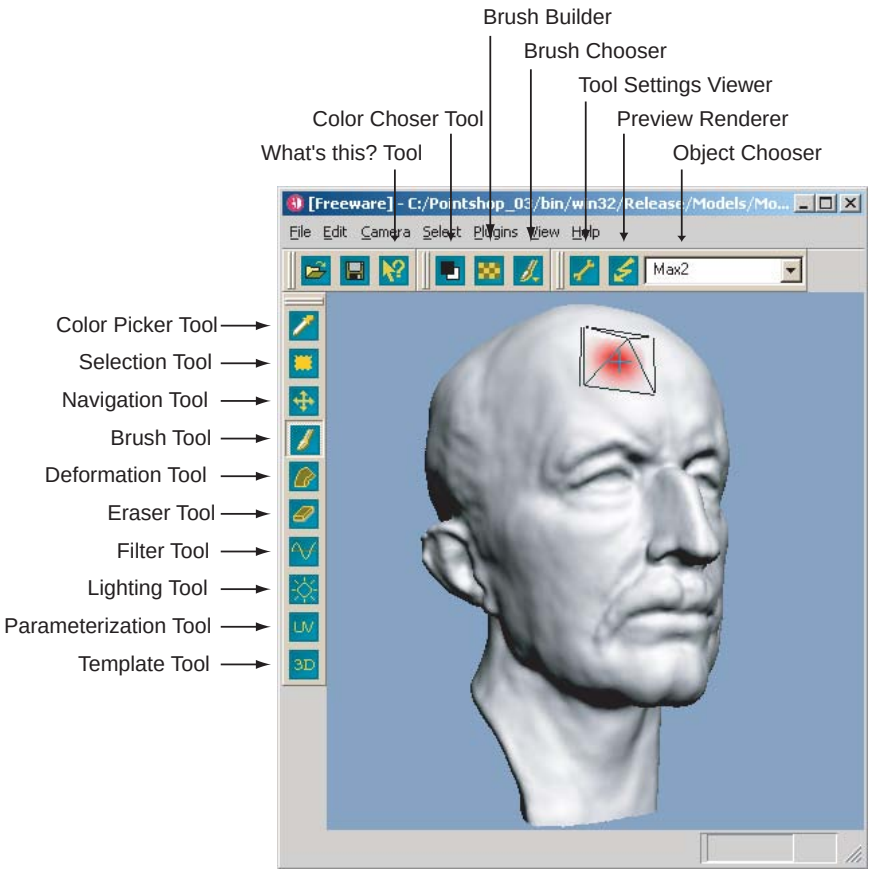


Figure 5.1: The main Pointshop3D interface.

Each of these tools can be configured as described in the online documentation of Pointshop3d. Additional functionality, e.g., a watermarking method, is implemented as plug-ins that are accessible via the main menu. Tools and plug-ins are implemented as dynamic link libraries and can be loaded dynamically at run-time, which makes the system easily extensible.

5.2 Data Structures

The central computational primitive required by the algorithms described in the previous chapters is closest points query: Given a point $\mathbf{x} \in \mathbb{R}^3$, find the point(s) $\mathbf{p}_i \in P$, such that $\|\mathbf{x} - \mathbf{p}_i\|$ is minimal. Closest point queries occur in two different contexts:

- The neighborhood relation defined in Section 1.1 is based on the k-closest points of a sample in P. In this case the query point is part of the set of point samples, i.e., $\mathbf{x} \in P$. The number of required closest points ranges from 8 to 20 for, e.g., normal estimation (Section 1.2), to up to about 200 for multi-scale variation estimation for feature classification [PKG03].
- Spatial queries are also required for the MLS projection (Section 1.3), boolean classification (Section 3.1), and collision detection (Section 3.2.1). Here the query point can be arbitrary, i.e., in general, and typically only a single closest point is required.

The performance of the algorithms implemented in Pointshop3D critically depends on efficient data structures for such queries. The requirements for these data structures vary considerably, depending on the specific context. For example, multi-scale feature classification [PKG03] applies nearest neighbor queries with varying neighborhood size for static objects, while particle simulation (Section 2.3) requires closest point queries for completely dynamic point distributions. Other operations, e.g., the filtering methods applied during free-form deformation (Section 3.2) operate on models with dynamic point locations, but mostly constant neighborhood relations.

In the design of appropriate data structures, one is typically faced with trade-off between efficient construction/updating of the data structure and fast query response. In Pointshop3D two main data structures, kd-trees and dynamic grids, have been implemented. Kd-trees feature fast query response, yet perform poorly when new points need to be inserted or existing points updated or removed. Dynamic grids provide efficient updating at the expense of slower query response times.

5.2.1 Kd-Trees

Bentley introduced kd-trees as a generalization of binary search trees in higher dimensions [Ben75]. A number of improvements have been proposed since then, see for example [FBF77, Spr91]. Kd-trees are binary trees, where each node is associated with a cubical cell and a plane orthogonal to one of the three coordinate axes. The plane splits the cell into two sub-cells, which correspond to two child nodes in the tree. The cells of the leaf nodes are called buckets and represent the whole space.

In Pointshop3D, kd-trees are built in a recursive top-down approach, where the sliding-midpoint rule is used to determine the split axis (see [AF00]). Average time complexity for the construction of the tree is $O(n \log n)$ for a point cloud consisting of n sample points. To compute the k-closest points from a given location $\mathbf{x} \in \mathbb{R}^3$, the tree is traversed from the root until the bucket containing $\mathbf{x}$ is found. From this leaf node, the k-closest points are determined by back-trapping towards the root. Finding the k-closest points takes $O(k + \log n)$ time on average, insertions and updates of single points require time $O(\log n)$ [Sam90].

5.2.2 Dynamic Grids

Dynamic grids have been implemented as a data structure for closest point queries in dynamic settings. The idea is to subdivide space into a regular grid of cubical cells, each of which stores a list of all points that fall within its cell [Hec97]. Range queries can be performed by scan-converting the query sphere to determine all cells that potentially contain samples located within the query range. Each of these cells is then processed by testing all points of the cell against the query sphere. To accommodate variations of point density over time, the grid can be re-structured dynamically as described in [Hec97].

Finding the correct grid cell for a point $\mathbf{x} \in \mathbb{R}^3$ that lies within the object's bounding box takes $O(1)$ time. Thus a dynamic grid for n samples can be constructed in $O(n)$ time and the insertion and

Data Structure	Construction	Insertion	Update	Query
List	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Grid	$O(n)$	$O(1)$	$O(1)$	$O(m)$
Kd-tree	$O(n \log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 5.1: Average time complexity for spatial data structures used for closest point queries. n is the number of points in P, m is the average number of points in each grid cell. Update refers to a change of sample position or a sample point deletion.

updating of a single point takes $O(1)$ time. Since each grid cell stores a linear list of points, the average time complexity of a query is $O(m)$, where m is the average number of points per cell. Note that a linear list of all sample points is a special case of a grid data structure consisting of only one grid cell.

Figure 5.2 shows a 2D illustration of the kd-tree and grid data structures. Table 5.1 compares the average time complexity of these data structures for typical operations used in the algorithms of Pointshop3D.

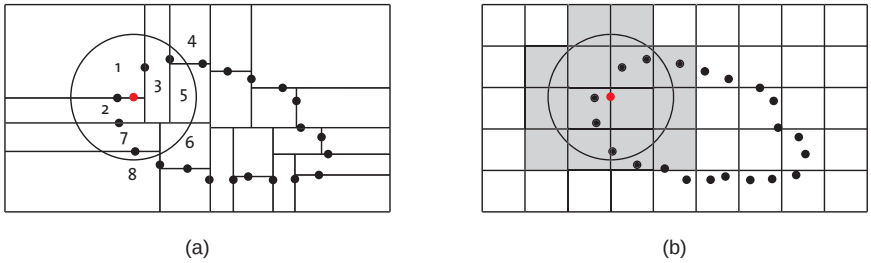


Figure 5.2: Spatial data structures for closest point queries, where the red dot indicates the query location and the black circle shows the query sphere. (a) kd-tree, where the numbers indicate the order in which the buckets are traversed, (b) dynamic grid, where the gray cells mark the intersection with the query sphere.

5.2.3 Neighborhood Caching

For many processing methods the k-closest points of a sample point have to be determined to establish local neighborhood relations (see Section 1.1). Often the point cloud is static or semi-dynamic, i.e., point locations vary but neighborhood relations remain (mostly) constant. If neighborhood relations have to be used multiple times, the efficiency of the computations can be improved considerably by caching these neighborhoods. Each sample point explicitly stores a list of its k-nearest neighbors computed using a kd-tree or a dynamic grid. This defines an adjacency graph that is comparable to a connectivity graph for triangle meshes. Unlike the latter, the adjacency graph does not define a consistent manifold surface, however. Thus the construction of the adjacency graph is significantly more efficient than the reconstruction of a triangle mesh from a given point cloud.

5.2.4 Spatial Queries in Pointshop3D

This section discusses the use of the data structures described above in the algorithms of Pointshop3D.

Surface Simplification

Spatial data structures for surface simplification have already been discussed in Section 2.4.4. Incremental clustering (Section 2.1.1) uses kd-trees for nearest neighbor queries, since the point cloud is static. Hierarchical clustering (Section 2.1.2) builds a BSP-tree and does not require closest point queries. Iterative simplification (Section 2.2) builds an explicit adjacency graph using kd-trees during initialization, which is maintained during simplification. Particle simulation (Section 2.3) makes use of dynamic grids, since individual particles experience significant drifts along the surface, which leads to dynamically changing point neighborhoods.

Boolean Classification

For boolean operations (Section 3.1), two static objects are positioned relative to each other by the user. Since the resulting affine transformation can be directly incorporated into the query, no updates of point locations are required. Thus closest point queries for classification are implemented using kd-trees.

Free-Form Deformation

For free-form deformation (Section 3.2) closest point queries are required to evaluate the distance functions d_j (see Equation 3.2) for computing the scale factor t. This is done once at the beginning of the modeling session, so no dynamic updates are required while the user interactively deforms the surface. Thus a kd-tree is most suitable for scale factor computation.

Collision Detection

For collision detection (Section 3.2.1) the classification predicate Ω (see Equation 3.1) has to be evaluated for dynamically varying point locations. Since performance is most critical, the algorithm only detects collisions of the deformable region with the zero-region. Since the latter is described by a static point cloud, kd-trees can be used for efficient closest point queries. Thus the collision detection algorithm in its current version cannot detect collisions of the deformable region with itself.

Particle-Based Blending

The blending method based on oriented particles (Section 3.1.4) uses the dynamic grid data structure. Since the particle simulation is mostly used to smooth out the sharp creases created by boolean operations (see Figure 3.9), particles experience a small spatial displacement during the simulation. Thus the neighborhood relations typically remain constant for a certain number of iterations. This can be exploited to improve the performance by caching the neighborhoods as discussed above.

Dynamic Re-sampling

When dynamically sampling the surface during deformation (Section 3.2.2), the relaxation and interpolation filters require nearest neighbor information for samples in the deformable region. Since these samples vary in position and new samples are inserted dynamically, kd-trees cannot be used efficiently. The dynamic grid data structure has also been found to be unsuitable, due to the relatively high cost of a spatial query. The relaxation filter shifts the point locations only slightly, which typically does not change the neighborhood relation significantly. Therefore, neighborhoods are computed at the beginning of the interaction and cached during deformation (see above). When new points are inserted, neighborhoods have to be updated, however. Since new samples are created by splitting an existing sample into two, the corresponding neighborhood relations can be propagated from the parent sample to its child samples. To maintain neighborhood consistency, each sample stores bidirectional neighborhood relations, i.e., a list of its neighbors and a list of samples of which itself is a neighbor.

Bibliography

[AA03a] Adamson A., Alexa M.: Approximating and intersecting surfaces from points. In *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing* (2003), Eurographics Association, pp. 230–239.

[AA03b] Adamson A., Alexa M.: Ray tracing point set surfaces. In *Proceedings of Shape Modeling International* (2003).

[ABCO*01] Alexa M., Behr J., Cohen-Or D., Fleishman S., Levin D., Silva C. T.: Point set surfaces. In *Proceedings of the conference on Visualization '01* (2001).

[ABK98] Amenta N., Bern M., Kamvysselis M.: A new Voronoi-based surface reconstruction algorithm. *Computer Graphics 32*, Annual Conference Series (1998), 415–421.

[AF00] Arya S., Fu H.-Y. A.: Expected-case complexity of approximate nearest neighbor searching. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms* (2000), Society for Industrial and Applied Mathematics, pp. 379–388.

[AS94] Agarwal P. K., Suri S.: Surface approximation and geometric partitions. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms* (1994), Society for Industrial and Applied Mathematics, pp. 24–33.

[Bar84] Barr A. H.: Global and local deformations of solid primitives. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (1984), ACM Press, pp. 21–30.

[Ben75] Bentley J. L.: Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[BW00] Brodsky D., Watson B.: Model simplification through refinement. In *Graphics Interface* (May 2000), pp. 221–228.

[CRS] Cignoni P., Rocchini C., Scopigno R.: Metro: Measruring error on simplififed surfaces. In *Computer Graphics Forum*, pp. 167–174.

[dC76] do Carmo M. P.: *Differential Geometry of Curves and Surfaces*. Prentice Hall, 1976.

[FBF77] Freidman J. H., Bentley J. L., Finkel R. A.: An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3*, 3 (1977), 209–226.

[FCOAS03] Fleishman S., Cohen-Or D., Alexa M., Silva C. T.: Progressive point set surfaces. *ACM Transactions on Graphics (TOG) 22*, 4 (2003), 997–1011.

[FR01] Floater M. S., Reimers M.: Meshless parameterization and surface reconstruction. *Comput. Aided Geom. Des. 18*, 2 (2001), 77–92.

[Gar99] Garland M.: *Quadric-Based Polygonal Surface Simplification*. PhD thesis, Computer Science Department, Carnegie Mellon University, 1999.

[GH97] Garland M., Heckbert P. S.: Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), ACM Press/Addison-Wesley Publishing Co., pp. 209–216.

[GJ02] Giesen J., John M.: Surface reconstruction based on a dynamical system. *Computer Graphics Forum 21* (2002).

[HDD*94] Hoppe H., DeRose T., Duchamp T., Halstead M., Jin H., McDonald J., Schweitzer J., Stuetzle W.: Piecewise smooth surface reconstruction. *Computer Graphics 28*, Annual Conference Series (1994), 295–302.

[Hec97] Heckbert P. S.: *Fast Surface Particle Repulsion*. Tech. rep., CMU Computer Science, 1997.

[Hop96] Hoppe H.: Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM Press, pp. 99–108.

[Jol86] Jolliffe I.: *Principal Component Analysis*. Springer Verlag, 1986.

[KCVS98] Kobbelt L., Campagna S., Vorsatz J., Seidel H.-P.: Interactive multi-resolution modeling on arbitrary meshes. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), ACM Press, pp. 105–114.

[KM97] Krishnan S., Manocha D.: An efficient surface intersection algorithm based on lower-dimensional formulation. *ACM Trans. Graph. 16*, 1 (1997), 74–106.

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

[Lev01] Levy B.: Constrained texture mapping for polygonal meshes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM Press, pp. 417–424.

[Lev03] Levin D.: Mesh-independent surface interpolation. *Geometric Modeling for Scientific Visualization* (2003).

[Lin01] Linsen L.: *Point Cloud Representation*. Tech. rep., Faculty of Computer Science, University of Karlsruhe, 2001.

[LPC*00] Levoy M., Pulli K., Curless B., Rusinkiewicz S., Koller D., Pereira L., Ginzton M., Anderson S., Davis J., Ginsberg J., Shade J., Fulk D.: The digital michelangelo project: 3d scanning of large statues. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), ACM Press/Addison-Wesley Publishing Co., pp. 131–144.

[LS86] Lancaster P., Salkauskas K.: *Curve and Surface Fitting: An Introduction*. Academic Press, 1986.

[PG01] Pauly M., Gross M.: Spectral processing of point-sampled geometry. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM Press, pp. 379–386.

[PGK02] Pauly M., Gross M., Kobbelt L. P.: Efficient simplification of point-sampled surfaces. In *Proceedings of the conference on Visualization '02* (2002), pp. 163–170.

[PKG02] Pauly M., Kobbelt L., Gross M.: *Multiresolution Modeling of Point-Sampled Geometry.* Tech. rep., ETH Zurich, Department of Computer Science, 2002.

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

[PKKG03] Pauly M., Keiser R., Kobbelt L. P., Gross M.: Shape modeling with point-sampled geometry. *ACM Transactions on Graphics (TOG) 22*, 3 (2003), 641–650.

[RB93] Rossignac J., Borrel P.: Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in Computer Graphics* (1993), pp. 455–465.

[Sam90] Samet H.: *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[Sed98] Sedgewick R.: *Algorithms in C++ (3rd edition).* Addison Wesley, 1998.

[SG01] Shaffer E., Garland M.: Efficient adaptive simplification of massive meshes. In *Proceedings of the conference on Visualization '01* (2001), IEEE Computer Society, pp. 127–134.

[SP86] Sederberg T. W., Parry S. R.: Free-form deformation of solid geometric models. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (1986), ACM Press, pp. 151–160.

[Spr91] Sproull R. L.: Refinements of nearest-neighbour searching in k-dimensional trees. *Algorithmica 6* (1991), 579–589.

[ST92] Szeliski R., Tonnesen D.: Surface modeling with oriented particle systems. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (1992), ACM Press, pp. 185–194.

[Tur92] Turk G.: Re-tiling polygonal surfaces. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (1992), ACM Press, pp. 55–64.

[Tur01] Turk G.: Texture synthesis on surfaces. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM Press, pp. 347–354.

[WH94] Witkin A. P., Heckbert P. S.: Using particles to sample and control implicit surfaces. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), ACM Press, pp. 269–277.

[ZPKG02] Zwicker M., Pauly M., Knoll O., Gross M.: Pointshop 3d: an interactive system for point-based surface editing. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), ACM Press, pp. 322–329.

[ZPvG01] Zwicker M., Pfister H., van Baar J., Gross M.: Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM Press, pp. 371–378.

[ZRB*04] Zwicker M., Rsnen J., Botsch M., Dachsbacher C., Pauly M.: Perspective accurate splatting. In *Proceedings of Graphics Interface* (2004).