New Directions in Shape Representations

Course Notes for SIGGRAPH 2001
Los Angeles, California

Course 33
Monday, August 13, 2001

Organizers

Hanspeter Pfister	Mitsubishi Electric Research Laboratories
Alyn Rockwood	Colorado School of Mines

Speakers

Sarah Frisken	Mitsubishi Electric Research Laboratories
Markus Gross	ETH Zürich, Switzerland
Leonard McMillan	Massachusetts Institute of Technology
Henry Moreton	NVIDIA
Ron Perry	Mitsubishi Electric Research Laboratories
Hanspeter Pfister	Mitsubishi Electric Research Laboratories
Wim Sweldens	Bell Laboratories, Lucent Technologies
Alyn Rockwood	Colorado School of Mines

Course Summary

Several recently developed shape representations go beyond conventional surface and volume techniques and offer advantages for compression, transmission, high resolution, editing, and rendering of complex shapes.

In this course, some of the world's leading computer graphics researchers and practitioners summarize the state of the art in shape representations and provide detailed information on how to implement the various methods. The course includes a discussion of various applications, including sculpting and 3D scanning of real-life objects.

Topics

Introduction and overview.
Displaced subdivision surfaces.
Normal meshes.
Point-based graphics and visualization.
Surface representations and signal processing.
Adaptively sampled distance fields.
Image-based representations.

Speaker Biographies

Sarah Frisken
Senior Research Scientist
MERL - Mitsubishi Electric Research Laboratories
201 Broadway, Cambridge, MA 02139
Email: frisken@merl.com
Web: http://www.merl.com/people/frisken/

Sarah Frisken is a Senior Research Scientist at MERL-Cambridge Research where she has worked in graphical modeling, visualization, volume graphics and surgical simulation. She recently completed a collaborative project to build a knee arthroscopy simulator which incorporated high quality rendering, haptic feedback and physical modeling of interactions between surgical tools and a 3D computer model derived from MRI data. Her current interests include the development of Adaptively Sampled Distance Fields, a new surface representation that can efficiently represent smooth surfaces, sharp edges, and fine detail, and which can be used as an electronic clay in design and CAD/CAM.

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

Markus Gross is a professor at the Computer Science Department of the Swiss Federal Institute of Technology (ETH) in Zürich. 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 he was with the Computer Graphics Center in Darmstadt, where he established and directed the Visual Computing Group. His research interests include Scientific Visualization, Multidimensional Data Analysis, Physics-based Modeling and Multiresolution Methods. He has widely published and lectured on Visual Computing, Computer Graphics and Scientific Visualization and he authored the book "Visual Computing", Springer, 1994. He has been a member of the editorial advisory board of the IEEE Transactions on Visualization and Computer Graphics, the IEEE Computer Graphics and Applications, the Computer Graphics Forum, the Computers & Graphics. In addition, he has served as a member of international program committees of major Graphics Conferences and as a papers co-chair of the IEEE Visualization 99 and Eurographics 2000 Conferences.

Leonard McMillan
Assistant Professor EECS
Laboratory for Computer Science
Computer Graphics Group
545 Technology Square, Cambridge, MA 02139
Email: mcmillan@lcs.mit.edu
Web: http://www.graphics.lcs.mit.edu/~mcmillan

Leonard McMillan is a pioneer in the area of image-based rendering. Image-based rendering (IBR) is a new approach to computer graphics in which scenes are modeled using a collection of reference images. These reference images can then be used to synthesize new renderings from a wide range of viewing positions. He has worked a wide range of different approaches to IBR including warping images with depth, light field rendering, and generating view-dependent models directly from live video streams. Leonard is also interested in a wide range of related topics including three-dimension display technologies, computer graphics hardware, and the fusion of image processing, multimedia, and computer graphics. Leonard is an Assistant Professor in the EECS Department and a member of the Computer Graphics Group of the Laboratory for Computer Science at MIT. Leonard received his BSEE and MSEE from Georgia Institute of Technology and his Ph.D. from the University of North Carolina at Chapel Hill. He has also worked at Bell Laboratories and Sun Microsystems.

Henry Moreton
3D Architect
NVIDIA
3535 Monroe Street
Santa Clara, CA 95051
Email: moreton@nvidia.com
Web: www.nvidia.com

Henry Moreton joined NVIDIA in the fall of 1998 as a member of the architecture group. From 1984 to 1998 he worked at Silicon Graphics. In 1992 he received a Ph.D. from the University of California, Berkeley. He has published in the areas of curve and surface modeling, rendering, texture mapping, video and image compression, and unmanned submarine control. He has patents issued and pending in the areas of optics, video compression, graphics, system and CPU architecture, and curve & surface modeling & rendering. Current interests include alternative representations for geometry, API design and hardware architecture of programmable graphics devices.

Ron Perry
Research Scientist
MERL - Mitsubishi Electric Research Laboratories
201 Broadway
Cambridge, MA 02139
Email: perry@merl.com
Web: http://www.merl.com/people/perry/

Ron Perry is a Research Scientist at Mitsubishi Electric Research Laboratories (MERL) where his interests include fundamental algorithms in computer graphics. Prior to joining MERL, Ron was a consulting engineer at Compaq and involved in the development of the three-dimensional rendering ASIC Neon. Ron has consulted for many companies including Kodak, Apple, and Quark over the last 20 years, architecting and developing software and hardware products in the area of computer graphics, imaging, color, and desktop publishing.

Hanspeter Pfister
Research Scientist
MERL - Mitsubishi Electric Research Laboratories
201 Broadway
Cambridge, MA 02139
Email: pfister@merl.com
Web: http://www.merl.com/people/pfister/

Hanspeter Pfister is a 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. Hanspeter Pfister received his Ph.D. in Computer Science in 1996 from the State University of New York at Stony Brook. In his doctoral research he developed Cube-4, a scalable architecture for real-time volume rendering. He received his M.S. in Electrical Engineering from the Swiss Federal Institute of Technology (ETH) Zürich, Switzerland, in 1991. He is a member of the ACM, ACM SIGGRAPH, IEEE, the IEEE Computer Society, and the Eurographics Association.

Alyn Paul Rockwood
Colorado School of Mines
18375 Highland Estates Drive
Colorado Springs, CO 80908
Office: (719) 495-7073
Fax: (719) 495-1248
Email: aprockwood@earthlink.net

Alyn Rockwood received a Ph.D. from the Dept. of Applied Math and Theoretical Physics at Cambridge University. He has had faculty positions at a German "Gymnasium," teaching math and physics, at BYU teaching math, and nine years at Arizona State University in computer science. He has additionally spent over 15 years in industrial research at Evans and Sutherland, Shape Data Ltd., SGI, a start-up company and Mitsubishi Electric Research Labs. Currently he is a professor at Colorado School of Mines, Department of Math and Computer Science. Altogether he has spent over 25 years as a researcher in mathematics, computer graphics, CAD/CAM and simulation. He has produced over 50 peer-reviewed articles, 9 patents and 3 books in these areas. He was recently the SIGGRAPH99 papers' chair and will be the SIGGRAPH conference chair for 2003.

Wim Sweldens
Director, Scientific Computing Research
Bell Laboratories
Lucent Technologies
600 Mountain Avenue, Rm 2C-276
Murray Hill, NJ 07974
Email: wim@lucent.com
Web: http://wim.sweldens.com

Wim Sweldens received his PhD in 1994 from the Katholieke Universiteit Leuven, Belgium. He is currently the director of Scientific Computing Research at Bell Laboratories, Lucent Technologies. His research is concerned with wavelets and multiscale analysis and its application in numerical analysis, signal processing, computer graphics, and wireless communications. He is the inventor of the lifting scheme, a new design and implementation technique for wavelets on which the JPEG2000 standard is based. Lifting also allows the generalization of multiresolution signal processing techniques to complex geometries. More recently he worked on Digital Geometry Processing: editing, compression, manipulation, and meshing of surfaces. He has lectured internationally on the use of multiscale techniques in computer graphics and received several best paper awards; he co-organized the Wavelet course in 96 and participated in three other SIGGRAPH courses. MIT's Technology Review recently selected him as one of 100 top young technological innovators. He is the founder and Editor-in-Chief of the Wavelet Digest.

Schedule and Contents

8:30 Welcome and Introduction
Hanspeter Pfister, MERL
Slides: Introduction to New Directions in Shape Representations

8:45 Motivation and Overview
Alyn Rockwood, Colorado School of Mines
Slides: Motivation and Overview

9:15 Displaced Subdivision Surfaces
Henry Moreton, NVIDIA
Slides: Displaced Subdivision Surfaces
Paper: Displaced Subdivision Surfaces

10:00 Break

10:30 Normal Meshes
Wim Sweldens, Bell Labs
Slides: Normal Meshes
Paper: Normal Meshes

11:15 Point-based Graphics and Visualization
Hanspeter Pfister, MERL
Slides: Point-based Graphics and Visualization
Paper: Surface Splatting
Paper: Surfels: Surface Elements as Rendering Primitives

12:00 Lunch

1:30 Surface Representations and Signal Processing
Markus Gross, ETH Zürich
Slides: Surface Representations and Signal Processing
Paper: Spectral Processing of Point-Sampled Geometry

2:15 ADFs – Adaptively Sampled Distance Fields
Sarah Frisken, Ron Perry, MERL
Slides: ADFs – Adaptively Sampled Distance Fields

3:00 Break

3:30 Image-Based Representations
Leonard McMillan, MIT

Slides: Image-Based Modeling and Rendering
Paper: Image-Based Visual Hulls
Paper: Creating and Rendering Image-Based Visual Hulls

4:15 Q&A

4:45 Adjourn

New Directions in Shape Representations

Course 33

Hanspeter Pfister, Alyn Rockwood
Sarah Frisken, Markus Gross,
Leonard McMillan, Henry Moreton,
Ron Perry, Wim Sweldens

SIGGRAPH 2001

Welcome!

In this course we will cover:

- Displaced subdivision surfaces.
- Normal meshes.
- Point-based graphics and visualization.
- Surface representations and signal processing.
- Adaptively sampled distance fields.
- Image-based representations.

SIGGRAPH 2001

Goals

Introduce new shape representations.

- General overview
- Specifics
- Applications

Demonstrate their advantages.

Get you to think beyond triangles and NURBS.

Schedule

8:30	Welcome and Introduction	(Pfister)
8:45	Motivation and Overview	(Rockwood)
9:15	Displaced Subdivision Surfaces	(Moreton)
10:00	Break	
10:30	Normal Meshes	(Sweldens)
11:15	Point-based Graphics and Visualization	(Pfister)
12:00	Lunch	
1:30	Surface Representations and Signal Processing	(Gross)
2:15	ADFs - Adaptively Sampled Distance Fields	(Frisken, Perry)
3:00	Break	
3:30	Image-Based Representations	(McMillan)
4:15	Q&A	
4:45	Adjourn	

New Directions in Shape Representations
Motivation and Overview

Alyn Rockwood
Colorado School of Mines

- Outside the box
- What's wrong with QWERTY?
- Rummaging through the attic

SIGGRAPH 2001

Outside the box

Topics

- Displaced Subdivision Surfaces
- Normal Meshes
- Semi-regular meshes
- Surfels
- Wavelets
- Adaptively Sampled Distance Fields
- Image-based representations

"It is the customary fate of new truths to begin as heresies and end up as superstitions."
- T.H. Huxley

SIGGRAPH 2001

What's wrong with QWERTY?

What are conventional representations?
e.g.

- Lines
- Polygons
- Bezier
- B-spline

Real Time Rendering of Trimmed Surfaces - SIGGRAPH89

What's right with QWERTY?

- Well understood
- standards
- software libraries
- hardware support
- great for boxes

What's wrong with QWERTY?

Polygons

- In low res - angularities, Mach bands, webbing, waffle, ball peen hammer effect.
- In high res - large overkill datasets, topology (vertices edges and faces)
- Cracking, backfacing

SIGGRAPH 2001

My Attic - point based rendering

1986 - cast shadow

1986 - 512^2

SIGGRAPH 2001

My Attic - point based rendering

- Easy to program
- variety of objects
- smooth quality

SIGGRAPH 2001

•My Attic - Subdivision Surfaces

- Sabin-Doo - REMUS
- GM Symposium 1983

SIGGRAPH 2001

•My Attic - Wavelets

Meyers - 1992

Muraki - 1992 IEEE VIS

SIGGRAPH 2001

My Desk - Distance Fields

Frisken et al - SIGGRAPH 2000

SIGGRAPH rejection - 2001

"One needs a number of failures before success - so we should be very active making them"

-Hestenes

SIGGRAPH 2001

Displaced Subdivision Surfaces

Henry Moreton
NVIDIA Corporation

1

Our Approach

DSS = Smooth Surface ⊗ Scalar Disp Field

Control mesh | Domain Surface | Displaced Subdivision Surface

2

Use a B&W Image to Define Height

3

Scaling Displacements

4

Deforming the Domain Surface

5

Others

6

Representation Overview

7

Displacement Maps Correspond to Subdivision Sampling

Maps are 2^n+1 in size…

Triangular Subdivision

Quad Subdivision

8

Advantages of DSS

- Level of detail
- Animation
- Intrinsic parameterization
- Unified representation

9

Level of Detail

- Vary tessellation
- Bump map to preserve visual detail
- Z-texture to preserve depth (!z-fighting)

10

More Detail!

11

Animation

Light-weight

- Manipulate a smooth base surface
- Animate a highly detailed mesh

Simple base surface

- Model so detail is scalar offset
- Model with sufficient animation control

12

Intrinsic Parameterization

- Governed by a subdivision surface
- No storage necessary
- Capture details as scalar displacement

13

Unified Representation

- Same subdivision rule for geometry and displacement
- Well-defined magnification

14

What is Different About DSS?

vs. displacement maps

- Forward mapping
- Magnification filter

DSS are *forward-mapped* displacement mapped subdivision surfaces with a well-defined *magnification filter*.

15

Forward Mapping is Efficient

- Midpoint subdivision mesh refinement
- Displacement sampling
  occurs in ***object-space***
- Filtering is redundant frame-to-frame
- Also true for displacement mapped triangles …
- In Contrast:
  texture sampling occurs in ***screen-space***

16

1D Magnification example…

Displacement

Curve and Offset

17

1D Cubic Filtering

DSS Curve
linear filter

DSS Curve
subdivision filter
≡ cubic B-spline

18

2D Polynomial Filtering

- Well-defined magnification filter on an arbitrary mesh
  - The surface is subdivided ***n*** times with map 2^n+1
  - Displacement is appended to x,y,z as w component
- A Loop subdivision surface is a generalized ***quartic triangular B-spline patch***
- A Catmull-Clark subdivision surface is a generalized ***cubic B-spline patch***

19

Analytic Behavior

- C^1 continuous everywhere except at extraordinary vertices

$$\vec{S} = \vec{P} + D\hat{n}$$

$$\vec{S}_u = \vec{P}_u + D_u\hat{n} + D\hat{n}_u$$

$$\hat{n}_u = \frac{\vec{n}_u - \hat{n}\left(\vec{n}_u \cdot \hat{n}\right)}{\left\|\vec{n}\right\|} \quad \text{and} \quad \vec{n}_u = \vec{P}_{uu} \times \vec{P}_v + \vec{P}_u \times \vec{P}_{uv}$$

20

Normal Evaluation

21

Normal Maps

- Can be used to calculate bump maps
- Relative to domain (base) surface

134,656 faces 8,416 faces 526 faces

22

Normal Map Calculation

$$\left\{\hat{n}_{base} \quad \hat{t}_{base} \quad \hat{b}_{base}\right\} \cdot \hat{n}_{map} = \hat{n}_{dss}$$

$$\hat{n}_{map} = \left\{\hat{n}_{base} \quad \hat{t}_{base} \quad \hat{b}_{base}\right\}^{-1} \cdot \hat{n}_{dss}$$

23

Bump, Stretch & Squeeze

Want a *static* bump map, but…

- Stretching should flatten bumps
- Squeezing increases normal variation

24

Modified Tangent Frame

Use a non-orthonormal tangent frame

$$\Leftarrow \left\{ \frac{\vec{S}_u}{\left\|\vec{S}_u\right\|^2} \quad \frac{\vec{S}_v}{\left\|\vec{S}_v\right\|^2} \quad \hat{n} \right\}$$

25

Improvements on Bump Mapping

Bump mapping simulates lighting of a rough surface…

- It does nothing for occlusion…
  - Silhouettes
    - We can't relocate pixels – adaptive tessellation
  - Z-fighting
    - We can adjust Z per pixel – Z texture

26

Z- Textures

27

Z-texture for Displacement Correction

$$\vec{p}'_e = \vec{p}_e + \Delta \hat{n}_e$$

$$z'_c = z_c + \Delta\left(\hat{n} \cdot \mathbf{T}_{\text{proj}}[3]\right)$$

$$w'_c = w_c + \Delta\left(\hat{n} \cdot \mathbf{T}_{\text{proj}}[4]\right)$$

$$z'_c = z_c + \Delta_{\text{B}}\left(\hat{n} \cdot \mathbf{T}_{\text{proj}}[3]\right) - \Delta_{\text{D}}\left(\hat{n} \cdot \mathbf{T}_{\text{proj}}[3]\right)$$

$$w'_c = w_c + \Delta_{\text{B}}\left(\hat{n} \cdot \mathbf{T}_{\text{proj}}[4]\right) - \Delta_{\text{D}}\left(\hat{n} \cdot \mathbf{T}_{\text{proj}}[3]\right)$$

28

Z-texture Characteristics

- Addresses Z-fighting problems
- Incorrect occlusion

Interaction with shadow buffers…

- Identifier-based shadow buffers are incompatible with z-textures
- Depth-based shadow buffers exploit planar triangles

29

A Simple Z-texture Example

Bump mapped
& Z-textured

Bump mapped

30

Adaptive Tessellation

- Test edges
- Test interior
- Interpolate for
  "Flat" vertices
- Draw degenerate triangles

Works for pixel lighting

31

Displaced Triangles Don't Animate Well

32

Variable *k* is Non-Trivial

Examples are uniform *k*

To guarantee smoothness the boundaries of finer regions must be forced to match coarse sampling

33

DSS Construction

- Published work is top-down fine→coarse
- Range-scan processing better bottom-up.
- Modeling also bottom-up.

34

Displaced Subdivision Surfaces

Aaron Lee
Department of Computer Science
Princeton University
http://www.aaron-lee.com/

Henry Moreton
NVIDIA Corporation
moreton@nvidia.com

Hugues Hoppe
Microsoft Research
http://research.microsoft.com/~hoppe

ABSTRACT

In this paper we introduce a new surface representation, the *displaced subdivision surface*. It represents a detailed surface model as a scalar-valued displacement over a smooth domain surface. Our representation defines both the domain surface and the displacement function using a unified subdivision framework, allowing for simple and efficient evaluation of analytic surface properties. We present a simple, automatic scheme for converting detailed geometric models into such a representation. The challenge in this conversion process is to find a simple subdivision surface that still faithfully expresses the detailed model as its offset. We demonstrate that displaced subdivision surfaces offer a number of benefits, including geometry compression, editing, animation, scalability, and adaptive rendering. In particular, the encoding of fine detail as a *scalar* function makes the representation extremely compact.

Additional Keywords: geometry compression, multiresolution geometry, displacement maps, bump maps, multiresolution editing, animation.

1. INTRODUCTION

Highly detailed surface models are becoming commonplace, in part due to 3D scanning technologies. Typically these models are represented as dense triangle meshes. However, the irregularity and huge size of such meshes present challenges in manipulation, animation, rendering, transmission, and storage. Meshes are an expensive representation because they store:

(1) the irregular connectivity of faces,

(2) the (x,y,z) coordinates of the vertices,

(3) possibly several sets of texture parameterization (u,v) coordinates at the vertices, and

(4) texture images referenced by these parameterizations, such as color images and bump maps.

An alternative is to express the detailed surface as a displacement from some simpler, smooth domain surface (see Figure 1). Compared to the above, this offers a number of advantages:

(1) the patch structure of the domain surface is defined by a control mesh whose connectivity is much simpler than that of the original detailed mesh;

(2) fine detail in the displacement field can be captured as a scalar-valued function which is more compact than traditional vector-valued geometry;

(3) the parameterization of the displaced surface is inherited from the smooth domain surface and therefore does not need to be stored explicitly;

(4) the displacement field may be used to easily generate bump maps, obviating their storage.

(a) control mesh (b) smooth domain surface (c) displaced subdivision surface

Figure 1: Example of a displaced subdivision surface.

A simple example of a displaced surface is terrain data expressed as a height field over a plane. The case of functions over the sphere has been considered by Schröder and Sweldens [33]. Another example is the 3D scan of a human head expressed as a radial function over a cylinder. However, even for this simple case of a head, artifacts are usually detectable at the ear lobes, where the surface is not a single-valued function over the cylindrical domain.

The challenge in generalizing this concept to arbitrary surfaces is that of finding a smooth underlying domain surface that can express the original surface as a scalar-valued offset function.

Krishnamurthy and Levoy [25] show that a detailed model can be represented as a displacement map over a network of B-spline patches. However, they resort to a vector-valued displacement map because the detailed model is not always an offset of their B-spline surface. Also, avoiding surface artifacts during animation requires that the domain surface be tangent-plane (C^1) continuous, which involves constraints on the B-spline control points.

We instead define the domain surface using subdivision surfaces, since these can represent smooth surfaces of arbitrary topological type without requiring control point constraints. Our representation, the *displaced subdivision surface*, consists of a control mesh and a scalar field that displaces the associated subdivision surface locally along its normal (see Figure 1). In this paper we use the Loop [27] subdivision surface scheme, although the representation is equally well defined using other schemes such as Catmull-Clark [5].

Both subdivision surfaces and displacement maps have been in use for about 20 years. One of our contributions is to unify these two ideas by defining the displacement function using the same subdivision machinery as the surface. The scalar displacements are stored on a piecewise regular mesh. We show that simple subdivision masks can then be used to compute analytic properties on the resulting displaced surface. Also, we make displaced subdivision surface practical by introducing a scheme for constructing them from arbitrary meshes.

We demonstrate several benefits of expressing a model as a displaced subdivision surface:

Compression: both the surface topology and parameterization are defined by the coarse control mesh, and fine geometric detail is captured using a scalar-valued function (Section 5.1).

Editing: the fine detail can be easily modified since it is a scalar field (Section 5.2).

Animation: the control mesh makes a convenient armature for animating the displaced subdivision surface, since geometric detail is carried along with the deformed smooth domain surface (Section 5.3).

Scalability: the scalar displacement function may be converted into geometry or a bump map. With proper multiresolution filtering (Section 5.4), we can also perform magnification and minification easily.

Rendering: the representation facilitates adaptive tessellation and hierarchical backface culling (Section 5.5).

2. PREVIOUS WORK

Subdivision surfaces: Subdivision schemes defining smooth surfaces have been introduced by Catmull and Clark [5], Doo and Sabin [13], and Loop [27]. More recently, these schemes have been extended to allow surfaces with sharp features [21] and fractionally sharp features [11]. In this paper we use the Loop subdivision scheme because it is designed for triangle meshes.

DeRose et al. [11] define scalar fields over subdivision surfaces using subdivision masks. Our scalar displacement field is defined similarly, but from a denser set of coefficients on a piecewise regular mesh (Figure 2).

Hoppe et al. [21] describe a method for approximating an original mesh with a much simpler subdivision surface. Unlike our conversion scheme of Section 4, their method does not consider whether the approximation residual is expressible as a scalar displacement map.

Displacement maps: The idea of displacing a surface by a function was introduced by Cook [9]. Displacement maps have become popular commercially as procedural *displacement shaders* in RenderMan [1]. The simplest displacement shaders interpolate values within an image, perhaps using standard bicubic filters. Though displacements may be in an arbitrary direction, they are almost always along the surface normal [1].

Typically, normals on the displaced surface are computed numerically using a dense tessellation. While simple, this approach requires adjacency information that may be unavailable or impractical with low-level APIs and in memory-constrained environments (e.g. game consoles). Strictly local evaluation requires that normals be computed from a continuous analytic surface representation. However, it is difficult to piece together multiple displacement maps while maintaining smoothness. One encounters the same vertex enclosure problem [32] as in the stitching of B-spline surfaces. While there are well-documented solutions to this problem, they require constructions with many more coefficients (9× in the best case), and may involve solving a global system of equations.

In contrast, our subdivision-based displacements are inherently smooth and have only quartic total degree (fewer DOF than bicubic). Since the displacement map uses the same parameterization as the domain surface, the surface representation is more compact and displaced surface normals may be computed more efficiently. Finally, unifying the representation around subdivision simplifies implementation and makes operations such as magnification more natural.

Krishnamurthy and Levoy [25] describe a scheme for approximating an arbitrary mesh using a B-spline patch network together with a vector-valued displacement map. In their scheme, the patch network is constructed manually by drawing patch boundaries on the mesh. The recent work on surface pasting by Chan et al. [7] and Mann and Yeung [29] uses the similar idea of adding a vector-valued displacement map to a spline surface.

Gumhold and Hüttner [19] describe a hardware architecture for rendering scalar-valued displacement maps over planar triangles. To avoid cracks between adjacent triangles of a mesh, they interpolate the vertex normals across the triangle face, and use this interpolated normal to displace the surface. Their scheme permits adaptive tessellation in screen space. They discuss the importance of proper filtering when constructing mipmap levels in a displacement map. Unlike our representation, their domain surface is not smooth since it is a polyhedron. As shown in Section 5.3, animating a displaced surface using a polyhedral domain surface results in many surface artifacts.

Kobbelt et al. [23] use a similar framework to express the geometry of one mesh as a displacement from another mesh, for the purpose of multiresolution shape deformation.

Bump maps: Blinn [3] introduces the idea of perturbing the surface normal using a bump map. Peercy et al. [31] present recent work on efficient hardware implementation of bump maps. Cohen et al. [8] drastically simplify meshes by capturing detail in the related *normal maps*. Both Cabral et al. [4] and Apodaca and Gritz [1] discuss the close relationship of bump mapping and displacement mapping. They advocate combining them into a unified representation and resorting to true displacement mapping only when necessary.

Multiresolution subdivision: Lounsbery et al. [28] apply multiresolution analysis to arbitrary surfaces. Given a parameterization of the surface over a triangular domain, they compress this (vector-valued) parameterization using a wavelet basis, where the basis functions are defined using subdivision of the triangular domain. Zorin et al. [39] use a similar subdivision framework for multiresolution mesh editing. To make this multiresolution framework practical, several techniques have been developed for constructing a parameterization of an arbitrary surface over a triangular base domain. Eck et al. [14] use Voronoi/Delaunay diagrams and harmonic maps, while Lee et al. [26] track successive mappings during mesh simplification.

In contrast, displaced subdivision surfaces do not support an arbitrary parameterization of the surface, since the parameterization is given by that of a subdivision surface. The benefit is that we need only compress a *scalar*-valued function instead of *vector*-valued parameterization. In other words, we store only geometric detail, not a parameterization. The drawback is that the original surface must be expressible as an offset of a smooth domain surface. An extremely bad case would be a fractal "snowflake" surface, where the domain surface cannot be made much simpler than the original surface. Fortunately, fine detail in most practical surfaces is expressible as an offset surface.

Guskov et al. [20] represent a surface by successively applying a hierarchy of displacements to a mesh as it is subdivided. Their construction allows most of the vertices to be encoded using scalar displacements, but a small fraction of the vertices require vector displacements to prevent surface folding.

3. REPRESENTATION OVERVIEW

A displaced subdivision surface consists of a triangle control mesh and a piecewise regular mesh of scalar displacement coefficients (see Figure 2). The domain surface is generated from the control mesh using Loop subdivision. Likewise, the displacements applied to the domain surface are generated from the scalar displacement mesh using Loop subdivision.

Figure 2: Control mesh (left) with its piecewise regular mesh of scalar displacement coefficients ($k = 3$).

Displacement map: The scalar displacement mesh is stored for each control mesh triangle as one half of the sample grid $(2^k+1)\times(2^k+1)$, where k depends on the sampling density required to achieve a desired level of accuracy or compression.

To define a *continuous* displacement function, these stored values are taken to be subdivision coefficients for the same (Loop) subdivision scheme that defines the domain surface. Thus, as the surface is magnified (i.e. subdivided beyond level k), both the domain surface geometry and the displacement field are subdivided using the same machinery. As a consequence, the displacement field is C^1 even at extraordinary vertices, and the displaced subdivision surface is C^1 everywhere *except* at extraordinary vertices. The handling of extraordinary vertices is discussed below.

For surface minification, we first compute the limit displacements for the subdivision coefficients at level k, and we then construct a mipmap pyramid with levels $\{0,\ldots,k-1\}$ by successive filtering of these limit values. We cover filtering possibilities in Section 4.5. As with ordinary texture maps, the content author may sometimes want more precise control of the filtered levels, so it may be useful to store the entire pyramid. (For our compression analysis in Section 5.1, we assume that the pyramid is built automatically.)

For many input meshes, it is inefficient to use the same value of k for all control mesh faces. For a given face, the choice of k may be guided by the number of original triangles associated it, which is easily estimated using MAPS [26]. Those regions with lower values of k are further subdivided *logically* to produce a mesh with uniform k.

Normal Calculation: We now derive the surface normal for a point $\vec{S}$ on the displaced subdivision surface. Let $\vec{S}$ be the displacement of the limit point $\vec{P}$ on the domain surface:

$$\vec{S} = \vec{P} + D\hat{n} ,$$

where D is the limit displacement and $\hat{n} = \vec{n} / \|\vec{n}\|$ is the unit normal on the domain surface. The normal $\vec{n}$ is obtained as $\vec{n} = \vec{P}_u \times \vec{P}_v$ where the tangent vectors $\vec{P}_u$ and $\vec{P}_v$ are computed using the first derivative masks in Figure 3.

The displaced subdivision surface normal at S is defined as $\vec{n}_s = \vec{S}_u \times \vec{S}_v$ where each tangent vector has the form

$$\vec{S}_u = \vec{P}_u + D_u\hat{n} + D\hat{n}_u .$$

If the displacements are relatively small, it is common to ignore the third term, which contains second-order derivatives [3].

However, if the surface is used as a modeling primitive, then the displacements may be quite large and the full expression must be evaluated. The difficult term $\hat{n}_u = \vec{n}_u / \|\vec{n}_u\|$ may be derived using the Weingarten equations [12]. Equivalently, it may be expressed as:

$$\hat{n}_u = \frac{\vec{n}_u - \hat{n}(\vec{n}_u \cdot \hat{n})}{\|\vec{n}\|} \quad \text{where} \quad \vec{n}_u = \vec{P}_{uu} \times \vec{P}_v + \vec{P}_u \times \vec{P}_{uv} .$$

At a regular (valence 6) vertex, the necessary partial derivatives are given by a simple set of masks (see Figure 3). At extraordinary vertices, the curvature of the domain surface vanishes and we omit the second-order term. In this case, the standard Loop tangent masks may be used to compute the first partial derivatives. Since there are few extraordinary vertices, this simplified normal calculation has not proven to be a problem.

Figure 3: Loop masks for limit position P and first and second derivatives at a regular control vertex.

Bump map: The displacement map may also be used to generate a bump map during the rendering of coarser tessellations (see Figure 13). This improves rendering performance on graphics systems where geometry processing is a bottleneck. The construction of this bump map is presented in Section 5.4.

Other textures: The domain surface parameterization is used for storing the displacement map (which also serves to define a bump map). It is natural to re-use this same inherent parameterization to store additional appearance attributes for the surface, such as color. Section 4.4 describes how such attributes are re-sampled from the original surface.

Alternatively, one could define more traditional surface parameterizations by explicitly specifying (u,v) texture coordinates at the vertices of the control mesh, as in [11]. However, since the domain of a (u,v) parameterization is a planar region, this generally requires segmenting the surface into a set of charts.

4. CONVERSION PROCESS

To convert an arbitrary triangle mesh (Figure 5a) into a displaced subdivision surface (Figure 5b), our process performs the following steps:

- Obtain an initial control mesh (Figure 5c) by simplifying the original mesh. Simplification is done using a traditional sequence of edge collapse transformations, but with added heuristics to attempt to preserve a scalar offset function.
- Globally optimize the control mesh vertices (Figure 5d) such that the domain surface (Figure 5e) more accurately fits the original mesh.
- Sample the displacement map by shooting rays along the domain surface normals until they intersect the original mesh. At the ray intersection points, compute the signed displacement, and optionally sample other appearance attributes like surface color. (The black line segments visible in Figure 5f correspond to rays with positive displacements.)

4.1 Simplification to control mesh

We simplify the original mesh using a sequence of edge collapse transformations [22] prioritized according to the quadric error metric of Garland and Heckbert [16]. In order to produce a good domain surface, we restrict some of the candidate edge collapses.

The main objective is that the resulting domain surface should be able to express the original mesh using a scalar displacement map. Our approach is to ensure that the space of normals on the domain surface remains locally similar to the corresponding space of normals on the original mesh.

To maintain an efficient correspondence between the original mesh and the simplified mesh, we use the MAPS scheme [26] to track parameterizations of all original vertices on the mesh simplified so far. (When an edge is collapsed, the parametrizations of points in the neighborhood are updated using a local 1-to-1 map onto the resulting neighborhood.)

For each candidate edge collapse transformation, we examine the mesh neighborhood that would result. In Figure 4, the thickened 1-ring is the neighborhood of the unified vertex. For vertices on this ring, we compute the subdivision surface normals (using tangent masks that involve vertices in the 2-ring of the unified vertex). The highlighted points within the faces in the 1-ring represent original mesh vertices that are currently parameterized on the neighborhood using MAPS.

Figure 4: Neighborhood after candidate edge collapse and, for one face, the spherical triangle about its domain surface normals.

(a) original mesh (b) displaced subdivision surface

(c) initial control mesh (d) optimized control mesh

(e) smooth domain surface (f) displacement field

Figure 5: Steps in the conversion process.

For each face in the 1-ring neighborhood, we gather the 3 subdivision surface normals at the vertices and form their spherical triangle on the Gauss sphere. Then, we test whether this spherical triangle encloses the normals of the original mesh vertices parameterized using MAPS. If this test fails on any face in the 1-ring, the edge collapse transformation is disallowed. To allow simplification to proceed further, we have found it useful to

broaden each spherical triangle by pushing its three vertices an additional 45 degrees away from its inscribed center, as illustrated in Figure 4.

We observe that the domain surface sometimes has undesirable undulations when the control mesh has vertices of high valence. Therefore, during simplification we also disallow an edge collapse if the resulting unified vertex would have valence greater than 8.

4.2 Optimization of domain surface

Having formed the initial control mesh, we optimize the locations of its vertices such that the associated subdivision surface more accurately fits the original mesh. This step is performed using the method of Hoppe et al. [21]. We sample a dense set of points from the original mesh and minimize their squared distances to the subdivision surface. This nonlinear optimization problem is approximated by iteratively projecting the points onto the surface and solving for the most accurate surface while fixing those parameterizations. The result of this step is shown in Figure 5d-e.

Note that this geometric optimization modifies the control mesh and thus affects the space of normals over the domain surface. Although this invalidates the heuristic used to guide the simplification process, this has not been a problem in our experiments. A more robust solution would be to optimize the subdivision surface for each candidate edge collapse (as in [21]) prior to testing the neighborhood normals, but this would be much more costly.

4.3 Sampling of scalar displacement map

We apply k steps of Loop subdivision to the control mesh. At each of these subdivided vertices, we compute the limit position and normal of the domain surface. We seek to compute the signed distance from the limit point to the original surface along the normal (Figure 5f).

The directed line formed by the point and normal is intersected with the original surface, using a spatial hierarchy [17] for efficiency. We disregard any intersection point if the intersected surface is oriented in the wrong direction with respect to the directed line. If multiple intersection points remain, we pick the one closest to the domain surface. Figure 6 illustrates a possible failure case if the domain surface is too far from the original.

Figure 6: The displacement sampling may "fold over itself" if the domain surface is too distant from the original mesh.

Near surface boundaries, there is the problem that the domain surface may extend beyond the boundary of the original surface, in which case the ray does not intersect any useful part of the original surface. (We detect this using a maximum distance threshold based on the mesh size.) In this case, the surface should really be left undefined, i.e. trimmed to the detailed boundary of the original mesh. One approach would be to store a special illegal value into the displacement map. Instead, we find the closest original triangle to the subdivided vertex, and intersect the ray with the plane containing that triangle. Precise surface trimming can be achieved using an alpha mask in the surface color image, but we have not yet implemented this.

4.4 Resampling of appearance attributes

Besides sampling the scalar displacement function, we also sample other appearance attributes such as diffuse color. These attributes are stored, filtered, and compressed just like the scalar displacements. An example is shown in Figure 11.

4.5 Filtering of displacement map

Since our displacement field has the same structure as the domain surface, we can apply the same subdivision mask for magnification. This is particular useful when we try to zoom in a tiny region on our displaced subdivision surface. For sampling the displacements at minified levels of the displacement pyramid, we compute the samples at any level $l<k$ by filtering the limit displacements of level $l+1$. We considered several filtering operations and opted for the non-shrinking filter of Taubin [35].

Because the displacement magnitudes are kept small, their filtering is not extremely sensitive. In many rendering situations much of the visual detail is provided by bump mapping. As has been discussed elsewhere [2], careful filtering of bump maps is both important and difficult.

4.6 Conversion results

The following table shows execution times for the various steps of the conversion process. These times are obtained on a Pentium III 550 MHz PC.

Model	armadillo	venus	bunny	dinosaur
Conversion Statistics				
Original mesh #F	210,944	100,000	69,451	342,138
Control mesh #F	1,306	748	526	1,564
Maximum level k	4	4	4	4
Execution Times (minutes)				
Simplification	61	28	19	115
Domain surface optimiz.	25	11	11	43
Displacement sampling	2	2	1	5
Total	88	41	31	163

5. BENEFITS

5.1 Compression

Mesh compression has recently been an active area of research. Several clever schemes have been developed to concisely encode the combinatorial structure of the mesh connectivity, in as few as 1-2 bits per face (e.g. [18] [35]). As a result, the major portion of a compressed mesh goes to storing the mesh geometry. Vertex positions are typically compressed using quantization, local prediction, and variable-length delta encoding. Geometry can also be compressed within a multiresolution subdivision framework as a set of wavelet coefficients [28]. To our knowledge, all previous compression schemes for arbitrary surfaces treat geometry as a vector-valued function.

In contrast, displaced subdivision surfaces allow fine geometric detail to be compressed as a scalar-valued function. Moreover, the domain surface is constructed to be close to the original surface, so the magnitude of the displacements tends to be small.

To exploit spatial coherence in the scalar displacement map, we use linear prediction at each level of the displacement pyramid, and encode the difference between the predicted and actual values. For each level, we treat the difference coefficients over all

faces as a subband. For each subband, we use the embedded quantizer and embedded entropy coder described in Taubman and Zakhor [37]. The subbands are merged using the bit allocation algorithm described by Shoham and Gersho [34], which is based on integer programming.

An alternative would be to use the compression scheme of Kolarov and Lynch [24], which is a generalization of the wavelet compression method in [33].

Figure 10 and Table 1 show results of our compression experiments. We compare storage costs for simplified triangle meshes and displaced subdivision surfaces, such that both compressed representations have the same approximation accuracy with respect to the original reference model. This accuracy is measured as L^2 geometric distance between the surfaces, computed using dense point sampling [16]. The simplified meshes are obtained using the scheme of Garland and Heckbert [16]. For mesh compression, we use the *VRML compressed binary format* inspired by the work of Taubin and Rossignac [36]. We vary the quantization level for the vertex coordinates to obtain different compressed meshes, and then adjust our displacement map compression parameters to obtain a displaced surface with matching L^2 geometric error.

For simplicity, we always compress the control meshes losslessly in the experiments (i.e. with 23-bits/coordinate quantization). Our compression results would likely be improved further by adapting the quantization of the control mesh as well. However, this would modify the domain surface geometry, and would therefore require re-computing the displacement field. Also, severe quantization of the control mesh would result in larger displacement magnitudes.

Table 1 shows that displaced subdivision surfaces consistently achieve better compression rates than mesh compression, even when the mesh is carefully simplified from detailed geometry.

5.2 Editing

The fine detail in the scalar displacement mesh can be edited conveniently, as shown in the example of Figure 7.

Figure 7: In this simple editing example, the embossing effect is produced by enhancing the scalar displacements according to a texture image of the character 'B' projected onto the displaced surface.

5.3 Animation

Displaced subdivision surfaces are a convenient representation for animation. Kinematic and dynamics computation are vastly more efficient when operating on the control mesh rather than the huge detailed mesh.

Because the domain surface is smooth, the surface detail deforms naturally without artifacts. Figure 8 shows that in contrast, the use of a polyhedron as a domain surface results in creases and folds even with a small deformation of a simple surface.

Figure 8: Comparison showing the importance of using a smooth domain surface when deforming the control mesh. The domain surface is a subdivision surface on the left, and a polyhedron on the right.

Figure 12 shows two frames from the animation of a more complicated surface. For that example, we used 3D Studio MAX to construct a skeleton of bones inside the control mesh, and manipulated the skeleton to deform this mesh. (The complete animation is on the accompanying video.)

Another application of our representation is the fitting of 3D head scans [30]. For this application, it is desirable to re-use a common control mesh structure so that deformations can be conveniently transferred from one face model to another.

5.4 Scalability

Depending on the level-of-detail requirements and hardware capabilities, the scalar displacement function can either be:

- rendered as **explicit geometry**: Since it is a continuous representation, the tessellation is not limited to the resolution of the displacement mesh. A scheme for adaptive tessellation is presented in Section 5.5.
- converted to a **bump map**: This improves rendering performance on graphics systems where geometry processing is a bottleneck. As described in [31], the calculation necessary for tangent-space bump mapping involves computing the displaced subdivision surface normal relative to a coordinate frame on the domain surface. A convenient coordinate frame is formed by the domain surface unit normal $\hat{n}$ and a tangent vector such as $\vec{P}_u$. Given these vectors, the coordinate frame is:

$$\left\{\hat{b},\hat{t},\hat{n}\right\} \text{ where } \begin{array}{l}\hat{t} = \vec{P}_u / \left\|\vec{P}_u\right\| \\ \hat{b} = \hat{n} \times \hat{t}\end{array} .$$

Finally, the normal $\hat{n}_s$ to the displaced subdivision surface relative to this tangent space is computed using the transform:

$$\hat{n}_{\text{tangent space}} = \left\{\hat{b},\hat{t},\hat{n}\right\}^{\mathrm{T}} \cdot \hat{n}_s .$$

The computations of $\hat{n}$, $\vec{P}_u$, and $\hat{n}_s$ are described in Section 3. Note that we use the precise analytic normal in the bump map calculation. As an example, Figure 13 shows renderings of the same model with different boundaries between explicit geometry and bump mapping. In the leftmost image, the displacements are all converted into geometry, and bump-mapping is turned off. In the rightmost image, the domain surface is sampled only at the control mesh vertices, but the entire displacement pyramid is converted into a bump map.

5.5 Rendering

Adaptive tessellation: In order to perform adaptive tessellation, we need to compute the approximation error of any intermediate tessellation level from the finely subdivided surface. This approximation error is obtained by computing the maximum distance between the dyadic points on the planar intermediate level and their corresponding surface points at the finest level (see Figure 9). Note that this error measurement corresponds to parametric error and is stricter than geometric error. Bounding parametric error is useful for preventing appearance fields (e.g. bump map, color map) from sliding over the rendered surface [8]. These precomputed error measurements are stored in a quadtree data structure. At runtime, adaptive tessellation prunes off the entire subtree beneath a node if its error measurement satisfies given level-of-detail parameters. By default, the displacements applied to the vertices of a face are taken from the corresponding level of the displacement pyramid.

Note that the pruning will make adjacent subtrees meet at different levels. To avoid cracks, if a vertex is shared among different levels, we choose the finest one from the pyramid. Also, we perform a retriangulation of the coarser face so that it conforms to the vertices along the common edges. Figure 14 shows some examples of adaptive tessellation.

Figure 9: Error computation for adaptive tessellation.

Backface patch culling: To improve rendering performance, we avoid rendering regions of the displaced subdivision surface that are entirely facing away from the viewpoint. We achieve this using the *normal masks* technique of Zhang and Hoff [38].

On the finely subdivided version of the domain surface, we compute the vertex normals of the displaced surface as described in Section 3. We convert these into a normal mask for each subdivided face. During a bottom-up traversal of the subdivision hierarchy, we propagate these masks to the parents using the logical *or* operation.

Given the view parameters, we then construct a viewing mask as in [38], and take its logical *and* with the stored masks in the hierarchy. Generally, we cull away 1/3 to 1/4 of the total number of triangles, thereby speeding up rendering time by 20% to 30%.

6. DISCUSSION

Remeshing creases: As in other remeshing methods [14] [26], the presence of creases in the original surface presents challenges to our conversion process. Lee et al. [26] demonstrate that the key is to associate such creases with edges in the control mesh. Our simplification process also achieves this since mesh simplification naturally preserves sharp features.

However, displaced subdivision surfaces have the further constraint that the displacements are strictly scalar. Therefore, the edges of the control mesh, when subdivided and displaced, do not generally follow original surface creases exactly. (A similar problem also arises at surface boundaries.) This problem can be resolved if displacements were instead vector-based, but then the representation would lose its simplicity and many of its benefits (compactness, ease of scalability, etc.).

Scaling of displacements: Currently, scalar displacements are simply multiplied by unit normals on the domain surface. With a "rubbery" surface, the displaced subdivision surface behaves as one would expect, since detail tends to smooth as the surface stretches. However, greater control over the magnitude of displacement is desirable in many situations. A simple extension of the current representation is to provide scale and bias factors (s,b) at control mesh vertices. These added controls enhance the basic displacement formula:

$$\vec{S} = \vec{P} + (sD + b)\hat{n}$$

Exploring such scaling controls is an interesting area of future work.

7. SUMMARY AND FUTURE WORK

Nearly all geometric representations capture geometric detail as a vector-valued function. We have shown that an arbitrary surface can be approximated by a displaced subdivision surface, in which geometric detail is encoded as a scalar-valued function over a domain surface. Our representation defines both the domain surface and the displacement function using a unified subdivision framework. This synergy allows simple and efficient evaluation of analytic surface properties.

We demonstrated that the representation offers significant savings in storage compared to traditional mesh compression schemes. It is also convenient for animation, editing, and runtime level-of-detail control.

Areas for future work include: a more rigorous scheme for constructing the domain surface, improved filtering of bump maps, hardware rendering, error measures for view-dependent adaptive tessellation, and use of detail textures for displacements.

ACKNOWLEDGEMENTS

Our thanks to Gene Sexton for his help in scanning the dinosaur.

REFERENCES

[1] Apodaca, A. and Gritz, L. Advanced RenderMan – Creating CGI for Motion Pictures, Morgan Kaufmann, San Francisco, CA, 1999.

[2] Becker, B. and Max, N. Smooth transitions between bump rendering algorithms. Proceedings of SIGGRAPH 93, Computer Graphics, Annual Conference Series, pp. 183-190.

[3] Blinn, J. F. Simulation of wrinkled surfaces. Proceedings of SIGGRAPH 78, Computer Graphics, pp. 286-292.

[4] Cabral, B., Max, N. and Springmeyer, R. Bidirectional reflection functions from surface bump maps. Proceedings of SIGGRAPH 87, Computer Graphics, Annual Conference Series, pp.273-281.

[5] Catmull, E., and Clark, J. Recursively generated B-spline surfaces on arbitrary topological meshes. Computer Aided Design 10, pp. 350-355 (1978).

[6] Certain, A., Popovic, J., DeRose, T., Duchamp, T., Salesin, D. and Stuetzle, W. Interactive multiresolution surface viewing. Proceedings of SIGGRAPH 96, Computer Graphics, Annual Conference Series, pp. 91-98.

[7] Chan, K., Mann, S., and Bartels, R. World space surface pasting. Graphics Interface '97, pp. 146-154.

[8] Cohen, J., Olano, M. and Manocha, D. Appearance preserving Simplification. Proceedings of SIGGRAPH 98, Computer Graphics, Annual Conference Series, pp. 115-122.

[9] Cook, R. Shade trees. Computer Graphics (Proceedings of SIGGRAPH 84), 18(3), pp. 223-231.

[10] Deering, M. Geometry compression. Proceedings of SIGGRAPH 95, Computer Graphics, Annual Conference Series, pp. 13-20.

[11] DeRose, T., Kass, M., and Truong, T. Subdivision surfaces in character animation. Proceedings of SIGGRAPH 98, Computer Graphics, Annual Conference Series, pp. 85-94.

[12] Do Carmo, M. P. Differential Geometry of Curves and Surfaces. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

[13] Doo, D., and Sabin, M. Behavior of recursive division surfaces near extraordinary points. Computer Aided Design 10, pp. 356-360 (1978).

[14] Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M., and Stuetzle, W. Multiresolution analysis of arbitrary meshes. Proceedings of SIGGRAPH 95, Computer Graphics, Annual Conference Series, pp. 173-182.

[15] Forsey, D., and Bartels, R. Surface fitting with hierarchical splines. ACM Transactions on Graphics, 14(2), pp. 134-161 (April 1995).

[16] Garland, M., and Heckbert, P. Surface simplification using quadric error metrics. Proceedings of SIGGRAPH 97, Computer Graphics, Annual Conference Series, pp. 209-216.

[17] Gottschalk, S., Lin, M., and Manocha, D. OBB-tree: a hierarchical structure for rapid interference detection. Proceedings of SIGGRAPH 96, Computer Graphics, Annual Conference Series, pp. 171-180.

[18] Gumhold, S., and Straßer, W. Real time compression of triangle mesh connectivity. Proceedings of SIGGRAPH 98, Computer Graphics, Annual Conference Series, pp. 133-140.

[19] Gumhold, S., and Hüttner, T. Multiresolution rendering with displacement mapping. SIGGRAPH workshop on Graphics hardware, Aug 8-9, 1999.

[20] Guskov, I., Vidimce, K., Sweldens, W., and Schröder, P. Normal meshes. Proceedings of SIGGRAPH 2000, Computer Graphics, Annual Conference Series.

[21] Hoppe, H., DeRose, T., Duchamp, T., Halstead, M., Jin, H., McDonald, J., Schweitzer, J., and Stuetzle, W. Piecewise smooth surface reconstruction. Proceedings of SIGGRAPH 94, Computer Graphics, Annual Conference Series, pp. 295-302.

[22] Hoppe, H. Progressive meshes. Proceedings of SIGGRAPH 96, Computer Graphics, Annual Conference Series, pp. 99-108.

[23] Kobbelt, L., Bareuther, T., and Seidel, H. P. Multi-resolution shape deformations for meshes with dynamic vertex connectivity. Proceedings of EUROGRAPHICS 2000, to appear.

[24] Kolarov, K. and Lynch, W. Compression of functions defined on surfaces of 3D objects. In J. Storer and M. Cohn, editors, Proc. of Data Compression Conference, IEEE, pp. 281-291, 1997.

[25] Krishnamurthy, V., and Levoy, M. Fitting smooth surfaces to dense polygon meshes. Proceedings of SIGGRAPH 96, Computer Graphics, Annual Conference Series, pp. 313-324.

[26] Lee, A., Sweldens, W., Schröder, P., Cowsar, L., and Dobkin, D. MAPS: Multiresolution adaptive parameterization of surfaces. Proceedings of SIGGRAPH 98, Computer Graphics, Annual Conference Series, pp. 95-104.

[27] Loop, C. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, Department of Mathematics, 1987.

[28] Lounsbery, M., DeRose, T., and Warren, J. Multiresolution analysis for surfaces of arbitrary topological type. ACM Transactions on Graphics, 16(1), pp. 34-73 (January 1997).

[29] Mann, S. and Yeung, T. Cylindrical surface pasting. Technical Report, Computer Science Dept., University of Waterloo (June 1999).

[30] Marschner, S., Guenter, B., and Raghupathy, S. Modeling and rendering for realistic facial animation. Submitted for publication.

[31] Peercy, M., Airey, J. and Cabral, B. Efficient bump mapping hardware. Proceedings of SIGGRAPH 97, Computer Graphics, Annual Conference Series, pp. 303-306.

[32] Peters, J. Local smooth surface interpolation: a classification. Computer Aided Geometric Design, 7(1990), pp. 191-195.

[33] Schröder, P., and Sweldens, W. Spherical wavelets: efficiently representing functions on the sphere. Proceedings of SIGGRAPH 95, Computer Graphics, Annual Conference Series, pp. 161-172.

[34] Shoham, Y. and Gersho, A. Efficient bit allocation for an arbitrary set of quantizers. IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 36, No. 9, pp. 1445-1453, Sept 1988.

[35] Taubin, G. A signal processing approach to fair surface design. Proceedings of SIGGRAPH 95, Computer Graphics, Annual Conference Series, pp. 351-358.

[36] Taubin, G. and Rossignac, J. Geometric compression through topological surgery. ACM Transactions on Graphics, 17(2), pp. 84-115 (April 1998).

[37] Taubman, D. and Zakhor, A. Multirate 3-D subband coding of video. IEEE Transactions on Image Processing, Vol. 3, No. 5, Sept, 1994.

[38] Zhang, H., and Hoff, K. Fast backface culling using normal masks. Symposium on Interactive 3D Graphics, pp. 103-106, 1997.

[39] Zorin, D., Schröder, P., and Sweldens, W. Interactive multiresolution mesh editing. Proceedings of SIGGRAPH 97, Computer Graphics, Annual Conference Series, pp. 259-268.

Original mesh
342,138 faces; 1011 KB

Simplified mesh
50,000 faces; 169 KB

Compressed simplified mesh
(12-bits/coord.); 68 KB

Displaced subdivision surface
1564 control mesh faces; 18 KB

Original mesh
100,000 faces; 346 KB

Simplified mesh
20,000 faces; 75 KB

Compressed simplified mesh
(12-bits/coord.); 33 KB

Displaced subdivision surface
748 control mesh faces; 16 KB

Figure 10: Compression results. Each example shows the approximation of a dense original mesh using a simplified mesh and a displaced subdivision surface, such that both have comparable L^2 approximation error (expressed as a percentage of object bounding box).

Dinosaur	Original mesh		Compressed simplified mesh		Displaced subdivision surface (k=4)		
	#V=171,074 #F=342,138		#V=25,005 #F=50,000		$\#V^0$=787 $\#F^0$=1564 ≡ 6.5KB		
Quantization (bits/coord.)	L^2 error	Size (KB)	L^2 error	Size (KB)	L^2 error	Size (KB)	Size ratio
23	0.002%	1011	0.024%	169	0.025%	22	7.7
12	0.014%	322	0.028%	68	0.028%	18	3.8
10	0.053%	217	0.059%	50	0.058%	10	5.0
8	0.197%	169	0.21%	35	0.153%	7	5.0

Venus	Original mesh		Compressed simplified mesh		Displaced subdivision surface (k=4)		
	#V=50,002 #F=100,000		#V=10,002 #F=20,000		$\#V^0$=376 $\#F^0$=748 ≡ 3.4KB		
Quantization (bits/coord.)	L^2 error	Size (KB)	L^2 error	Size (KB)	L^2 error	Size (KB)	Size ratio
23	0.001%	346	0.027%	75	0.027%	17	4.4
12	0.014%	140	0.030%	33	0.031%	16	2.0
10	0.054%	102	0.059%	26	0.053%	8	3.2
8	0.207%	69	0.210%	18	0.149%	4	4.5

Table 1: Quantitative compression results for the two examples in Figure 10. Numbers in red refer to figures above.

Original colored mesh | Displaced subdivision surface | Domain surface | Displacement samples *(k=4)*

Figure 11: Example of a displaced subdivision surface with resampled color.

Original mesh | Control mesh | Displaced subdiv. surface | Modified control mesh | Resulting deformed surface

Figure 12: The control mesh makes a convenient armature for animating the displaced subdivision surface.

Level 4 (134,656 faces) | Level 3 (33,664 faces) | Level 2 (8,416 faces) | Level 1 (2,104 faces) | Level 0 (526 faces)

Figure 13: Replacement of scalar displacements by bump-mapping at different levels.

Threshold = 1.87% diameter
12,950 triangles; L^2 error = 0.104%

Threshold = 0.76% diameter
88,352 triangles; L^2 error = 0.035%

Threshold = 0.39% diameter
258,720 triangles; L^2 error = 0.016%

Figure 14: Example of adaptive tessellation, using the view-independent criterion of comparing residual error with a global threshold.

Normal Meshes

Wim Sweldens

Joint work with Igor Guskov,
Kiril Vidimce, Peter Schröder

Motivation

Surfaces
- highly detailed meshes

Scanned meshes
- manufacturing
- entertainment
- science

Efficient and compact representation

Surfaces

Height fields, terrains

- f(u,v)
- one float / vertex

3d surfaces

- x(u,v),y(u,v),z(u,v)
- three floats / vertex

Curves

Function

- f(u)
- one float / vertex

Curve

- x(u), y(u)
- two floats / vertex

Local Detail (x,y)

(t,n) stored

One float!

Simple Algorithm

Our Contribution

Normal meshes

- multiresolution
- geometry with one float per vertex

Conversion algorithm

- irregular to normal

What is in a mesh?

Connectivity + Vertices

sample locations geometry

Normal Meshes

Mesh		Normal mesh
■ geometry	····· ▶	same
■ connectivity	····· ▶	semi-regular
■ sample locations	····· ▶	optimal one float/vertex

Related work

Remeshing

- Hoppe et al. '94
- Eck et al. '95
- Krishnamurthy, Levoy '96
- Lee et al. '98

Siggraph 2000

- Lee et al. , Khodakovsky et al.

Piercing

Storage
- one float/vertex

Naïve algorithm:
- start with coarse mesh
- pierce recursively

But…

Naïve piercing does not work!

We need
- more control over the process

Algorithm

1. Initial patch layout
   - progressive hierarchy
   - global relaxation
2. Remeshing procedure
   - piercing step
   - one-to-one correspondence

Patch Layout

Base mesh
- simplification

Boundaries
- propagated

Global vertices
- relocated

Remeshing Procedure

Correspondence: patches ↔ faces

Piercing & Patches

Canonical:

Exception:

Maintenance

Operations

- new vertex
  - curve updated
- new edges
  - new curves

Drawing Curves

1. parameterize

2. draw in the plane

Result: curve on the original surface

Parameterization

Floater '97

- linear system

$$\Sigma_j \alpha_{ij} u_j = u_i$$

- conjugate gradient solver
- good initial guess

Results

(mean-square) error = 0.0056%

More Results

Error = 0.01%

Error = 0.004%

Adaptive Remeshing

Table

model	size	not normal	error	time
Feline	50,000	789	.015%	5 min
Torus3	6,000	421	.03%	3 min
Skull	25,000	817	.02%	3 min
Horse	60,000	644	.004%	7 min

Conclusions

Normal meshes
- remeshing procedure

Future work
- approximating subdivision
  - Catmull-Clark, Loop
- applications of normal meshes
  - compression, processing

Normal Meshes

Igor Guskov
Caltech

Kiril Vidimče
Mississippi State University

Wim Sweldens
Bell Laboratories

Peter Schröder
Caltech

Figure 1: *Left: original mesh (3 floats/vertex). Middle: two stages of our algorithm. Right: normal mesh (1 float/vertex). (Skull dataset courtesy Headus, Inc.)*

Abstract

Normal meshes are new fundamental surface descriptions inspired by differential geometry. A normal mesh is a multiresolution mesh where each level can be written as a normal offset from a coarser version. Hence the mesh can be stored with a single float per vertex. We present an algorithm to approximate any surface arbitrarily closely with a normal semi-regular mesh. Normal meshes can be useful in numerous applications such as compression, filtering, rendering, texturing, and modeling.

CR Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling - *curve, surface, solid, and object representations; hierarchy and geometric transformations*; G.1.2 [**Numerical Analysis**]: Approximation - *approximation of surfaces and contours, wavelets and fractals*

Additional Keywords: Meshes, subdivision, irregular connectivity, surface parameterization, multiresolution, wavelets.

1 Introduction

The standard way to parameterize a surface involves *three* scalar functions $x(u,v)$, $y(u,v)$, $z(u,v)$. Yet differential geometry teaches us that smooth surfaces locally can be described by a *single* scalar height function over the tangent plane. Loosely speaking one can say that the geometric information of a surface can be contained in only a single dimension, the height over this plane. This observation holds infinitesimally; only special cases such as terrains and star-shaped surfaces can globally be described with a single function.

In practice we often approximate surfaces using a triangle mesh. While describing meshes is relatively easy, they have lost much of the structure inherent in the original surface. For example, the above observation that locally a surface can be characterized by a scalar function is not reflected in the fact that we store 3 floats per vertex. In other words, the correlation between neighboring sample locations implied by the smoothness assumption is not reflected, leading to an inherently redundant representation.

While vertex locations come as 3-dimensional quantities, the above considerations tell us that locally two of those dimensions represent parametric information and only the third captures geometric, or shape, information. For a given smooth shape one may choose different parameterizations, yet the geometry remains the same. In the case of a mesh we can observe this by noticing that infinitesimal tangential motion of a vertex does not change the geometry, only the sampling pattern, or parameterization. Moving in the normal direction on the other hand changes the geometry and leaves parameter information undisturbed.

1.1 Goals and Contributions

Based on the above observations, the aim of the present paper is to compute mesh representations that only require a single scalar per vertex. We call such representations *normal meshes*. The main insight is that this can be done using multiresolution and local frames. A normal mesh has a hierarchical representation so that all detail coefficients when expressed in local frames are scalar, i.e., they only have a normal component. In the context of compression, for example, this implies that parameter information can be perfectly predicted and residual error is entirely constrained to the normal direction, i.e., contains only geometric information. Note that because of the local frames normal mesh representations are non-linear.

Of course we cannot expect a given arbitrary input mesh to possess a hierarchical representation which is normal. Instead we de-

scribe an algorithm which takes an arbitrary topology input mesh and produces a semi-regular normal mesh describing the same geometry. Aside from a small amount of base domain information, *our normal mesh transform converts an arbitrary mesh from a 3 parameter representation into a purely scalar representation*. We demonstrate our algorithm by applying it to a number of models and experimentally characterize some of the properties which make normal meshes so attractive for computations.

The study of normal meshes is of interest for a number of reasons: they

- bring our computational representations back towards the "first principles" of differential geometry;
- are very storage and bandwidth efficient, describing a surface as a succinctly specified base shape plus a hierarchical normal map;
- are an excellent representation for compression since all variance is "squeezed" into a single dimension.

1.2 Related Work

Efficient representations for irregular connectivity meshes have been pursued by a number of researchers. This research is motivated by our ability to acquire densely sampled, highly detailed scans of real world objects [19] and the need to manipulate these efficiently. Semi-regular—or subdivision connectivity—meshes offer many advantages over the irregular setting due of their well developed mathematical foundations and data structure simplicity [23]; many powerful algorithms require their input to be in semi-regular form [21, 22, 25, 1]. This has led to the development of a number of algorithms to convert existing irregular meshes to semi-regular form through remeshing. Eck et al. [9] use Voronoi tiling and harmonic maps to build a parameterization and remesh onto a semi-regular mesh. Krischnamurthy and Levoy [15] demonstrated user driven remeshing for the case of bi-cubic patches, while Lee et al. [18] proposed an algorithm based on feature driven mesh reduction to develop smooth parameterizations of meshes in an automatic fashion. These methods use the parameterization subsequently for semi-regular remeshing.

Our work is related to these approaches in that we also construct a semi-regular mesh from an arbitrary connectivity input mesh. However, in previous work prediction residuals, or detail vectors, were not optimized to have properties such as normality. The main focus was on the establishment of a smooth parameterization which was then semi-regularly sampled.

The discussion of parameter versus geometry information originates in the work done on irregular curve and surface subdivision [4] [13] and intrinsic curvature normal flow [5]. There it is shown that unless one has the correct parameter side information, it is not possible to build an irregular smooth subdivision scheme. While such schemes are useful for editing and texturing applications, they cannot be used for succinct representations because the parameter side-information needed is excessive. In the case of normal meshes these issues are entirely circumvented in that all parameter information vanishes and the mesh is reduced to purely geometric, i.e., scalar in the normal direction, information.

Finally, we mention the connection to displacement maps [3], and in particular normal displacement maps. These are popular for modeling purposes and used extensively in high end rendering systems such as RenderMan. In a sense we are solving here the associated inverse problem. Given some geometry, find a simpler geometry and a set of normal displacements which together are equivalent to the original geometry. Typically, normal displacement maps are single level, whereas we aim to build them in a fully hierarchical way. For example, single level displacements maps were used in [15] to capture the fine detail of a 3D photography model. Cohen et al. [2] sampled normal fields of geometry and maintained these in texture maps during simplification. While these approaches all differ significantly from our interests here, it is clear that maps of this and related nature are of great interest in many contexts.

In independent work, Lee et al. pursue a goal similar to ours [17]. They introduce displaced subdivision surfaces which can be seen as a two level normal mesh. Because only two levels are used, the base domain typically contains more triangles than in our case. Also the normal offsets are oversampled while in our case, the normal offsets are critically sampled.

2 Normal Polylines

Before we look at surfaces and normal meshes, we introduce some of the concepts using curves and normal polylines. A curve in the plane is described by a pair of parametric functions $\mathbf{s}(t) = (x(t), y(t))$ with $t \in [0, 1]$. We would like to describe the points on the curve with a single scalar function. In practice one uses polylines to approximate the function. Let $\mathbf{l}(\mathbf{p}, \mathbf{p}')$ be the linear segment between the points $\mathbf{p}$ and $\mathbf{p}'$. A standard way to build a polyline multiresolution approximation is to sample the curve at points $\mathbf{s}_{j,k}$ where $\mathbf{s}_{j,k} = \mathbf{s}_{j+1,2k}$ and define the jth level approximation as

$$\mathbf{L}_j = \bigcup_{0 \le k < 2^j} \mathbf{l}(\mathbf{s}_{j,k}, \mathbf{s}_{j,k+1}).$$

To move from $\mathbf{L}_j$ to $\mathbf{L}_{j+1}$ we need to insert the points $\mathbf{s}_{j+1,2k+1}$ (Figure 2, left). Clearly this requires two scalars: the two coordinates of $\mathbf{s}_{j+1,2k+1}$. Alternatively one could compute the difference $\mathbf{s}_{j+1,2k+1} - \mathbf{m}$ between the new point and some predicted point $\mathbf{m}$, say the midpoint of the neighboring points $\mathbf{s}_{j,k}$ and $\mathbf{s}_{j,k+1}$. This detail has a tangential component $\mathbf{m} - \mathbf{b}$ and a normal component $\mathbf{b} - \mathbf{s}_{j+1,2k+1}$. The normal component is the *geometric* information while the tangential component is the *parameter* information. The way to build polylines that can be described with one

Figure 2: *Removing one point $\mathbf{s}_{j+1,2k+1}$ in a polyline multiresolution and recording the difference with the midpoint $\mathbf{m}$. On the left a general polyline where the detail has both a normal and a tangential component. On the right a normal polyline where the detail is purely normal.*

scalar per point, is to make sure that the parameter information is always zero, i.e., $\mathbf{b} = \mathbf{m}$, see Figure 2, right. If the triangle $\mathbf{s}_{j,k}$, $\mathbf{s}_{j+1,2k+1}$, $\mathbf{s}_{j,k+1}$ is Isosceles, there is no parameter information. Consequently we say that a polyline is normal if a multiresolution structure exists where every removed point forms an Isosceles triangle with its neighbors. Then there is zero parameter information and the polyline can be represented with one scalar per point, namely the normal component of the associated detail.

For a general polyline the removed triangles are hardly ever exactly Isosceles and hence the polyline is not normal. Below we describe a procedure to build a normal polyline approximation for any continuous curve. The easiest is to start building Isosceles triangles from the coarsest level. Start with the first base $\mathbf{l}(\mathbf{s}_{0,0}, \mathbf{s}_{0,1})$, see Figure 3. Next take its midpoint and check where the normal direction crosses the curve. Because the curve is continuous, there has to be at least one such point. If there are multiple pick any one.

Figure 3: *Construction of a normal polyline. We start with the coarsest level and each time check where the normal to the midpoint crosses the curve. For simplicity only the indices of the* $\mathbf{s}_{j,k}$ *points are shown and only certain segments are subdivided. The polyline* $(0,0)-(2,1)-(3,3)-(1,1)-(0,1)$ *is determined by its endpoints and three scalars, the heights of the Isosceles triangles.*

Call this point $\mathbf{s}_{1,1}$ and define the first triangle. Now split the curve into two parts and repeat the procedure on each subcurve. Each time $\mathbf{s}_{j+1,2k+1}$ is found where the normal to the midpoint of $\mathbf{s}_{j,k}$ and $\mathbf{s}_{j,k+1}$ crosses the portion of the curve between $\mathbf{s}_{j,k}$ and $\mathbf{s}_{j,k+1}$. Thus any continuous curve can be approximated arbitrarily closely with a normal polyline. The result is a series of polylines $\mathbf{L}_j$ all of which are normal with respect to midpoint prediction. Effectively each level is parameterized with respect to the one coarser level. Because the polylines are normal, only a single scalar value, the normal component, needs to be recorded for each point. We have a polyline with no parameter information.

One can also consider normal polylines with respect to fancier predictors. For example one could compute a base point and normal estimate using the well known 4 point rule. Essentially any predictor which only depends on the coarser level is allowed. For example one can also use irregular schemes [4]. Also one does not need to follow the standard way of building levels by downsampling every other point, but instead could take any ordering. This leads to the following definition of a normal polyline:

Definition 1 *A polyline is normal if a removal order of the points exists such that each removed point lies in the normal direction from a base point, where the normal direction and base point only depend on the remaining points.*

Hence a normal polyline is completely determined by a scalar component per vertex.

Normal polylines are closely related to certain well known fractal curves such as the Koch Snowflake[1], see Figure 4. Here each time a line segment is divided into three subsegments. The left and right get a normal coefficient of zero, while the middle receives a normal coefficient such that the resulting triangle is equilateral. Hence the polylines leading to the snowflake are normal with respect to midpoint subdivision.

Figure 4: *Four normal polylines converging to the Koch snowflake.*

[1]Niels Fabian Helge von Koch (Sweden, 1870-1924)

There is also a close connection with wavelets. The normal coefficients can be seen as a piecewise linear wavelet transform of the original curve. Because the tangential components are always zero there are half as many wavelet coefficients as there are original scalar coefficients. Thus one saves 50% memory right away. In addition of course the wavelets have their usual decorrelation properties. In the functional case the above transform corresponds to an unlifted interpolating piecewise linear wavelet transform as introduced by Donoho [6]. There it is shown that interpolating wavelets with no primal, but many dual moments are well suited for smooth functions. Unlike in the function setting, not all wavelets from the same level j have the same physical scale. Here the scale of each coefficient is essentially the length of the base of its Isosecles triangle.

3 Normal Meshes

We begin by establishing terminology. A triangle mesh $\mathcal{M}$ is a pair $(\mathcal{P}, \mathcal{K})$, where $\mathcal{P}$ is a set of N point positions $\mathcal{P} = \{\mathbf{p}_i = (x_i, y_i, z_i) \in \mathbf{R}^3 \mid 1 \leq i \leq N\}$, and $\mathcal{K}$ is an *abstract simplicial complex* which contains all the topological, i.e., adjacency information. The complex $\mathcal{K}$ is a set of subsets of $\{1, \ldots, N\}$. These subsets come in three types: vertices $\{i\}$, edges $\{i, j\}$, and faces $\{i, j, k\}$. Two vertices i and j are *neighbors* if $\{i, j\} \in \mathcal{E}$. The 1-ring neighbors of a vertex i form a set $\mathcal{V}(i) = \{j \mid \{i, j\} \in \mathcal{E}\}$.

We can derive a definition of normal triangle meshes inspired by the curve case. Consider a hierarchy of triangle meshes $\mathcal{M}_j$ built using mesh simplification with vertex removals. These meshes are nested in the sense that $\mathcal{P}_j \subset \mathcal{P}_{j+1}$. Take a removed vertex $\mathbf{p}_i \in \mathcal{P}_{j+1} \setminus \mathcal{P}_j$. For the mesh to be normal we need to be able to find a base point $\mathbf{b}$ and normal direction N that only depend on $\mathcal{P}_j$, so that $\mathbf{p}_i - \mathbf{b}$ lies in the direction N. This leads to the following definition.

Definition 2 *A mesh* $\mathcal{M}$ *is normal in case a sequence of vertex removals exists so that each removed vertex lies on a line defined by a base point and normal direction which only depends on the remaining vertices.*

Thus a normal mesh can be described by a small base domain and one scalar coefficient per vertex.

As in the curve case, a mesh is in general not normal. The chance that the difference between a removed point and a predicted base point lies exactly in a direction that only depends on the remaining vertices is essentially zero. Hence the only way to obtain a normal mesh is to change the triangulation. We decide to use semi-regular meshes, i.e., meshes whose connectivity is formed by successive quadrisection of coarse base domain faces.

As in the curve setting, the way to build a normal mesh is to start from the coarse level or base domain. For each new vertex we compute a base point as well as a normal direction and check where the line defined by the base point and normal intersects the surface. The situation, however, is much more complex than in the curve case for two reasons: (1) There could be no intersection point. (2) There could be many intersection points, but only one correct one.

In case there are no intersection points, strictly speaking no fully normal mesh can be built from this base domain. If that happens, we relax the definition of normal meshes some and allow a small number of cases where the new points do not lie in the normal direction. Thus the algorithm needs to find a suitable non-normal location for the new point. In case there are many intersection points the algorithm needs to figure out which one is the right one. If the wrong one is chosen the normal mesh will start folding over itself or leave creases. Any algorithm which blindly picks an intersection point is doomed.

Parameterization In order to find the right piercing point or suggest a good alternate, one needs to be able to easily navigate around the surface. The way to do this is to build a smooth parameterization of the surface region of interest. This is a basic building block of our algorithm. Several parameterization methods have been proposed and our method takes components from each of them: mesh simplification and polar maps from MAPS [18], patchwise relaxation from [9], and a specific smoothness functional similar to the one used in [10] and [20]. The algorithm will use local parameterizations which need to be computed fast and robustly. Most of them are temporary and are quickly discarded unless they can be used as a starting guess for another parameterization.

Consider a region $\mathcal{R}$ of the mesh homeomorphic to a disc that we want to parameterize onto a convex planar region $\mathcal{B}$, i.e., find a bijective map $u : \mathcal{R} \rightarrow \mathcal{B}$. The map u is fixed by a boundary condition $\partial\mathcal{R} \rightarrow \partial\mathcal{B}$ and minimizes a certain energy functional. Several functionals can be used leading to, e.g., conformal or harmonic mappings. We take an approach based on the work of Floater [10]. In short, the function u needs to satisfy the following equation in the interior:

$$u(\mathbf{p}_i) = \sum_{k \in \mathcal{V}(i)} \alpha_{ik} u(\mathbf{p}_k), \tag{1}$$

where $\mathcal{V}(i)$ is the 1-ring neighborhood of the vertex i and the weights α_{ik} come from the shape-preserving parameterization scheme [10]. The main advantage of the Floater weights is that they are always positive, which, combined with the convexity of the parametric region, guarantees that no triangle flipping can occur within the parametric domain. This is crucial for our algorithm. Note that this is not true in general for harmonic maps which can have negative weights. We use the iterative biconjugate gradient method [12] to obtain the solution to the system (1). Given that we often have a good starting guess this converges quickly.

Algorithm Our algorithm consists of 7 stages which are described below, some of which are shown for the molecule model in Figure 5. The molecule is a highly detailed and curved model. Any naive procedure for finding normal meshes is very unlikely to succeed.

The first four stages of the algorithm prepare the ground for the piercing procedure and build the net of curves splitting the original mesh into triangular patches that are in one-to-one correspondence with the faces of the base mesh, i.e., the coarsest level of the semi-regular mesh we build.

1. Mesh simplification: We use the Garland-Heckbert [11] simplification based on half-edge collapses to create a mesh hierarchy $(\mathcal{P}_j, \mathcal{K}_j)$. We use the coarsest level $(\mathcal{P}_0, \mathcal{K}_0)$ as an initial guess for our base domain $(\mathcal{Q}_0, \mathcal{K}_0)$. The first image of Figure 5 shows the base domain for the molecule.

2. Building an initial net of curves: The purpose of this step is to connect the vertices of the base domain with a net of non intersecting curves on the different levels of the mesh simplification hierarchy. This can easily be done using the MAPS parameterization [18]. MAPS uses polar maps to build a bijection between a 1-ring and its retriangulation after the center vertex is removed. The concatenation of these maps is a bijective mapping between different levels $(\mathcal{P}_j, \mathcal{K}_j)$ in the hierarchy. The desired curves are simply the image of the base domain edges under this mapping. Because of the bijection no intersection can occur. Note that the curves start and finish at a vertex of the base domain, but need not follow the edges of the finer triangulation, i.e., they can cut across triangles. These curves define a network of triangular shaped patches corresponding to the base domain triangles. Later we will adjust these curves on some intermediate level and again use MAPS to propagate these changes to other levels. The top middle image of Figure 5 shows these curves for some intermediate level of the hierarchy.

3. Fixing the global vertices: A normal mesh is almost completely determined by the base domain. One has to choose the base domain vertices $\mathcal{Q}_0$ very carefully to reduce the number of non-normal vertices to a minimum. The coarsest level of the mesh simplification $\mathcal{P}_0$ is only a first guess. In this section we describe a procedure for repositioning the global vertices $\mathbf{q}_i$ with $\{i\} \in \mathcal{K}_0$. We impose the constraint that the $\mathbf{q}_i$ needs to coincide with some vertex $\mathbf{p}_k$ of the original mesh, but not necessarily $\mathbf{p}_i$.

The repositioning is typically done on some intermediate level j. Take a base domain vertex $\mathbf{q}_i$. We build a parameterization from the patches incident to vertex $\mathbf{q}_i$ to a disk in the plane, see Figure 6. Boundary conditions are assigned using arclength parameterization, and parameter coordinates are iteratively computed for each level j vertex inside the shaded region. It is now easy to replace the point $\mathbf{q}_i$ with any level point from $\mathcal{P}_j$ in the shaded region. In particular we let the new $\mathbf{q}'_i$ be the point of $\mathcal{P}_j$ that in the parameter domain is closest to the center of the disk. The exact center of the disk, in general, does not correspond to a vertex of the mesh.

Once a new position $\mathbf{q}'_i$ is chosen, the curves can be redrawn by taking the inverse mapping of straight lines from the new point in the parameter plane. One can keep iterating this procedure, but we found that if suffices to cycle once through all base domain vertices.

We also provide for a user controlled repositioning. Then the user can replace the center vertex with any $\mathcal{P}_j$ point in the shaded region. The algorithm again uses the parameterization to recompute the curves from that point.

The top right of Figure 5 shows the repositioned vertices. Notice how some of them like the rightmost one have moved considerably.

Figure 6: *Base domain vertex repositioning. Left: original patches around $\mathbf{q}_i$, middle: parameter domain, right: repositioned $\mathbf{q}_i$ and new patch boundaries. This is replaced with the vertex whose parameter coordinate are the closest to the center. The inverse mapping (right) is used to find the new position $\mathbf{q}'_i$ and the new curves.*

4. Fixing the global edges: The image of the global edges on the finest level will later be the patch boundaries of the normal mesh. For this reason we need to improve the smoothness of the associated curves at the finest level. We use a procedure similar to [9]. For each base domain edge $\{i, k\}$ we consider the region formed on the finest level mesh by its two incident patches. Let l and m be the opposing global vertices. We then compute a parameter function ρ within the diamond-shaped region of the surface. The boundary condition is set as $\rho(\mathbf{q}_i) = \rho(\mathbf{q}_k) = 0$, $\rho(\mathbf{q}_l) = 1$, $\rho(\mathbf{q}_m) = -1$, with linear variation along the edges. We then compute the parameterization and let its zero level set be our new curve. Again one could iterate this procedure till convergence but in practice one cycle suffices. The curves of the top right image in Figure 5 are the result of the curve smoothing on the finest level.

Note that a similar result can be achieved by allowing the user to position the global vertices and draw the boundaries of the patches manually. Indeed, the following steps of the algorithm do not depend on how the initial net of surface curves is produced.

Figure 5: *The entire procedure shown for the molecule model. 1. Base domain. 2. Initial set of curves. 3. Global vertex repositioning 4. Initial Parameterization 5. Adjusting parameterization 6. Final normal mesh. (HIV protease surface model courtesy of Arthur Olson, The Scripps Research Institute)*

5. Initial parameterization: Once the global vertices and edges are fixed, one can start filling in the interior. This is done by computing the parameterization of each patch to a triangle while keeping the boundary fixed. The parameter coordinates from the last stage can serve as a good initial guess. We now have a smooth global parameterization. This parameterization is shown in the bottom left of Figure 5. Each triangle is given a triangular checkerboard texture to illustrate the parameterization.

6. Piercing: In this stage of the algorithm we start building the actual normal mesh. The canonical step is for a new vertex of the semi-regular mesh to find its position on the original mesh. In quadrisection every edge of level j generates a new vertex on level $j+1$. We first compute a base point using interpolating Butterfly subdivision [8] [24] as well as an approximation of the normal. This defines a straight line. This line may have multiple intersection points in which case we need to find the right one, or it could have none, in which case we need to come up with a good alternate.

Suppose that we need to produce the new vertex $\mathbf{q}$ that lies halfway along the edge $\{\mathbf{a}, \mathbf{c}\}$ with incident triangles $\{\mathbf{a}, \mathbf{c}, \mathbf{b}\}$ and $\{\mathbf{c}, \mathbf{a}, \mathbf{d}\}$, see Figure 7. Let the two incident patches form the region $\mathcal{R}$.

Build the straight line L defined by the base point $\mathbf{s}$ predicted by the Butterfly subdivision rule [24] and the direction of the normal computed from the coarser level points. We find all the intersection points of L with the region $\mathcal{R}$ by checking all triangles inside.

If there is no intersection we take the point $\mathbf{v}$ that lies midway between the points $\mathbf{a}$ and $\mathbf{c}$ in the parameter domain: $u(\mathbf{v}) = (u(\mathbf{a}) + u(\mathbf{c}))/2$. This is the same point a standard parameterization based remesher would use. Note that in this case the detail vector is non-normal and its three components need to be stored.

In the case when there exist several intersections of the mesh region $\mathcal{R}$ with the piercing line L we choose the intersection point that is closest to the point $u(\mathbf{v})$ in the parameter domain. Let us denote by $u(\mathbf{q})$ the parametric coordinates of that piercing point.

Figure 7: *Upper left: piercing, the Butterfly point is* $\mathbf{s}$*, the surface is pierced at the point* $\mathbf{q}$*, the parametrically suggested point* $\mathbf{v}$ *lies on the curve separating two regions of the mesh. Right: parameter domain, the pierced point falls inside the aperture and gets accepted. Lower left: the parameterization is adjusted to let the curve pass through* q*.*

We accept this point as a valid point of the semi-regular mesh if $\|u(\mathbf{q}) - u(\mathbf{v})\| < \kappa \|u(\mathbf{a}) - u(\mathbf{v})\|$, where κ is an "aperture" parameter that specifies how much the parameter value of a pierced

point is allowed to deviate from the center of the diamond. Otherwise, the piercing point is rejected and the mesh takes the point with the parameter value $u(\mathbf{v})$, resulting in a non-normal detail.

7. Adjusting the parameterization: Once we have a new piercing point, we need to adjust the parameterization to reflect this. Essentially, the adjusted parameterization u should be such that the piercing point has the parameters $u(\mathbf{v}) =: u(\mathbf{q})$. When imposing such an isolated point constraint on the parameterization, there is no mathematical guarantee against flipping. Hence we draw a new piecewise linear curve through $u(\mathbf{q})$ in the parameter domain. This gives a new curve on the surface which passes through $\mathbf{q}$, see Figure 7. We then recompute the parameterization for each of the patches onto a triangle separately. We use a piecewise linear boundary condition with the half point at $\mathbf{q}$ on the common edge.

When all the new midpoints for the edges of a face of level j are computed, we can build the faces of level $j+1$. This is done by drawing three new curves inside the corresponding region of the original mesh, see Figure 8. Before that operation happens we need to ensure that a valid parameterization is available within the patch. The patch is parameterized onto a triangle with three piecewise linear boundary conditions each time putting the new points at the midpoint. Then the new points are connected in the parameter domain which allows us to draw new finer level curves on the original mesh. This produces a metamesh similar to [16], so that the new net of curves replicates the structure of the semi-regular hierarchy on the surface of the original. The construction of the semi-regular mesh can be done adaptively with the error driven procedure from MAPS [18]. An example of parameterization adjustment after two levels of adaptive subdivision is shown in the bottom middle of Figure 5. Note that as the regions for which we compute parameterizations become smaller, the starting guesses are better and the solver convergence becomes faster and faster.

Figure 8: *Face split: Quadrisection in the parameter plane (left) leads to three new curves within the triangular patch (right).*

The aperture parameter κ of the piercing procedure provides control over how much of the original parameterization is preserved in the final mesh and consequently, how many non-normal details will appear. At $\kappa = 0$ we build a *non-normal* mesh entirely based on the original global parameterization. At $\kappa = 1$ we attempt to build a purely *normal* mesh independent of the parameterization. In our experience, the best results were achieved when the aperture was set low (0.2) at the coarsest levels, and then increased to 0.6 on finer levels. On the very fine levels of the hierarchy, where the geometry of the semi-regular meshes closely follows the original geometry, one can often simply use a naive piercing procedure without parameter adjustment.

One may wonder if the continuous readjustment of parameterizations is really necessary. We have tried the naive piercing procedure without parameterization from the base domain and found that it typically fails on all models. An example is Figure 9 which shows 4 levels of naive piercing for the torus starting from a 102 vertex base mesh. Clearly, there are several regions with flipped and self-intersecting triangles. The error is about 20 times larger than the true normal mesh.

Figure 9: *Naive piercing procedure. Clearly, several regions have flipped triangles and are self-intersecting.*

Dataset	Size	Base	Normal mesh size	Not normal (%)	% L^2 error	Time (min)
Feline	49864	156	40346	729 (1.8%)	.015	4
Molecule	10028	37	9521	270 (2.8%)	.075	1.5
Rabbit	16760	33	8235	196 (2.4%)	.037	2
Torus3	5884	98	5294	421 (8.0%)	.03	3
Skull	20002	112	25376	817 (3.2%)	.02	2.5
Horse	48485	234	59319	644 (1.1%)	.004	6.8

Table 1: *Summary of normal meshing results for different models. The normal mesh is computed adaptively and contains roughly the same number of triangles as the original mesh. The relative L^2 errors are computed with the I.E.I.-CNR Metro tool. The times are reported on a 700MHz Pentium III machine.*

4 Results

We have implemented the algorithms described in the preceding section, and performed a series of experiments in which normal meshes for various models were built. The summary of the results is given in Table 1. As we can see from the table, the normal semi-regular meshes have very high accuracy and hardly any non normal details.

One interesting feature of our normal meshing procedure is the following: while the structure of patches comes from performing simplification there are far fewer restrictions on how coarse the base mesh can be. Note for example that the skull in Figure 1 was meshed with the tetrahedron as base mesh. This is largely due to the robust mesh parameterization techniques used in our approach.

Figure 10 shows normal meshes for rabbit, torus, feline, and skull, as well as close-up of feline (bottom left) normal mesh. Note how smooth the meshes are across global edges and global vertices. This smoothness mostly comes from the normality, not the parameterization. It is thus an intrinsic quantity.

One of the most interesting observations coming from this work is that locally the normal meshes do not differ much from the non-normal ones, while offering huge benefits in terms of efficiency of representation. For example, Table 2 shows how the "aperture parameter" κ that governs the construction of normal meshes affects the number of detail coefficients with non-trivial tangential components for the model of the three hole torus (these numbers are typical for other models as well). In particular, we see that already a very modest acceptance strategy ($\kappa = 0.2$) gets rid of more than 90% of the tangential components in the remeshed model, and the more aggressive strategies offer even more benefits without affecting the error of the representation.

5 Summary and Conclusion

In this paper we introduce the notion of *normal meshes*. Normal meshes are multiresolution meshes in which vertices can be found in the normal direction, starting from some coarse level. Hence only one scalar per vertex needs to be stored. We presented a robust

κ	normal	error (10^{-4})
0	0%	1.02
0.2	91.9%	1.05
0.4	92.4%	1.04
best	98.3%	1.02

Table 2: *The relation between the acceptance strategy during the piercing procedure and the percentage of perfectly normal details in the hierarchy. The original model has 5884 vertices, all the normal meshes have 26002 vertices (4 levels uniformly), and the base mesh contained 98 vertices. The best strategy in the last line used* $\kappa = 0.2$ *on the first three levels and afterward always accepted the piercing candidates.*

algorithm for computing normal semi-regular meshes of any input mesh and showed that it produces very smooth triangulations on a variety of input models.

It is clear that normal meshes have numerous applications. We briefly discuss a few.

Compression Usually a wavelet transform of a standard mesh has three components which need to be quantized and encoded. Information theory tells us that the more non uniform the distribution of the coefficients the lower the first order entropy. Having 2/3 of the coefficients exactly zero will further reduce the bit budget. From an implementation viewpoint, we can almost directly hook the normal mesh coefficients up to the best known scalar wavelet image compression code.

Filtering It has been shown that operations such as smoothing, enhancement, and denoising can be computed through a suitable scaling of wavelet coefficients [7]. In a normal mesh any such algorithm will require only 1/3 as many computations. Also large scaling coefficients in a standard mesh will introduce large tangential components leading to flipped triangles. In a normal mesh this is much less likely to happen.

Texturing Normal semi-regular meshes are very smooth inside patches, across global edges, and around global vertices even when the base domain is exceedingly coarse, cf. the skull model. The implied parameterizations are highly suitable for all types of mapping applications.

Rendering Normal maps are a very powerful tool for decoration and enhancement of otherwise smooth geometry. In particular in the context of bandwidth bottlenecks it is attractive to be able to download a normal map into hardware and only send smooth coefficient updates for the underlying geometry. The normal mesh transform effectively solves the associated inverse problem: construct a normal map for a given geometry.

The concept of normal meshes opens up many new areas of research.

- Our algorithm uses interpolating subdivision to find the base point. Building normal meshes with respect to approximating subdivision is not straightforward.
- The theoretical underpinnings of normal meshes need to be studied. Do continuous variable normal descriptions of surfaces exist? What about stability? What about connections with curvature normal flow which acts to reduce normal information?
- We only addressed semi-regular normal meshes here, while the definition allows for the more flexible setting of progressive irregular mesh hierarchies.
- Purely scalar compression schemes for geometry need to be compared with existing coders.
- Generalize normal meshes to higher dimensions. It should be possible to represent a M dimensional manifold in N dimensions with $N - M$ variables as opposed to the usual N.
- The current implementation only works for surfaces without boundaries and does not deal with feature curves. We will address these issues in our future research.

Acknowledgments This work was supported in part by NSF (ACI-9624957, ACI-9721349, DMS-9874082, DMS 9872890), Alias|Wavefront, a Packard Fellowship, and a Caltech Summer Undergraduate Research Fellowship (SURF). Special thanks to Nathan Litke for his subdivision library, to Andrei Khodakovsky, Mathieu Desbrun, Adi Levin, Arthur Olson, and Zoë Wood for helpful discussions, Chris Johnson for the use of the SGI-Utah Visual Supercomputing Center resources, and to Cici Koenig for production help. Datasets are courtesy Cyberware, Headus, The Scripps Research Institute, and University of Washington.

References

[1] CERTAIN, A., POPOVIC, J., DEROSE, T., DUCHAMP, T., SALESIN, D., AND STUETZLE, W. Interactive Multiresolution Surface Viewing. *Proceedings of SIGGRAPH 96* (1996), 91–98.

[2] COHEN, J., OLANO, M., AND MANOCHA, D. Appearance-Preserving Simplification. *Proceedings of SIGGRAPH 98* (1998), 115–122.

[3] COOK, R. L. Shade trees. *Computer Graphics (Proceedings of SIGGRAPH 84) 18*, 3 (1984), 223–231.

[4] DAUBECHIES, I., GUSKOV, I., AND SWELDENS, W. Regularity of Irregular Subdivision. *Constr. Approx. 15* (1999), 381–426.

[5] DESBRUN, M., MEYER, M., SCHRÖDER, P., AND BARR, A. H. Implicit Fairing of Irregular Meshes Using Diffusion and Curvature Flow. *Proceedings of SIGGRAPH 99* (1999), 317–324.

[6] DONOHO, D. L. Interpolating wavelet transforms. Preprint, Department of Statistics, Stanford University, 1992.

[7] DONOHO, D. L. Unconditional Bases are Optimal Bases for Data Compression and for Statistical Estimation. *Appl. Comput. Harmon. Anal. 1* (1993), 100–115.

[8] DYN, N., LEVIN, D., AND GREGORY, J. A. A Butterfly Subdivision Scheme for Surface Interpolation with Tension Control. *ACM Transactions on Graphics 9*, 2 (1990), 160–169.

[9] ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., AND STUETZLE, W. Multiresolution Analysis of Arbitrary Meshes. *Proceedings of SIGGRAPH 95* (1995), 173–182.

[10] FLOATER, M. S. Parameterization and Smooth Approximation of Surface Triangulations. *Computer Aided Geometric Design 14* (1997), 231–250.

[11] GARLAND, M., AND HECKBERT, P. S. Surface Simplification Using Quadric Error Metrics. In *Proceedings of SIGGRAPH 96*, 209–216, 1996.

[12] GOLUB, G. H., AND LOAN, C. F. V. *Matrix Computations*, 2nd ed. The John Hopkins University Press, Baltimore, 1983.

[13] GUSKOV, I., SWELDENS, W., AND SCHRÖDER, P. Multiresolution Signal Processing for Meshes. *Proceedings of SIGGRAPH 99* (1999), 325–334.

[14] KHODAKOVSKY, A., SCHRÖDER, P., SWELDENS, W. Progressive Geometry Compression. *Proceedings of SIGGRAPH 2000* (2000).

[15] KRISHNAMURTHY, V., AND LEVOY, M. Fitting Smooth Surfaces to Dense Polygon Meshes. *Proceedings of SIGGRAPH 96* (1996), 313–324.

[16] LEE, A. W. F., DOBKIN, D., SWELDENS, W., AND SCHRÖDER, P. Multiresolution Mesh Morphing. *Proceedings of SIGGRAPH 99* (1999), 343–350.

[17] LEE, A. W. F., MORETON, H., HOPPE, H. Displaced Subdivision Surfaces. *Proceedings of SIGGRAPH 00* (2000).

[18] LEE, A. W. F., SWELDENS, W., SCHRÖDER, P., COWSAR, L., AND DOBKIN, D. MAPS: Multiresolution Adaptive Parameterization of Surfaces. *Proceedings of SIGGRAPH 98* (1998), 95–104.

[19] LEVOY, M. The Digital Michelangelo Project. In *Proceedings of the 2nd International Conference on 3D Digital Imaging and Modeling*, October 1999.

[20] LÉVY, B., AND MALLET, J. Non-Distorted Texture Mapping for Sheared Triangulated Meshes. *Proceedings of SIGGRAPH 98* (1998), 343–352.

[21] LOUNSBERY, M., DEROSE, T. D., AND WARREN, J. Multiresolution Analysis for Surfaces of Arbitrary Topological Type. *ACM Transactions on Graphics 16*, 1 (1997), 34–73. Originally available as TR-93-10-05, October, 1993, Department of Computer Science and Engineering, University of Washington.

[22] SCHRÖDER, P., AND SWELDENS, W. Spherical Wavelets: Efficiently Representing Functions on the Sphere. *Proceedings of SIGGRAPH 95* (1995), 161–172.

[23] ZORIN, D., AND SCHRÖDER, P., Eds. *Subdivision for Modeling and Animation*. Course Notes. ACM SIGGRAPH, 1999.

[24] ZORIN, D., SCHRÖDER, P., AND SWELDENS, W. Interpolating Subdivision for Meshes with Arbitrary Topology. *Proceedings of SIGGRAPH 96* (1996), 189–192.

[25] ZORIN, D., SCHRÖDER, P., AND SWELDENS, W. Interactive Multiresolution Mesh Editing. *Proceedings of SIGGRAPH 97* (1997), 259–268.

Figure 10: *Colorplate.*

Point-Based Computer Graphics and Visualization

SIGGRAPH 2001
Course 33

Hanspeter Pfister, MERL

Trends in Computer Graphics

- Rich and realistic graphics with detailed models in complex scenes.
- Sophisticated appearance models that go beyond texture-mapped objects.

Issues

- Our capacity to create high quality models is limited.
  - Art production is the bottleneck.
  - Real-world object appearances can be very complex.
- Laser-range scanning and computer vision are a means for capturing the real world.
- Scanned high-resolution models contain a lot of small triangles.
  - Project to < 1 pixel.
- To accurately capture appearance requires to store a lot of data per triangle.

Point-Based Computer Graphics

- Point-based models allow us to efficiently acquire and display complex 3D objects.
- We capture shape and appearance from images.
- We call these point-based models “3D images”.

Our Goals

- Develop a systems architecture to capture and display complex models with:
  - automatic acquisition,
  - transmission,
  - and interactive rendering.
- Display complex object appearances (specularity and transparency) under novel lighting conditions.
- Render point-based volume and surface data using the same framework.

Outline

- 3D Images
  - Acquisition
  - Display
    - Surfels as Rendering Primitives
    - EWA Volume and Surface Splatting
- Conclusions

Acquisition

- From images / video of real objects.
  - With mounted or handheld camera.

The System Prototype

Three System Implementations

V1.0 (July-Oct 2000)

V2.0 (Oct-Dec 2000)

Current System

Current System

Current System

Image-Based Visual Hull

[Matusik *et al*., SIGGRAPH 2000]

Why use Visual Hulls?

- They rely on the simplest CV algorithms.
- They can be computed robustly.
- They can be computed efficiently.

Approximate Geometry

- The approximate visual hull is augmented by radiance data to render concavities, reflections, and transparency.

Surface Light Field

- A surface light field is a function that assigns a color to each ray originating on a surface. [Wood *et al*., 2000]

- We store up to 100 rays and colors per surface point. [Matusik, Pfister, Beardsley, McMillan, 2001]

Shading Algorithm

- A view-dependent strategy.

IBVH Acquisition

- Advantages:
  - Very robust for all kinds of objects.
  - Simultaneous acquisition of shape and appearance.
- Limitations:
  - Requires very good camera calibration.
  - Nearest neighbor shading algorithm can be improved.

Outline

- 3D Images
  - Acquisition
  - Display
    - Surfels as Rendering Primitives
    - EWA Volume and Surface Splatting
- Conclusions

Related Work

- Texture mapping
- Image-based Rendering
- Volume Graphics (“surface voxels”)
- Point Sample Rendering
  - Animatek, www.animatek.com
  - Levoy and Whitted, 1985
  - Grossman and Dally, 1999
  - Rusinkiewicz and Levoy, 2000
- Texture Mapping
  - Heckbert, 1986

Surfels as Rendering Primitives

- Typical triangles in complex 3D geometry models project to < 1 pixel.
- Point samples (surfels) allow us to efficiently render complex 3D objects.

[Pfister, Zwicker, van Baar, Gross, SIGGRAPH 2000]

Data Structure - LDC Tree

- Hierarchical octree-like data structure for progressive transmission and rendering.

Level 2

Level 1

Level 0

Rendering

- Traverse tree top to bottom.
- Visibility culling of blocks outside the viewing frustum.
- Forward projection of surfels from object to screen space using incremental forward warping.
  [Grossman, Dally 99]
- Visibility resolution using a z-buffer.

Potential Problems

- After projection the image may contain holes.

- Texture and edge aliasing.

Surface Texture Function

- Texture function on the surface of a point-based object is a sum of 2D reconstruction kernels.

Rendering Framework

Rendering Framework

- Express texture functions as sums of reconstruction kernels.
- Forward projection of reconstruction kernels to screen space
  - Footprint.
- Bandlimit the continuous image function by low-pass filtering the individual reconstruction kernels.
  - Footprint ⊗ low-pass filter = resampling kernel.
- Accumulation of resampling kernels in screen space.

Surface Splatting

Volume Splatting [Westover, 1989]

- Volume is a field of 3D reconstruction kernels.
  - One kernel at each voxel.
- Pre-integrate the 3D reconstruction kernels into 2D footprints.

$$q_k(x,y) = \int_{\Re} r_k(x,y,\xi)\,d\xi$$

Volume Resampling Filter

- We make several simplifying assumptions to combine the low-pass filter with the footprint.
  - Constant extinction and emission inside each kernel.
- Resulting resampling filters are rendered in back-to-front order and blended into the image buffer.

Extinction

$$(I_\lambda \otimes h)(\mathbf{x}) \approx \sum_k c_{\lambda k} o_k g_k (q_k \otimes h)(\mathbf{x})$$

Emission

2D Resampling Filter
(Footprint ⊗ Low-Pass Filter)

Elliptical Gaussian Kernels

- We choose elliptical Gaussians as reconstruction kernels and low-pass filters.
- They are closed under affine mappings and convolution.

- The integration of a 3D Gaussian is a 2D Gaussian.

EWA Resampling Filter

- We can compute an analytic formulation of the EWA resampling filter in screen space.
- The EWA resampling filter combines the footprint with a screen space (Gaussian) low-pass filter.

Footprint Filter

$$\rho(\mathbf{x}) = (q \otimes h)(\mathbf{x}) = \frac{1}{\left|W^{-1}\right|} G(\underline{W^T V^q W} + \underline{V^h})(\mathbf{x} - C)$$

Warp and Projection Matrix

3D reconstruction Kernel

Algorithm

For each point in BTF order
- Compute the warp and projection matrix
- Project point to screen space
- Compute the resampling filter
- Rasterize resampling filter
- Blend pixels into image

For each pixel in the image
- Shade pixel

EWA Volume Splatting

- Volume renderings using the EWA resampling filter. [Zwicker, Pfister, van Baar, Gross, 2001]

- Compare to uniformly scaled footprints [Swan et al. 1997]

EWA Volume and Surface Splatting

- Render volume iso-surfaces with flattened kernels for better quality.

- Volume rendering and surface rendering using the same algorithm.

EWA Surface Splatting

- Texture anti-aliasing and edge anti-aliasing.
- Transparency.

[Zwicker, Pfister, van Baar, Gross, SIGGRAPH 2001]

EWA Volume and Surface Splatting

- Advantages:
  - Correct antialiasing without excessive blurring.
  - Works for rectilinear and irregular volumes.
  - Combined visualization of volume and surface data.
  - Amenable to parallelism and hardware acceleration.
- Limitations:
  - Currently about 0.5 to 15 seconds per frame.
  - Not efficient for flat surfaces with uniform color.

Conclusions

- Image-based visual hulls combined with surface lightfields provide realistic 3D object models.
- EWA volume and surface splatting provides a unifying rendering framework for volume and surface samples.
- Point-based computer graphics and visualization is able to capture and render complex objects efficiently.

Future Work

- Improved 3D scanners.
- Compression and progressive transmission.
- Deformable point-sample models.
- Custom hardware for EWA splatting.
- Virtual humans and 3D teleconferencing.
- Virtual scenes and mixed reality.

Acknowledgements

- Colleagues:
  - MERL: Jeroen van Baar, Paul Beardsley, Bill Yerazunis, Darren Leigh, Ron Perry.
  - MIT: Wojciech Matusik, Chris Buehler, Leonard McMillan.
  - ETH Zürich: Matthias Zwicker, Markus Gross.
- Bibliography and Papers:
  - http://www.merl.com/people/pfister/

Surface Splatting

Matthias Zwicker * Hanspeter Pfister † Jeroen van Baar† Markus Gross*

Figure 1: *Surface splatting of a scan of a human face, textured terrain, and a complex point-sampled object with semi-transparent surfaces.*

Abstract

Modern laser range and optical scanners need rendering techniques that can handle millions of points with high resolution textures. This paper describes a point rendering and texture filtering technique called *surface splatting* which directly renders opaque and transparent surfaces from point clouds without connectivity. It is based on a novel screen space formulation of the Elliptical Weighted Average (EWA) filter. Our rigorous mathematical analysis extends the texture resampling framework of Heckbert to irregularly spaced point samples. To render the points, we develop a surface splat primitive that implements the screen space EWA filter. Moreover, we show how to optimally sample image and procedural textures to irregular point data during pre-processing. We also compare the optimal algorithm with a more efficient view-independent EWA pre-filter. Surface splatting makes the benefits of EWA texture filtering available to point-based rendering. It provides high quality anisotropic texture filtering, hidden surface removal, edge anti-aliasing, and order-independent transparency.

Keywords: Rendering Systems, Texture Mapping, Antialiasing, Image-Based Rendering, Frame Buffer Algorithms.

1 Introduction

Laser range and image-based scanning techniques have produced some of the most complex and visually stunning models to date [9]. One of the challenges with these techniques is the huge volume of point samples they generate. A commonly used approach is generating triangle meshes from the point data and using mesh reduction techniques to render them [7, 2]. However, some scanned meshes are too large to be rendered interactively [9], and some applications cannot tolerate the inherent loss in geometric accuracy and texture fidelity that comes from polygon reduction.

Recent efforts have focused on direct rendering techniques for point samples without connectivity [16, 4, 15]. These techniques use hierarchical data structures and forward warping to store and render the point data efficiently. One important challenge for point rendering techniques is to properly reconstruct continuous surfaces from the irregularly spaced point samples while maintaining the high texture fidelity of the scanned data. In addition, the point rendering should correctly handle hidden surface removal and transparency.

In this paper we propose a new point rendering technique called *surface splatting*, focusing on high quality texture filtering. In contrast to previous point rendering approaches, surface splatting uses a novel screen space formulation of the Elliptical Weighted Average (EWA) filter [3], the best anisotropic texture filtering algorithm for interactive systems. Extending the framework of Heckbert [6], we derive a screen space form of the EWA filter for irregularly spaced point samples without global texture parameterization. This makes surface splatting applicable to high-resolution laser range scans, terrain with high texture detail, or point-sampled geometric objects (see Figure 1). A modified A-buffer [1] provides hidden surface removal, edge anti-aliasing, and order-independent transparency at a modest increase in computation efforts.

The main contribution of this paper is a rigorous mathematical formulation of screen space EWA texture filtering for irregular point data, presented in Section 3. We show how the screen space EWA filter can be efficiently implemented using surface splatting in Section 4. If points are used as rendering primitives for complex geometry, we want to apply regular image textures to point samples during conversion from geometric models. Hence, Section 5 introduces an optimal texture sampling and pre-filtering method for irregular point samples. Sections 6 and 7 present our modified A-buffer method for order-independent transparency and edge anti-aliasing, respectively. Finally, we discuss implementation, timings, and image quality issues in Section 8.

*ETH Zürich, Switzerland. Email: [zwicker,grossm]@inf.ethz.ch
†MERL, Cambridge, MA. Email: [pfister,jeroen]@merl.com

2 Previous Work

Texture mapping increases the visual complexity of objects by mapping functions for color, normals, or other material properties onto the surfaces [5]. If these texture functions are inappropriately band-limited, texture aliasing may occur during projection to raster images. For a general discussion of this problem see [21]. Although we develop our contributions along similar lines to the seminal work of Heckbert [6], our approach is fundamentally different from conventional texture mapping. We present the first systematic analysis for representing and rendering texture functions on irregularly point-sampled surfaces.

The concept of representing objects as a set of points and using these as rendering primitives has been introduced in a pioneering report by Levoy and Whitted [10]. Due to the continuing increase in geometric complexity, their idea has recently gained more interest. QSplat [16] is a point rendering system that was designed to interactively render large datasets produced by modern scanning devices. Other researchers demonstrated the efficiency of point-based methods for rendering geometrically complex objects [4, 15]. In some systems, point-based representations are temporarily stored in the rendering pipeline to accelerate rendering [11, 17]. Surprisingly, nobody has systematically addressed the problem of representing texture functions on point-sampled objects and avoiding aliasing during rendering. We present a surface splatting technique that can replace the heuristics used in previous methods and provide superior texture quality.

Volume splatting [19] is closely related to point rendering and surface splatting. A spherical 3D reconstruction kernel centered at each voxel is integrated along one dimension into a 2D "footprint function." As each voxel is projected onto the screen, the 2D footprints are accumulated directly into the image buffer or into image-aligned sheet buffers. Some papers [18, 14] address aliasing caused by insufficient resampling rates during perspective projections. To prevent aliasing, the 3D reconstruction kernels are scaled using a heuristic. In contrast, surface splatting models both reconstructing and band-limiting the texture function in a unified framework. Moreover, instead of pre-integrating isotropic 3D kernels, it uses oriented 2D kernels, providing anisotropic filtering for surface textures.

3 The Surface Splatting Framework

The basis of our surface splatting method is a model for the representation of continuous texture functions on the surface of point-based graphics objects, which is introduced in Section 3.1. Since the 3D points are usually positioned irregularly, we use a weighted sum of radially symmetric basis functions. With this model at hand, we look at the task of rendering point-based objects as a concatenation of warping, filtering, and sampling the continuous texture function. In Section 3.2 we extend Heckbert's resampling theory [6] to process point-based objects and develop a mathematical framework of the rendering procedure. In Section 3.3 we derive an alternative formulation of the EWA texture filter that we call *screen space* EWA, leading to the surface splatting algorithm discussed in Section 4. In Section 5, we describe how to acquire the texture functions, which can be regarded as a scattered data approximation problem. A continuous approximation of the unknown original texture function needs to be computed from an irregular set of samples. We distinguish between scanned objects with color per point and regular textures that are explicitly applied to point-sampled geometry.

3.1 Texture Functions on Point-Based Objects

In conventional polygonal rendering, texture coordinates are usually stored per vertex. This enables the graphics engine to combine the mappings from 2D texture space to 3D object space and from there to 2D screen space into a compound 2D to 2D mapping between texture and screen space. Using this mapping, pixel colors are computed by looking up and filtering texture samples in 2D texture space at rendering time. There is no need for a sampled representation of the texture in 3D object space. By contrast, the compound mapping function is not available with point-based objects at rendering time. Consequently, we must store an explicit texture representation in object space.

We represent point-based objects as a set of irregularly spaced points $\{\mathbf{P}_k\}$ in three dimensional object space without connectivity. A point $\mathbf{P}_k$ has a position and a normal. It is associated with a radially symmetric basis function r_k and coefficients w_k^r, w_k^g, w_k^b that represent continuous functions for red, green, and blue color components. Without loss of generality, we perform all further calculations with scalar coefficients w_k. Note that the basis functions r_k and coefficients w_k are determined in a pre-processing step, described in Section 5.

We define a continuous function on the surface represented by the set of points as illustrated in Figure 2. Given a point $\mathbf{Q}$ any-

Figure 2: *Defining a texture function on the surface of a point-based object.*

where on the surface, shown left, we construct a local parameterization of the surface in a small neighborhood of $\mathbf{Q}$, illustrated on the right. The points $\mathbf{Q}$ and $\mathbf{P}_k$ have *local coordinates* $\mathbf{u}$ and $\mathbf{u}_k$, respectively. We define the continuous surface function $f_c(\mathbf{u})$ as the weighted sum:

$$f_c(\mathbf{u}) = \sum_{k \in \mathbb{N}} w_k r_k(\mathbf{u} - \mathbf{u}_k). \tag{1}$$

We choose basis functions r_k that have *local* support or that are appropriately truncated. Then $\mathbf{u}$ lies in the support of a small number of basis functions. Note that in order to evaluate (1), the local parameterization has to be established in the union of these support areas only, which is very small. Furthermore, we will compute these local parameterizations on the fly during rendering as described in Section 4.

3.2 Rendering

Heckbert introduced a general resampling framework for texture mapping and the EWA texture filter in [6]. His method takes a regularly sampled input function in *source space*, reconstructs a continuous function, warps it to *destination space*, and computes the properly sampled function in destination space. Properly sampled means that the Nyquist criterion is met. We will use the term *screen space* instead of destination space.

We extend this framework towards a more general class of input functions as given by Equation (1) and describe our rendering process as a resampling problem. In contrast to Heckbert's regular setting, in our representation the basis functions r_k are irregularly spaced. In the following derivation, we adopt Heckbert's notation.

Given an input function as in Equation (1) and a mapping $\mathbf{x} = \mathbf{m}(\mathbf{u}) : \mathbb{R}^2 \to \mathbb{R}^2$ from source to screen space, rendering involves the three steps illustrated in Figure 3:

Figure 3: *Warping, filtering, and sampling the texture function.*

1. Warp $f_c(\mathbf{u})$ to screen space, yielding the warped, continuous screen space signal $g_c(\mathbf{x})$:

$$g_c(\mathbf{x}) = (f_c \circ \mathbf{m}^{-1})(\mathbf{x}) = f_c(\mathbf{m}^{-1}(\mathbf{x})),$$

where $\circ$ denotes function concatenation.

2. Band-limit the screen space signal using a prefilter h, resulting in the continuous output function $g'_c(\mathbf{x})$:

$$g'_c(\mathbf{x}) = g_c(\mathbf{x}) \otimes h(\mathbf{x}) = \int_{\mathbb{R}^2} g_c(\xi) h(\mathbf{x} - \xi) d\xi,$$

where $\otimes$ denotes convolution.

3. Sample the continuous output function by multiplying it with an impulse train i to produce the discrete output $g(\mathbf{x})$:

$$g(\mathbf{x}) = g'_c(\mathbf{x}) i(\mathbf{x}).$$

An explicit expression for the warped continuous output function can be derived by expanding the above relations in reverse order:

$$\begin{aligned} g'_c(\mathbf{x}) &= \int_{\mathbb{R}^2} h(\mathbf{x} - \xi) \sum_{k \in \mathbb{N}} w_k r_k(\mathbf{m}^{-1}(\xi) - \mathbf{u}_k) d\xi \\ &= \sum_{k \in \mathbb{N}} w_k \rho_k(\mathbf{x}), \end{aligned} \tag{2}$$

$$\text{where} \quad \rho_k(\mathbf{x}) = \int_{\mathbb{R}^2} h(\mathbf{x} - \xi) r_k(\mathbf{m}^{-1}(\xi) - \mathbf{u}_k) d\xi. \tag{3}$$

We call a warped and filtered basis function $\rho_k(\mathbf{x})$ a *resampling kernel*, which is expressed here as a screen space integral. Equation (2) states that we can first warp and filter each basis function r_k individually to construct the resampling kernels ρ_k and then sum up the contributions of these kernels in screen space. We call this approach *surface splatting*, as illustrated in Figure 4. In contrast to Heckbert, who transformed the screen space integral of Equation (2) to a source space integral and formulated a *source space resampling kernel*, we proceed with (3) to derive a *screen space resampling kernel*.

In order to simplify the integral for $\rho_k(\mathbf{x})$ in (3), we replace a general mapping $\mathbf{m}(\mathbf{u})$ by its local affine approximation $\mathbf{m}_{\mathbf{u}_k}$ at a point $\mathbf{u}_k$,

$$\mathbf{m}_{\mathbf{u_k}}(\mathbf{u}) = \mathbf{x_k} + \mathbf{J}_{\mathbf{u_k}} \cdot (\mathbf{u} - \mathbf{u_k}), \tag{4}$$

$$\text{where} \quad \mathbf{x_k} = \mathbf{m}(\mathbf{u_k}) \quad \text{and the Jacobian} \quad \mathbf{J}_{\mathbf{u_k}} = \tfrac{\partial \mathbf{m}}{\partial \mathbf{u}}(\mathbf{u_k}).$$

Figure 4: *Rendering by* surface splatting, *resampling kernels are accumulated in screen space.*

Heckbert relied on the same approximation in his derivation [6]. Since the basis functions r_k have local support, $\mathbf{m}_{\mathbf{u}_k}$ is used only in a small neighborhood around $\mathbf{u}_k$ in (3). Moreover, the approximation is most accurate in the neighborhood of $\mathbf{u}_k$ and so it does not cause visual artifacts. We use it to rearrange Equation (3), and after a few steps we find:

$$\begin{aligned} \rho_k(\mathbf{x}) &= \int_{\mathbb{R}^2} h(\mathbf{x} - \mathbf{m}_{\mathbf{u}_k}(\mathbf{u}_k) - \xi) r'_k(\xi) d\xi \\ &= (r'_k \otimes h)(\mathbf{x} - \mathbf{m}_{\mathbf{u}_k}(\mathbf{u}_k)), \end{aligned} \tag{5}$$

where $r'_k(\mathbf{x}) = r_k(\mathbf{J}_{\mathbf{u}_k}^{-1}\mathbf{x})$ denotes a warped basis function. Thus, although the texture function is defined on an irregular grid, Equation (5) states that the resampling kernel in screen space, $\rho_k(\mathbf{x})$, can be written as a *convolution* of a warped basis function r'_k and the low-pass filter kernel h. This is essential for the derivation of screen space EWA in the next section. Note that from now on we are omitting the subscript $\mathbf{u}_k$ for $\mathbf{m}$ and $\mathbf{J}$.

3.3 Screen Space EWA

Like Greene and Heckbert [3], we choose elliptical Gaussians both for the basis functions and the low-pass filter, since they are closed under affine mappings and convolution. In the following derivation we apply these mathematical properties to the results of the previous section. This enables us to express the resampling kernel as a single Gaussian, facilitating efficient evaluation during rendering.

An elliptical Gaussian $\mathcal{G}_{\mathbf{V}}(\mathbf{x})$ with variance matrix $\mathbf{V}$ is defined as:

$$\mathcal{G}_{\mathbf{V}}(\mathbf{x}) = \frac{1}{2\pi|\mathbf{V}|^{\frac{1}{2}}} e^{-\frac{1}{2}\mathbf{x}^T \mathbf{V}^{-1}\mathbf{x}},$$

where $|\mathbf{V}|$ is the determinant of $\mathbf{V}$.We denote the variance matrices of the basis functions r_k and the low-pass filter h with $\mathbf{V}_k^r$ and $\mathbf{V}^h$, respectively. The warped basis function and the low-pass filter are:

$$\begin{aligned} r'_k(\mathbf{x}) &= r(\mathbf{J}^{-1}\mathbf{x}) = \mathcal{G}_{\mathbf{V}_k^r}(\mathbf{J}^{-1}\mathbf{x}) = \frac{1}{|\mathbf{J}^{-1}|}\mathcal{G}_{\mathbf{J}\mathbf{V}_k^r\mathbf{J}^T}(\mathbf{x}) \\ h(\mathbf{x}) &= \mathcal{G}_{\mathbf{V}^h}(\mathbf{x}). \end{aligned}$$

The resampling kernel ρ_k of (5) can be written as a single Gaussian with a variance matrix that combines the warped basis function and the low-pass filter. Typically $\mathbf{V}^h = \mathbf{I}$, yielding:

$$\begin{aligned} \rho_k(\mathbf{x}) &= (r'_k \otimes h)(\mathbf{x} - \mathbf{m}(\mathbf{u}_k)) \\ &= \frac{1}{|\mathbf{J}^{-1}|}(\mathcal{G}_{\mathbf{J}\mathbf{V}_k^r\mathbf{J}^T} \otimes \mathcal{G}_{\mathbf{I}})(\mathbf{x} - \mathbf{m}(\mathbf{u}_k)) \\ &= \frac{1}{|\mathbf{J}^{-1}|}\mathcal{G}_{\mathbf{J}\mathbf{V}_k^r\mathbf{J}^T + \mathbf{I}}(\mathbf{x} - \mathbf{m}(\mathbf{u}_k)). \end{aligned} \tag{6}$$

We will show how to determine $\mathbf{J}^{-1}$ in Section 4, and how to compute $\mathbf{V}_k^r$ in Section 5. Substituting the Gaussian resampling kernel

(6) into (2), the continuous output function is the weighted sum:

$$g_c'(\mathbf{x}) = \sum_{k \in \mathbb{N}} w_k \frac{1}{|\mathbf{J}^{-1}|} \mathcal{G}_{\mathbf{J}\mathbf{V}_k^r \mathbf{J}^T + \mathbf{I}}(\mathbf{x} - \mathbf{m}(\mathbf{u}_k)). \qquad (7)$$

We call this novel formulation *screen space* EWA. Note that Equation (7) can easily be converted to Heckbert's original formulation of the EWA filter by transforming it to source space. Remember that $\mathbf{m}$ denotes the local affine approximation (4), hence:

$$\mathbf{x} - \mathbf{m}(\mathbf{u}_k) = \mathbf{m}(\mathbf{m}^{-1}(\mathbf{x}) - \mathbf{u}_k) = \mathbf{J} \cdot (\mathbf{m}^{-1}(\mathbf{x}) - \mathbf{u}_k).$$

Substituting this into (7) we get:

$$g_c'(\mathbf{x}) = \sum_{k \in \mathbb{N}} w_k \mathcal{G}_{\mathbf{V}_k^r + \mathbf{J}^{-1}\mathbf{J}^{-1T}}(\mathbf{m}^{-1}(\mathbf{x}) - \mathbf{u}_k). \qquad (8)$$

Equation (8) states the well known *source space* EWA method extended for irregular sample positions, which is mathematically equivalent to our screen space formulation. However, (8) involves backward mapping a point $\mathbf{x}$ from screen to the object surface, which is impractical for interactive rendering. It amounts to ray tracing the point cloud to find surface intersections. Additionally, the locations $\mathbf{u}_k$ are irregularly positioned such that the evaluation of the resampling kernel in object space is laborious. On the other hand, Equation (7) can be implemented efficiently for point-based objects as described in the next section.

4 The Surface Splatting Algorithm

Intuitively, screen space EWA filtering (7) starts with projecting a radially symmetric Gaussian basis function from the object surface onto the image plane, resulting in an ellipse. The ellipse is then convolved with a Gaussian low-pass filter yielding the resampling filter, whose contributions are accumulated in screen space. Algorithmically, surface splatting proceeds as follows:

```
for each point P[k] {
  project P[k] to screen space;
  determine the resampling kernel rho[k];
  splat rho[k];
}
for each pixel x in the frame buffer {
  shade x;
}
```

We describe these operations in detail:

Determining the resampling kernel The resampling kernel $\rho_k(\mathbf{x})$ in Equation (6) is determined by the Jacobian $\mathbf{J}$ of the 2D to 2D mapping that transforms coordinates of the local surface parameterization to viewport coordinates. This mapping consists of a concatenation of an affine viewing transformation that maps the object to camera space, a perspective projection to screen space, and the viewport mapping to viewport coordinates.

Note that in the viewing transformation we do not allow non-uniform scaling or shearing. This means we preserve the rotation invariance of our basis functions in camera space. Hence, the Jacobian of this transformation can be written as a uniform scaling matrix with scaling factor s_{mv}. Similarly, since we restrict the viewport mapping to translations and uniform scaling, we can describe its Jacobian with a scaling factor s_{vp}.

To compute the Jacobian $\mathbf{J}_{pr}$ of the perspective projection, we have to compute the local surface parameterization. After the viewing transformation, objects are given in camera coordinates that can be projected simply by division by the z coordinate. The center of projection is at the origin of camera space and the projection plane is the plane $z = 1$. The following explanations are illustrated in Figure 5. We construct a local parameterization of the object surface around a point $\mathbf{P}_k$ by approximating the surface with its tangent plane given by the normal $\mathbf{n}_k$ (transformed to camera space) at $\mathbf{P}_k$. We define the parameterization by choosing two orthogonal basis vectors $\mathbf{u}_0$ and $\mathbf{u}_1$ in the tangent plane. Since our basis functions are radially symmetric, the orientation of these vectors is arbitrary. Note that the tangent plane approximation leads to the same inconsistencies of the local parameterizations as in conventional rendering pipelines. There, the perspective mapping from texture space to screen space is determined per triangle. However, when the EWA kernel of a pixel near a triangle edge is warped to texture space, it may overlap a region of the texture that is mapped to several triangles, leading to slightly incorrect filtering of the texture. Yet, for both rendering methods, the error introduced is too small to cause visual artifacts.

Figure 5: *Calculating the Jacobian* $\mathbf{J}_{pr}^{-1}$.

The mapping of coordinates of the local parameterization to screen coordinates is given by the perspective projection of the tangent plane to screen space. We find the Jacobian $\mathbf{J}_{pr}^{-1}$ of the inverse mapping at the point $\mathbf{P}_k$ by projecting the basis vectors of screen space $\mathbf{x}_0$ and $\mathbf{x}_1$ along the viewing ray that connects the center of projection with $\mathbf{P}_k$ onto the tangent plane. This results in the vectors $\tilde{\mathbf{x}}_0$ and $\tilde{\mathbf{x}}_1$. Specifically, we choose $\mathbf{u}_0 = \tilde{\mathbf{x}}_0 / \|\tilde{\mathbf{x}}_0\|$ and construct $\mathbf{u}_1$ such that $\mathbf{u}_0, \mathbf{u}_1$, and the normal $\mathbf{n}_k$ form a right-handed orthonormal coordinate system. This simplifies the Jacobian, since $\tilde{\mathbf{x}}_0 \cdot \mathbf{u}_1 = 0$ and $\tilde{\mathbf{x}}_0 \cdot \mathbf{u}_0 = \|\tilde{\mathbf{x}}_0\|$, which is then given by:

$$\mathbf{J}_{pr}^{-1} = \begin{pmatrix} \tilde{\mathbf{x}}_0 \cdot \mathbf{u}_0 & \tilde{\mathbf{x}}_1 \cdot \mathbf{u}_0 \\ \tilde{\mathbf{x}}_0 \cdot \mathbf{u}_1 & \tilde{\mathbf{x}}_1 \cdot \mathbf{u}_1 \end{pmatrix} = \begin{pmatrix} \|\tilde{\mathbf{x}}_0\| & \tilde{\mathbf{x}}_1 \cdot \mathbf{u}_0 \\ 0 & \tilde{\mathbf{x}}_1 \cdot \mathbf{u}_1 \end{pmatrix},$$

where $\cdot$ denotes the vector dot product.

Concatenating the Jacobians of viewing transformation, projection, and viewport mapping, we finally get $\mathbf{J}$:

$$\mathbf{J} = s_{vp} \cdot \mathbf{J}_{pr} \cdot s_{mv}.$$

Splatting the resampling kernel First, each point $\mathbf{P}_k$ is mapped to the position $\mathbf{m}(\mathbf{u}_k)$ on screen. Then the resampling kernel is centered at $\mathbf{m}(\mathbf{u}_k)$ and is evaluated for each pixel. In other words, the contributions of all points are splatted into an *accumulation buffer*. The projected normals of the points are filtered in the same way. Besides color and normal components, each frame buffer pixel contains the sum of the accumulated contributions of the resampling kernels and camera space z values as well (see Table 1).

Although the Gaussian resampling kernel has infinite support in theory, in practice it is computed only for a limited range of the exponent $\beta(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T(\mathbf{I} + \mathbf{J}^{-1}\mathbf{J}^{-1^T})\mathbf{x}$. We choose a cutoff radius c, such that $\beta(\mathbf{x}) < c$. Bigger values for c increase image quality but also the cost of splatting the kernel. A typical choice is $c = 1$, providing good image quality at moderate splatting cost [6, 13].

Data	Storage
RGBA color components	4×4 Bytes
XYZ normal components	3×4 Bytes
Accumulated contributions	4 Bytes
Camera space z value	4 Bytes
Material index	2 Bytes
Total per pixel:	38 Bytes

Table 1: *Data storage per frame buffer pixel.*

Because the resampling kernels are truncated to a finite support, an additional normalization by the sum of the accumulated contributions is required, yielding the final pixel value:

$$g(\mathbf{x}) = \sum_{k \in \mathbb{N}} w_k \frac{\rho_k(\mathbf{x})}{\sum_{j \in \mathbb{N}} \rho_j(\mathbf{x})}. \tag{9}$$

Since the pixel grid in screen space is regular, the kernel can be evaluated efficiently by forward differencing in a rectangular bounding box and using lookup tables.

In general, the depth complexity of a scene is greater than one, thus a mechanism is required that separates the contributions of different surfaces when they are splatted into the frame buffer. Consequently, the z value of the tangent plane at $\mathbf{P}_k$ is computed at each pixel that is covered by the kernel, which can be done by forward differencing as well. This is similar to the visibility splatting approach of [15]. To determine whether a new contribution belongs to the same surface as is already stored in a pixel, the difference between the new z value and the z value stored in the frame buffer is compared to a threshold. If the difference is smaller than the threshold, the contribution is added to the pixel. Otherwise, given that it is closer to the eye-point, the data of the frame buffer is replaced by the new contribution.

Deferred shading The frame buffer is shaded after all points of a scene have been splatted. This avoids shading invisible points. Instead, each pixel is shaded using the filtered normal. Parameters for the shader are accessed via an index to a table with material properties (see Table 1). Advanced pixel shading methods, such as reflection mapping, can be easily implemented as well.

5 Texture Acquisition

In this section we address the problem of *pre-computing* the texture coefficients w_k and the basis functions r_k of the continuous texture function in (1).

Determining the basis functions As in Section 3.2, the basis functions r_k are Gaussians with variance matrices $\mathbf{V}_k^r$. For each point $\mathbf{P}_k$, this matrix has to be chosen appropriately in order to match the local density of points around $\mathbf{P}_k$. In some applications, we can assume that the sampling pattern in the local planar area around $\mathbf{u}_k$ is a jittered grid with sidelength h in object space. Then a simple solution to choose $\mathbf{V}_k^r$ is:

$$\mathbf{V}_k^r = \begin{pmatrix} \frac{1}{h^2} & 0 \\ 0 & \frac{1}{h^2} \end{pmatrix},$$

which scales the Gaussian by h. For example in the Surfel system [15], h is globally given by the object acquisition process that samples the positions $\mathbf{u}_k$. Another possibility is to choose h as the maximum distance between points in a small neighborhood. This value can be pre-computed and stored in a hierarchical data structure as in [16].

Computing the coefficients We distinguish between two different settings when computing the coefficients w_k:

1. *Objects with per point color.* The object acquisition method provides points with per point color samples.

2. *Texture mapping point-based objects.* Image or procedural textures from external sources are applied to a given point-sampled geometry.

Objects with per point color Many of today's imaging systems, such as laser range scanners or passive vision systems [12], acquire range and color information. In such cases, the acquisition process provides a color sample c_k with each point. We have to compute a continuous approximation $f_c(\mathbf{u})$ of the unknown original texture function from the irregular set of samples c_k.

A computationally reasonable approximation is to normalize the basis functions r_k to form a partition of unity, i.e., to sum up to one everywhere. Then we use the samples as coefficients, hence $w_k = c_k$, and build a weighted sum of the samples c_k:

$$f_c(\mathbf{u}) = \sum_{k \in \mathbb{N}} c_k \hat{r}_k(\mathbf{u} - \mathbf{u}_k) = \sum_{k \in \mathbb{N}} c_k \frac{r_k(\mathbf{u} - \mathbf{u}_k)}{\sum_{j \in \mathbb{N}} r_j(\mathbf{u} - \mathbf{u}_j)},$$

where $\hat{r}_k$ are the normalized basis functions. However, these are rational functions, invalidating the derivation of the resampling kernel in Section 3.2. Instead, we normalize the resampling kernels, which are warped and band-limited basis functions. This normalization does not require additional computations, since it is performed during rendering, as described in Equation (9).

Texture mapping of point-based objects When an image or procedural texture is explicitly applied to point-sampled geometry, a mapping function from texture space to object space has to be available at pre-processing time. This allows us to warp the continuous texture function from texture space with coordinates $\mathbf{s}$ to object space with coordinates $\mathbf{u}$. We determine the unknown coefficients w_k of $f_c(\mathbf{u})$ such that $f_c(\mathbf{u})$ optimally approximates the texture.

From the samples c_i and the sampling locations $\mathbf{s}_i$ of the texture, the continuous texture function $c_c(\mathbf{s})$ is reconstructed using the reconstruction kernel $n(\mathbf{s})$, yielding:

$$c_c(\mathbf{s}) = \sum_i c_i n(\mathbf{s} - \mathbf{s}_i) = \sum_i c_i n_i(\mathbf{s}).$$

In our system, the reconstruction kernel is a Gaussian with unit variance, which is a common choice for regular textures. Applying the mapping $\mathbf{u} = \mathbf{t}(\mathbf{s})$ from texture space to object space, the warped texture function $\tilde{f}_c(\mathbf{u})$ is given by:

$$\tilde{f}_c(\mathbf{u}) = c_c(\mathbf{t}^{-1}(\mathbf{u})) = \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u})).$$

With $\tilde{f}_c(\mathbf{u})$ in place, our goal is to determine the coefficients w_k such that the error of the approximation provided by $f_c(\mathbf{u})$ is minimal. Utilizing the L_2 norm, the problem is minimizing the following functional:

$$\begin{aligned} F(\mathbf{w}) &= \|\tilde{f}_c(\mathbf{u}) - f_c(\mathbf{u})\|_{L_2}^2 \\ &= \|\textstyle\sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u})) - \sum_k w_k r_k(\mathbf{u} - \mathbf{u_k})\|_{L_2}^2, \end{aligned} \tag{10}$$

where $\mathbf{w} = (w_j)$ denotes the vector of unknown coefficients. Since $F(\mathbf{w})$ is a quadratic function of the coefficients, it takes its minimum at $\nabla F(\mathbf{w}) = 0$, yielding a set of linear equations. After some algebraic manipulations, detailed in Appendix A, we find the linear

system $\mathbf{R}\mathbf{w} = \mathbf{c}$. The elements of the matrix $\mathbf{R}$ and the vector $\mathbf{c}$ are given by the inner products:

$$(\mathbf{R})_{kj} = < r_k, r_j > \quad \text{and}$$
$$(\mathbf{c})_k = \sum_i c_i \langle r_k, n_i \circ \mathbf{t}^{-1} \rangle. \tag{11}$$

We compare this optimization method with a view-independent EWA approach, similar as proposed in [4, 15]. Our novel technique is a generalization of view-independent EWA, which can be derived from (11) by means of the simplifying assumption that the basis functions r_k are orthonormal. In this case, the inner products are given by $\langle r_k, r_j \rangle = \delta_{kj}$, where $\delta_{kj} = 1$ if $k = j$ and $\delta_{kj} = 0$ otherwise. Consequently, $\mathbf{R}$ is the identity matrix and the coefficients are determined as in EWA filtering by:

$$w_k = \sum_i c_i \langle r_k, n_i \circ \mathbf{t}^{-1} \rangle.$$

In Figure 6, we show a checkerboard texture that was sampled to an irregular set of points. On the left, we applied our optimized texture sampling technique. On the right, we used view-independent EWA. In the first row, the textures are rendered under minification, which does not reveal the difference between the continuous texture functions, since the filter used for rendering is dominated by the band-limiting step. In the second row, however, magnification clearly illustrates that optimized texture sampling produces a much sharper approximation of the original texture. In the third row, we use extreme magnification to visualize the irregular pattern of points, depicted in the middle.

Figure 6: *Left: optimized texture sampling. Right: view-independent EWA. Bottom middle: Irregular grid of points in the area shown on the left and right.*

6 Transparency

The basic algorithm described in Section 4 can be easily extended to handle transparent surfaces as well. Our approach provides order-independent transparency using a single rendering pass and a fixed amount of frame buffer memory.

The general idea is to use a frame buffer that consists of several layers, each containing the data listed in Table 1. A layer stores a *fragment* at each pixel. The purpose of a fragment is to collect the contributions of a single surface to the pixel. After all points have been splatted, the fragments are blended back-to-front to produce the final pixel color.

We adopt the strategy presented in [8], which avoids the disadvantages of both multi-pass (e.g., [20]) and basic A-buffer (e.g., [1]) algorithms. Providing a small fixed number l of fragments per pixel, fragments are merged whenever the number of fragments exceeds the preset limit l. We apply the same rendering procedure as described in Section 4, where the *splatting* is extended as follows:

Splatting the resampling kernel In contrast to the single layered frame buffer of Section 4, the frame buffer now contains several layers, each storing a fragment per pixel. Each contribution that is splatted into a pixel is processed in three steps:

1. *Accumulate-or-Separate Decision.* Using a z threshold as described in Section 4, all fragments of the pixel are checked to see if they contain data of the same surface as the new contribution. If this is the case, the contribution is added to the fragment and we are done. Otherwise, the new contribution is treated as a separate surface and a temporary fragment is initialized with its data.

2. *New Fragment Insertion.* If the number of fragments including the temporary fragment is smaller than the limit l, the temporary fragment is copied into a free slot in the frame buffer and we are done.

3. *Fragment Merging.* If the above is not true, then two fragments have to be merged. Before merging, the fragments have to be shaded.

When fragments are merged, some information is inevitably lost and visual artifacts may occur. These effects are minimized by using an appropriate merging strategy. Unfortunately, the situation is complicated by the fact that a decision has to be taken as the scene is being rendered, without knowledge about subsequent rendering operations. The main criterion for merging fragments is the difference between their z values. This reduces the chance that there are other surfaces, which are going to be rendered later, that lie between the two merged surfaces. In this case, incorrect back-to-front blending may introduce visual artifacts.

Before fragments can be merged, their final color has to be determined by shading them. Shaded fragments are indicated by setting their accumulated weight (see Table 1) to a negative value to guarantee that they are shaded exactly once. The color and alpha values of the front and back fragment to be merged are $\mathbf{c}_f, \alpha_f$ and $\mathbf{c}_b, \alpha_b$, respectively. The color and alpha values $\mathbf{c}_o, \alpha_o$ of the merged fragment are computed using:

$$\mathbf{c}_o = \mathbf{c}_f \alpha_f + \mathbf{c}_b \alpha_b (1 - \alpha_f)$$
$$\alpha_o = \alpha_f + \alpha_b (1 - \alpha_f). \tag{12}$$

Similar to Section 4, fragments are shaded if necessary in a second pass. After shading, the fragments of each pixel are blended back-to-front as described in Equation (12) to produce the final pixel color.

Figure 7 shows a geometric object consisting of semi-transparent, intersecting surfaces, rendered with the extended surface splatting algorithm. In most areas, the surfaces are blended flawlessly back-to-front. The geometry of the surfaces, however, is not reconstructed properly around the intersection lines, as illustrated in the close-up on the right. In these regions, contributions of different surfaces are mixed in the fragments, which can cause visual artifacts. On the other hand, in areas of high surface curvature the local tangent plane approximation poorly matches the actual surface. Hence, it may happen that not all contributions of a surface are collected in a single fragment, leading to similar artifacts. Essentially, both cases arise because of geometry undersampling. We can avoid the problem by increasing the geometry sampling rate or by using a higher order approximation of the surface. However, the latter is impracticable to compute for interactive rendering.

7 Edge Antialiasing

In order to perform edge antialiasing, information about partial coverage of surfaces in fragments is needed. For point-based representations, one way to approximate coverage is to estimate the density

Figure 7: *Geometric object with intersecting, semi-transparent surfaces, rendered with the extended surface splatting algorithm and edge-antialiasing.*

of projected points per pixel area [10]. Coverage is then computed by measuring the actual density of points in a fragment and dividing the measured value by the estimated value.

Rather than explicitly calculating this estimation, we make the simplifying assumption that the Gaussian basis functions are located on a regular grid and have unit variance. In this case, they approximate a partition of unity. In other words, we assume that they sum up to one at any point. Warping and band-limiting this constant function results in a constant function again. Therefore the sum q of the resampling kernels is approximately one at any point, specifically in all fragments $\mathbf{x}$:

$$q = \sum_{k \in \mathbb{N}} \rho_k(\mathbf{x}) \approx 1.$$

If q is smaller than one in a fragment, this indicates that the texture does not completely cover the pixel and q can be used as a coverage coefficient.

For an irregular grid, the approximation of the partition of unity becomes less reliable. Furthermore, the Gaussians are truncated to a finite support. With a cutoff radius $c = 1$ (see Section 4), we found that a threshold $\tau = 0.4$ for indicating full coverage produces good results in general. The coverage q' of a pixel is $q' = q/\tau$. We implement edge antialiasing by multiplying the alpha value of a fragment with its coverage coefficient. The final alpha value α' of the fragment is:

$$\alpha' = \begin{cases} \alpha & \text{if } q' \geq 1 \\ \alpha \cdot q' & \text{if } q' < 1. \end{cases}$$

8 Results

We implemented a point-sample rendering pipeline based on surface splatting in software. Furthermore, we can convert geometric models into point-based objects in a pre-processing step. Our sampler generates a hierarchical data structure similar to [15], facilitating multiresolution and progressive rendering. It applies view-independent EWA texture filtering to sample image textures onto point objects. We implemented the optimized texture sampling technique discussed in Section 5 in Matlab.

Figure 1, left, shows a face that was rendered using a point cloud acquired by a laser range scanner. Figure 1, middle and right, show point-sampled geometric objects. We illustrate high quality texturing on terrain data and semi-transparent surfaces on the complex model of a helicopter.

Table 2 shows the performance of our unoptimized C implementation of surface splatting. The frame rates were measured on a 1.1 GHz AMD Athlon system with 1.5 GByte memory. We rendered to a frame buffer with three layers and a resolution of 256×256 and 512×512 pixels, respectively. A pixel needs $3 \times 38 = 114$ bytes of storage. The entire frame buffer requires 6.375 MB and 25.5 MB of memory, respectively.

Data	**# Points**	**256 × 256**	**512 × 512**
Scanned Head	429075	1.3 fps	0.7 fps
Matterhorn	4782011	0.2 fps	0.1 fps
Helicopter	987552	0.6 fps	0.3 fps

Table 2: *Rendering performance for frame buffer resolutions 256 × 256 and 512 × 512.*

The texture quality of the surface splatting algorithm is equivalent to conventional source space EWA texture quality. Figure 8, top and second row, compare screen space EWA and source space EWA on a high frequency texture with regular sampling pattern. Note that screen space EWA is rendered with edge antialiasing and there was no hierarchical data structure used. Moreover, the third row illustrates splatting with circular Gaussians, similar to the techniques of Levoy [10] and Shade [17]. We use the major axis of the screen space EWA ellipse as the radius for the circular splats, which corresponds to the choices of [10] and [17]. This leads to overly blurred images in areas where the texture is magnified. In case of

Figure 8: *Top row: screen space EWA. Second row: source space EWA. Third row: circular splats. Bottom: elliptical splats.*

minification, the circular splats approximate the screen space EWA ellipses more closely, leading to better filtering. The bottom row shows splatting with elliptical Gaussians that are determined using the normal direction of the surface as discussed in [16]. This

amounts to omitting the band-limiting step of EWA, which causes aliasing artifacts in regions where the texture is minified. In magnified areas, the elliptical splats result in nice anisotropic filtering. In contrast to these methods, screen space EWA provides a continuous transition between minification and magnification and renders high quality textures in both cases.

9 Conclusions

Surface splatting is a new algorithm that makes the benefits of EWA texture filtering accessible to point-based surface representations and rendering techniques. We have provided a through mathematical analysis of the process of constructing a continuous textured image from irregular points. We have also developed an optimized texture sampling technique to sample image or procedural textures onto point-based objects. Surface splatting provides stable textures with no flickering during animations. A modified A-buffer and simple merging strategy provides transparency and edge-antialiasing with a minimum of visual artifacts.

We will apply surface splatting to procedurally generated objects, such as parametric surfaces or fractal terrain. Rendering the generated points in the order of computation should yield high performance. We think it is possible to extend surface splatting to render volumetric objects, such as clouds, fire, and medical CT scans. This will require extending the screen space EWA framework to 3D spherical kernels. By rendering voxels in approximate front-to-back order we could use our modified A-buffer without undue increase of the number of fragments to be merged per pixel. Because of the simplicity of the surface splatting algorithm we are investigating an efficient hardware implementation. Increasing processor performance and real-time hardware will expand the utility of this high quality point-rendering method.

References

[1] L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics*, volume 18 of *SIGGRAPH '84 Proceedings*, pages 103–108. July 1984.

[2] B. Curless and M. Levoy. A Volumetric Method for Building Complex Models from Range Images. In *Computer Graphics*, SIGGRAPH '96 Proceedings, pages 303–312. New Orleans, LA, August 1996.

[3] N. Greene and P. Heckbert. Creating Raster Omnimax Images from Multiple Perspective Views Using the Elliptical Weighted Average Filter. *IEEE Computer Graphics & Applications*, 6(6):21–27, June 1986.

[4] J. P. Grossman and W. Dally. Point Sample Rendering. In *Rendering Techniques '98*, pages 181–192. Springer, Wien, Vienna, Austria, July 1998.

[5] P. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics & Applications*, 6(11):56–67, November 1986.

[6] P. Heckbert. *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Science, June 17 1989.

[7] H. Hoppe, T. DeRose, T. Duchampt, J. McDonald, and W. Stuetzle. Surface Reconstruction from Unorganized Points. In *Computer Graphics*, SIGGRAPH '92 Proceedings, pages 71–78. Chicago, IL, July 1992.

[8] N. Jouppi and C. Chang. $\mathbb{Z}^3$: An Economical Hardware Technique for High-Quality Antialiasing and Transparency. In *Proceedings of the Eurographics/SIGGRAPH Workshop on Graphics Hardware 1999*, pages 85–93. Los Angeles, CA, August 1999.

[9] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, pages 131–144. Los Angeles, CA, July 2000.

[10] M. Levoy and T. Whitted. The Use of Points as Display Primitives. Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, 1985.

[11] T. W. Mark, L. McMillan, and G. Bishop. Post-Rendering 3D Warping. In *1997 Symposium on Interactive 3D Graphics*, pages 7–16. ACM SIGGRAPH, April 1997.

[12] W. Matusik, C. Buehler, R. Raskar, S. Gortler, and L. McMillan. Image-Based Visual Hulls. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, pages 369–374. Los Angeles, CA, July 2000.

[13] J. McCormack, R. Perry, K. Farkas, and N. Jouppi. Feline: Fast Elliptical Lines for Anisotropic Texture Mapping. In *Computer Graphics*, SIGGRAPH '99 Proceedings, pages 243–250. Los Angeles, CA, August 1999.

[14] K. Mueller, T. Moeller, J.E. Swan, R. Crawfis, N. Shareef, and R. Yagel. Splatting Errors and Antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):178–191, April-June 1998.

[15] H. Pfister, M. Zwicker, J. van Baar, and M Gross. Surfels: Surface Elements as Rendering Primitives. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, pages 335–342. Los Angeles, CA, July 2000.

[16] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, pages 343–352. Los Angeles, CA, July 2000.

[17] J. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered Depth Images. In *Computer Graphics*, SIGGRAPH '98 Proceedings, pages 231–242. Orlando, FL, July 1998.

[18] J. E. Swan, K. Mueller, T. Möller, N. Shareef, R. Crawfis, and R. Yagel. An Anti-Aliasing Technique for Splatting. In *Proceedings of the 1997 IEEE Visualization Conference*, pages 197–204. Phoenix, AZ, October 1997.

[19] L. Westover. Footprint Evaluation for Volume Rendering. In *Computer Graphics*, Proceedings of SIGGRAPH 90, pages 367–376. August 1990.

[20] S. Winner, M. Kelley, B. Pease, B. Rivard, and A. Yen. Hardware Accelerated Rendering of Antialiasing Using a Modified A-Buffer Algorithm. pages 307–316, August 1997.

[21] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, California, 1990.

Appendix A: Mathematical Framework

The L_2 Norm of equation (10) can be computed using inner products $\langle f, f\rangle$:

$$\|f\|_{L_2}^2 = \langle f, f\rangle = \int f(x)f(x)dx,$$

and exploiting the linearity of the operator we obtain:

$$\begin{aligned} F(\mathbf{w}) &= \|\sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u})) - \sum_j w_j r_j(\mathbf{u}-\mathbf{u}_j)\|_{L_2}^2 \\ &= \langle \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u})), \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u}))\rangle \\ &+ \langle \sum_j w_j r_j(\mathbf{u}-\mathbf{u}_j), \sum_j w_j r_j(\mathbf{u}-\mathbf{u}_j)\rangle \\ &- 2\langle \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u})), \sum_j w_j r_j(\mathbf{u}-\mathbf{u}_j)\rangle. \end{aligned}$$

We minimize $F(\mathbf{w})$ by computing the roots of the gradient, i.e.:

$$\nabla F(\mathbf{w}) = \begin{pmatrix} \vdots \\ \frac{\partial F}{\partial w_k} \\ \vdots \end{pmatrix} = 0.$$

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

$$\begin{aligned} \frac{\partial F(\mathbf{w})}{\partial w_k} &= \frac{\partial}{\partial w_k} \sum_i \sum_j w_i w_j \langle r_i(\mathbf{u}-\mathbf{u}_i), r_j(\mathbf{u}-\mathbf{u}_j)\rangle \\ &- 2\frac{\partial}{\partial w_k} \langle \sum_j w_j r_j(\mathbf{u}-\mathbf{u}_j), \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u}))\rangle \\ &= 2\sum_j w_j \langle r_j(\mathbf{u}-\mathbf{u}_j), r_k(\mathbf{u}-\mathbf{u}_k)\rangle \\ &- 2\sum_i c_i \langle r_k(\mathbf{u}-\mathbf{u}_k), n_i(\mathbf{t}^{-1}(\mathbf{u}))\rangle = 0. \end{aligned}$$

This set of linear equations can be written in matrix form:

$$\underbrace{\begin{pmatrix} \ddots & & \\ & \langle r_k, r_j\rangle & \\ & & \ddots \end{pmatrix}}_{\mathbf{R}} \underbrace{\begin{pmatrix} \vdots \\ w_j \\ \vdots \end{pmatrix}}_{\mathbf{w}} = \underbrace{\begin{pmatrix} \vdots \\ \sum_i c_i \langle r_k, n_i \cdot \mathbf{t}^{-1}\rangle \\ \vdots \end{pmatrix}}_{\mathbf{c}}.$$

Surfels: Surface Elements as Rendering Primitives

Hanspeter Pfister * Matthias Zwicker † Jeroen van Baar* Markus Gross†

Figure 1: *Surfel rendering examples.*

Abstract

Surface elements (surfels) are a powerful paradigm to efficiently render complex geometric objects at interactive frame rates. Unlike classical surface discretizations, i.e., triangles or quadrilateral meshes, surfels are point primitives without explicit connectivity. Surfel attributes comprise depth, texture color, normal, and others. As a pre-process, an octree-based surfel representation of a geometric object is computed. During sampling, surfel positions and normals are optionally perturbed, and different levels of texture colors are prefiltered and stored per surfel. During rendering, a hierarchical forward warping algorithm projects surfels to a z-buffer. A novel method called visibility splatting determines visible surfels and holes in the z-buffer. Visible surfels are shaded using texture filtering, Phong illumination, and environment mapping using per-surfel normals. Several methods of image reconstruction, including supersampling, offer flexible speed-quality tradeoffs. Due to the simplicity of the operations, the surfel rendering pipeline is amenable for hardware implementation. Surfel objects offer complex shape, low rendering cost and high image quality, which makes them specifically suited for low-cost, real-time graphics, such as games.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation – Viewing Algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques – Graphics Data Structures and Data Types.

Keywords: Rendering Systems, Texture Mapping.

*MERL, Cambridge, MA. Email: [pfister,jeroen]@merl.com
†ETH Zürich, Switzerland. Email: [zwicker,gross]@inf.ethz.ch

1 Introduction

3D computer graphics has finally become ubiquitous at the consumer level. There is a proliferation of affordable 3D graphics hardware accelerators, from high-end PC workstations to low-priced gamestations. Undoubtedly, key to this success is interactive computer games that have emerged as the "killer application" for 3D graphics. However, interactive computer graphics has still not reached the level of realism that allows a true immersion into a virtual world. For example, typical foreground characters in real-time games are extremely minimalistic polygon models that often exhibit faceting artifacts, such as angular silhouettes.

Various sophisticated modeling techniques, such as implicit surfaces, NURBS, or subdivision surfaces, allow the creation of 3D graphics models with increasingly complex shapes. Higher order modeling primitives, however, are eventually decomposed into triangles before being rendered by the graphics subsystem. The triangle as a rendering primitive seems to meet the right balance between descriptive power and computational burden [7]. To render realistic, organic-looking models requires highly complex shapes with ever more triangles, or, as Alvy Ray Smith puts it: "Reality is 80 million polygons" [26]. Processing many small triangles leads to bandwidth bottlenecks and excessive floating point and rasterization requirements [7].

To increase the apparent visual complexity of objects, texture mapping was introduced by Catmull [3] and successfully applied by others [13]. Textures convey more detail inside a polygon, thereby allowing larger and fewer triangles to be used. Today's graphics engines are highly tailored for high texture mapping performance. However, texture maps have to follow the underlying geometry of the polygon model and work best on flat or slightly curved surfaces. Realistic surfaces frequently require a large number of textures that have to be applied in multiple passes during rasterization. And phenomena such as smoke, fire, or water are difficult to render using textured triangles.

In this paper we propose a new method of rendering objects with rich shapes and textures at interactive frame rates. Our rendering architecture is based on simple surface elements (*surfels*) as rendering primitives. Surfels are point samples of a graphics model. In a preprocessing step, we sample the surfaces of complex geometric models along three orthographic views. At the same time, we perform computation-intensive calculations such as texture, bump, or displacement mapping. By moving rasterization and texturing from

Figure 2: *Algorithm overview: a) Preprocessing. b) Rendering of the hierarchical LDC tree.*

the core rendering pipeline to the preprocessing step, we dramatically reduce the rendering cost.

From a modeling point of view, the surfel representation provides a mere discretization of the geometry and hence reduces the object representation to the essentials needed for rendering. By contrast, triangle primitives implicitly store connectivity information, such as vertex valence or adjacency – data not necessarily available or needed for rendering. In a sense, a surfel relates to what Levoy and Whitted call the *lingua franca* of rendering in their pioneering report from 1985 [18].

Storing normals, prefiltered textures, and other per surfel data enables us to build high quality rendering algorithms. Shading and transformations applied per surfel result in Phong illumination, bump, and displacement mapping, as well as other advanced rendering features. Our data structure provides a multiresolution object representation, and a hierarchical forward warping algorithm allows us to estimate the surfel density in the output image for speed-quality tradeoffs.

The surfel rendering pipeline complements the existing graphics pipeline and does not intend to replace it. It is positioned between conventional geometry-based approaches and image-based rendering and trades memory overhead for rendering performance and quality. The focus of this work has been interactive 3D applications, not high-end applications such as feature films or CAD/CAM. Surfels are not well suited to represent flat surfaces, such as walls or scene backgrounds, where large, textured polygons provide better image quality at lower rendering cost. However, surfels work well for models with rich, organic shapes or high surface details and for applications where preprocessing is not an issue. These qualities make them ideal for interactive games.

2 Related Work

The use of points as rendering primitives has a long history in computer graphics. As far back as 1974, Catmull [3] observed that geometric subdivision may ultimately lead to points. Particles were subsequently used for objects that could not be rendered with geometry, such as clouds, explosions, and fire [23]. More recently, image-based rendering has become popular because its rendering time is proportional to the number of pixels in the source and output images and not the scene complexity.

Visually complex objects have been represented by dynamically generated image sprites [25], which are quick to draw and largely retain the visual characteristics of the object. A similar approach was used in the Talisman rendering system [27] to maintain high and approximately constant frame rates. However, mapping objects onto planar polygons leads to visibility errors and does not allow for parallax and disocclusion effects. To address these problems, several methods add per-pixel depth information to images, variously called layered impostors [24], sprites with depth, or layered depth images [25], just to name a few. Still, none of these techniques provide a complete object model that can be illuminated and rendered from arbitrary points of view.

Some image-based approaches represent objects without explicitly storing any geometry or depth. Methods such as view interpolation and Quicktime VR [5] or plenoptic modeling [21] create new views from a collection of 2D images. Lightfield [17] or lumigraph [9] techniques describe the radiance of a scene or object as a function of position and direction in a four- or higher-dimensional space, but at the price of considerable storage overhead. All these methods use view-dependent samples to represent an object or scene. However, view-dependent samples are ineffective for dynamic scenes with motion of objects, changes in material properties, and changes in position and intensities of light sources.

The main idea of representing objects with surfels is to describe them in a view-independent, object-centered rather than image-centered fashion. As such, surfel rendering is positioned between geometry rendering and image-based rendering. In volume graphics [16], synthetic objects are implicitly represented with surface voxels, typically stored on a regular grid. However, the extra third dimension of volumes comes at the price of higher storage requirements and longer rendering times. In [8], Perlin studies "surflets," a flavor of wavelets that can be used to describe free-form implicit surfaces. Surflets have less storage overhead than volumes, but rendering them requires lengthy ray casting.

Our research was inspired by the following work: Animatek's Caviar player [1] provides interactive frame rates for surface voxel models on a Pentium class PC, but uses simplistic projection and illumination methods. Levoy and Whitted [18] use points to model objects for the special case of continuous, differentiable surfaces. They address the problem of texture filtering in detail. Max uses point samples obtained from orthographic views to model and render trees [20]. Dally et al. [6] introduced the delta tree as an object-centered approach to image-based rendering. The movement of the viewpoint in their method, however, is still confined to particular locations. More recently, Grossman and Dally [12] describe a point sample representation for fast rendering of complex objects. Chang et al. [4] presented the LDI tree, a hierarchical space-partitioning data structure for image-based rendering.

We extend and integrate these ideas and present a complete point sample rendering system comprising an efficient hierarchical representation, high quality texture filtering, accurate visibility calculations, and image reconstruction with flexible speed-quality tradeoffs. Our surfel rendering pipeline provides high quality rendering of exceedingly complex models and is amenable for hardware implementation.

3 Conceptual Overview

Similar to the method proposed by Levoy and Whitted [18], our surfel approach consists of two main steps: sampling and surfel rendering. Sampling of geometry and texture is done during preprocessing, which may include other view-independent methods such as bump and displacement mapping. Figure 2 gives a conceptual overview of the algorithm.

The sampling process (Section 5) converts geometric objects and their textures to surfels. We use ray casting to create three orthogonal layered depth images (LDIs) [25]. The LDIs store multiple surfels along each ray, one for each ray-surface intersection point. Lischinski and Rappaport [19] call this arrangement of three orthogonal LDIs a *layered depth cube (LDC)*. An important and novel aspect of our sampling method is the distinction between sampling of *shape*, or geometry, and *shade*, or texture color. A surfel stores both shape, such as surface position and orientation, and shade, such as multiple levels of prefiltered texture colors. Because of the similarities to traditional texture mipmaps we call this hierarchical color information a *surfel mipmap*.

From the LDC we create an efficient hierarchical data structure for rendering. Chang et al.[4] introduce the LDI tree, an octree with an LDI attached to each octree node. We use the same hierarchical space-partitioning structure, but store an LDC at each node of the octree (Section 6). Each LDC node in the octree is called a *block*. We call the resulting data structure the *LDC tree*. In a step called *3-to-1 reduction* we optionally reduce the LDCs to single LDIs on a block-by-block basis for faster rendering.

The rendering pipeline (Section 7) hierarchically projects blocks to screen space using perspective projection. The rendering is accelerated by block culling [12] and fast incremental forward warping. We estimate the projected surfel density in the output image to control rendering speed and quality of the image reconstruction. A conventional z-buffer together with a novel method called *visibility splatting* solves the visibility problem. Texture colors of visible surfels are filtered using linear interpolation between appropriate levels of the surfel mipmap. Each visible surfel is shaded using, for example, Phong illumination and reflection mapping. The final stage performs image reconstruction from visible surfels, including hole filling and antialiasing. In general, the resolution of the output image and the resolution of the z-buffer do not have to be the same.

4 Definition of a Surfel

We found the term surfel as an abbreviation for *surface element* or *surface voxel* in the volume rendering and discrete topology literature. Herman [15] defines a surfel as an oriented $(n-1)$-dimensional object in R^n. For $n = 3$, this corresponds to an oriented unit square (voxel face) and is consistent with thinking of voxels as little cubes. However, for our discussion we find it more useful to define surfels as follows:

A surfel is a zero-dimensional n-tuple with shape and shade attributes that locally approximate an object surface.

We consider the alternative term, point sample, to be too general, since voxels and pixels are point samples as well.

5 Sampling

The goal during sampling is to find an optimal surfel representation of the geometry with minimum redundancy. Most sampling methods perform object discretization as a function of geometric parameters of the surface, such as curvature or silhouettes. This *object space* discretization typically leads to too many or too few primitives for rendering. In a surfel representation, object sampling is aligned to *image space* and matches the expected output resolution of the image.

5.1 LDC Sampling

We sample geometric models from three sides of a cube into three orthogonal LDIs, called a *layered depth cube (LDC)* [19] or *block*. Figure 3 shows an LDC and two LDIs using a 2D drawing. Ray casting records all intersections, including intersections with back-facing surfaces. At each intersection point, a surfel is created with floating point depth and other shape and shade properties. Perturbation of the surface normal or of the geometry for bump and displacement mapping can be performed on the geometry before sampling or during ray casting using procedural shaders.

Figure 3: *Layered depth cube sampling (shown in 2D).*

Alternatively, we could sample an object from predetermined directions on a surrounding convex hull using orthographic depth images [6, 12]. However, combining multiple reference images and eliminating the redundant information is a difficult problem [21], and sampling geometry with reference images works best for smooth and convex objects. In addition, LDC sampling allows us to easily build a hierarchical data structure, which would be difficult to do from dozens of depth images.

5.2 Adequate Sampling Resolution

Given a pixel spacing of h_0 for the full resolution LDC used for sampling, we can determine the resulting sampling density on the surface. Suppose we construct a Delaunay triangulation on the object surface using the generated surfels as triangle vertices. As was observed in [19], the imaginary triangle mesh generated by this sampling process has a maximum sidelength s_{max} of $\sqrt{3}h_0$. The minimum sidelength s_{min} is 0 when two or three sampling rays intersect at the same surface position.

Similarly to [12], we call the object *adequately sampled* if we can guarantee that at least one surfel is projected into the support of each ouptut pixel filter for orthographic projection and unit magnification. That condition is met if s_{max}, the maximum distance between adjacent surfels in object space, is less than the radius r'_{rec} of the desired pixel reconstruction filter. Typically, we choose the LDI resolution to be slightly higher than this because of the effects of magnification and perspective projection. We will revisit these observations when estimating the number of projected surfels per pixel in Section 7.2.

5.3 Texture Prefiltering

A feature of surfel rendering is that textures are prefiltered and mapped to object space during preprocessing. We use *view-independent* texture filtering as in [12]. To prevent view-dependent texture aliasing we also apply per-surfel texture filtering during rendering (see Sections 7.4 and 7.6).

To determine the extent of the filter footprint in texture space, we center a circle at each surfel on its tangent plane, as shown in Figure 4a. We call these circles *tangent disks*. The tangent disks are

Figure 4: *Texture prefiltering with tangent disks.*

mapped to ellipses in texture space (see Figure 4b) using the predefined texture parameterization of the surface. An EWA filter [14] is applied to filter the texture and the resulting color is assigned to the surfel. To enable adequate texture reconstruction, the elliptical filter footprints in texture space must overlap each other. Consequently, we choose $r^0_{pre} = s_{max}$, the maximum distance between adjacent surfels in object space, as the radius for the tangent disks. This usually guarantees that the tangent disks intersect each other in object space and that their projections in texture space overlap.

Grossman and Dally [12] also use view-independent texture filtering and store one texture sample per surfel. Since we use a modified z-buffer algorithm to resolve visibility (Section 7.3), not all surfels may be available for image reconstruction, which leads to texture aliasing artifacts. Consequently, we store several (typically three or four) prefiltered texture samples per surfel. Tangent disks with dyadically larger radii $r^k_{pre} = s_{max}2^k$ are mapped to texture space and used to compute the prefiltered colors. Because of its similarity to mipmapping [13], we call this a *surfel mipmap*. Figure 4b shows the elliptical footprints in texture space of consecutively larger tangent disks.

6 Data Structure

We use the LDC tree, an efficient hierarchical data structure, to store the LDCs acquired during sampling. It allows us to quickly estimate the number of projected surfels per pixel and to trade rendering speed for higher image quality.

6.1 The LDC Tree

Chang et al. [4] use several reference depth images of a scene to construct the LDI tree. The depth image pixels are resampled onto multiple LDI tree nodes using splatting [29]. We avoid these interpolation steps by storing LDCs at each node in the octree that are subsampled versions of the highest resolution LDC.

The octree is recursively constructed bottom up, and its height is selected by the user. The highest resolution LDC — acquired during geometry sampling — is stored at the lowest level $n = 0$. If the highest resolution LDC has a pixel spacing of h_0, then the LDC at level n has a pixel spacing of $h_n = h_0 2^n$. The LDC is subdivided into blocks with user-specified dimension b, i.e., the LDIs in a block have b^2 layered depth pixels. b is the same for all levels of the tree. Figure 5a shows two levels of an LDC tree with $b = 4$ using a 2D drawing. In the figure, neighboring blocks are differently shaded,

Figure 5: *Two levels of the LDC tree (shown in 2D).*

and empty blocks are white. Blocks on higher levels of the octree are constructed by subsampling their children by a factor of two. Figure 5b shows level $n = 1$ of the LDC tree. Note that surfels at higher levels of the octree reference surfels in the LDC of level 0, i.e., surfels that appear in several blocks of the hierarchy are stored only once and shared between blocks.

Empty blocks (shown as white squares in the figure) are not stored. Consequently, the block dimension b is not related to the dimension of the highest resolution LDC and can be selected arbitrarily. Choosing $b = 1$ makes the LDC tree a fully volumetric octree representation. For a comparison between LDCs and volumes see [19].

6.2 3-to-1 Reduction

To reduce storage and rendering time it is often useful to optionally reduce the LDCs to one LDI on a block-by-block basis. Because this typically corresponds to a three-fold increase in warping speed, we call this step *3-to-1 reduction*. First, surfels are resampled to integer grid locations of ray intersections as shown in Figure 6. Currently we use nearest neighbor interpolation, although a more

Figure 6: *3-to-1 reduction example.*

sophisticated filter, e.g., splatting as in [4], could easily be implemented. The resampled surfels of the block are then stored in a single LDI.

The reduction and resampling process degrades the quality of the surfel representation, both for shape and for shade. Resampled surfels from the same surface may have very different texture colors and normals. This may cause color and shading artifacts that are worsened during object motion. In practice, however, we did not encounter severe artifacts due to 3-to-1 reduction. Because our rendering pipeline handles LDCs and LDIs the same way, we could store blocks with thin structures as LDCs, while all other blocks could be reduced to single LDIs.

As in Section 5.2, we can determine bounds on the surfel density on the surface after 3-to-1 reduction. Given a sampling LDI with pixel spacing h_0, the maximum distance between adjacent surfels on the object surface is $s_{max} = \sqrt{3}h_0$, as in the original LDC tree. The minimum distance between surfels increases to $s_{min} = h_0$ due to the elimination of redundant surfels, making the imaginary Delaunay triangulation on the surface more uniform.

7 The Rendering Pipeline

The rendering pipeline takes the surfel LDC tree and renders it using hierarchical visibility culling and forward warping of blocks. Hierarchical rendering also allows us to estimate the number of projected surfels per output pixel. For maximum rendering efficiency, we project approximately one surfel per pixel and use the same resolution for the z-buffer as in the output image. For maximum image quality, we project multiple surfels per pixel, use a finer resolution of the z-buffer, and high quality image reconstruction.

7.1 Block Culling

We traverse the LDC tree from top (the lowest resolution blocks) to bottom (the highest resolution blocks). For each block, we first perform view-frustum culling using the block bounding box. Next, we use *visibility cones*, as described in [11], to perform the equivalent of backface culling of blocks. Using the surfel normals, we precompute a visibility cone per block, which gives a fast, conservative visibility test: no surfel in the block is visible from any viewpoint within the cone. In contrast to [11], we perform all visibility tests hierarchically in the LDC tree, which makes them more efficient.

7.2 Block Warping

During rendering, the LDC tree is traversed top to bottom [4]. To choose the octree level to be projected, we conservatively estimate for each block the number of surfels per pixel. We can choose one surfel per pixel for fast rendering or multiple surfels per pixel for supersampling. For each block at tree level n, the number of surfels per pixel is determined by i^n_{max}, the maximum distance between adjacent surfels in image space. We estimate i^n_{max} by dividing the maximum length of the projected four major diagonals of the block bounding box by the block dimension b. This is correct for orthographic projection. However, the error introduced by using perspective projection is small because a block typically projects to a small number of pixels.

For each block, i^n_{max} is compared to the radius r'_{rec} of the desired pixel reconstruction filter. r'_{rec} is typically $\frac{\sqrt{2}}{2}s_o$, where s_o

is the sidelength of an output pixel. If i^n_{max} of the current block is larger than r'_{rec} then its children are traversed. We project the block whose i^n_{max} is smaller than r'_{rec}, rendering approximately one surfel per pixel. Note that the number of surfels per pixel can be increased by requiring that i^n_{max} is a fraction of r'_{rec}. The resulting i^n_{max} is stored as i_{max} with each projected surfel for subsequent use in the visibility testing and the image reconstruction stages. The radius of the actual reconstruction filter is $r_{rec} = \max(r'_{rec}, i_{max})$ (see Section 7.6).

To warp a block to screen space we use the optimized incremental block warping by Grossman and Dally, presented in detail in [11]. Its high efficiency is achieved due to the regularity of LDCs. It uses only 6 additions, 3 multiplications, and 1 reciprocal per sample. The LDIs in each LDC block are warped independently, which allows us to render an LDC tree where some or all blocks have been reduced to single LDIs after 3-to-1 reduction.

7.3 Visibility Testing

Perspective projection, high z-buffer resolution, and magnification may lead to undersampling or holes in the z-buffer. A z-buffer pixel is a *hole* if it does not contain a visible surfel or background pixel after projection. Holes have to be marked for image reconstruction. Each pixel of the z-buffer stores a pointer to the closest surfel and the current minimum depth. Surfel depths are projected to the z-buffer using nearest neighbor interpolation.

To correctly resolve visibility in light of holes, we scan-convert the orthographic projection of the surfel tangent disks into the z-buffer. The tangent disks have a radius of $r^n_t = s_{max}2^n$, where s_{max} is the maximum distance between adjacent surfels in object space and n is the level of the block. We call this approach *visibility splatting*, shown in Figure 7. Visibility splatting effectively separates visibility calculations and reconstruction of the image, which produces high quality images and is amenable to hardware implementation [22].

Figure 7: *Visibility splatting.*

After orthographic projection, the tangent disks form an ellipse around the surfel, as shown in Figure 7b. We approximate the ellipse with a partially axis-aligned bounding box, shown in red. The bounding box parallelogram can be easily scan-converted, and each z-buffer pixel is filled with the appropriate depth (indicated by the shaded squares in the figure), depending on the surfel normal N. This scan conversion requires only simple setup calculations, no interpolation of colors, and no perspective divide.

The direction of the minor axis a_{min} of the projected ellipse is parallel to the projection of the surfel normal N. The major axis a_{max} is orthogonal to a_{min}. The length of a_{max} is the projection of the tangent disk radius r^n_t, which is approximated by i_{max}. This approximation takes the orientation and magnification of the LDC tree during projection into account. Next, we calculate the coordinate axis that is most parallel to a_{min} (the y-axis in Figure 7). The short side of the bounding box is axis aligned with this coordinate axis to simplify scan conversion. Its height h is computed by intersecting the ellipse with the coordinate axis. The width w of the bounding box is determined by projecting the vertex at the intersection of the major axis and the ellipse onto the second axis (the x-axis in Figure 7).

$\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ are the partial derivatives of the surfel depth z with respect to the screen x and y direction. They are constant because of the orthographic projection and can be calculated from the unit normal N. During scan conversion, the depth at each pixel inside the bounding box is calculated using $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. In addition, we add a small threshold to each projected z value. The threshold prevents surfels that lie on the foreground surface to be accidentally discarded. Pixels that have a larger z than the z values of the splatted tangent disk are marked as holes.

If the surface is extremely bent, the tangential planes do not cover it completely, potentially leaving tears and holes. In addition, extreme perspective projection makes orthographic projection a bad approximation to the actual projected tangent disk. In practice, however, we did not see this as a major problem. If the projected tangent disk is a circle, i.e., if N is almost parallel to the viewing direction, the bounding box parallelogram is a bad approximation. In this case, we use a square bounding box instead.

Using a somewhat related approach, Grossman and Dally [12] use a hierarchical z-buffer for visibility testing. Each surfel is projected and the hole size around the surfel is estimated. The radius of the hole determines the level of the hierarchical z-buffer where the z-depth of the surfel will be set. This can be regarded as visibility splatting using a hierarchical z-buffer. The advantage is that the visibility splat is performed with a single depth test in the hierarchical z-buffer. However, the visibility splat is always square, essentially representing a tangential disk that is parallel to the image plane. In addition, it is not necessarily centered around the projected surfel. To recover from those drawbacks, [12] introduces weights indicating coverage of surfels. But this makes the reconstruction process more expensive and does not guarantee complete coverage of hidden surfaces.

7.4 Texture Filtering

As explained in Section 5.3, each surfel in the LDC tree stores several prefiltered texture colors of the surfel mipmap. During rendering, the surfel color is linearly interpolated from the surfel mipmap colors depending on the object minification and surface orientation. Figure 8a shows all visible surfels of a sampled surface projected to the z-buffer. The ellipses around the centers of the surfels mark the projection of the footprints of the highest resolution texture prefilter (Section 5.3). Note that during prefiltering, we try to guarantee that the footprints cover the surface completely. In figure 8b the number of samples per z-buffer pixel is limited to one by applying z-buffer depth tests. In order to fill the gaps appearing in the coverage of the surface with texture footprints, the footprints of the remaining surfels have to be enlarged. If surfels are discarded in a given z-buffer pixel, we can assume that the z-buffer pixels in the 3x3 neighborhood around it are not holes. Thus the gaps can be filled if the texture footprint of each surfel covers at least the area of a z-buffer pixel. Consequently, the ellipse of the projected footprint has to have a minor radius of $\sqrt{2}s_z$ in the worst case, where s_z is the z-buffer pixel spacing. But we ignore that worst case and use $\frac{\sqrt{2}}{2}s_z$, implying that surfels are projected to z-buffer pixel centers.

Figure 8: *Projected surfel mipmaps.*

Figure 8b shows the scaled texture footprints as ellipses around projected surfels.

To select the appropriate surfel mipmap level, we use traditional *view-dependent* texture filtering, as shown in Figure 9. A circle with

Figure 9: *Projected pixel coverage.*

radius $\frac{\sqrt{2}}{2}s_z$ is projected through a pixel onto the tangent plane of the surface from the direction of the view, producing an ellipse in the tangent plane. In this calculation, the projection of the circle is approximated with an orthographic projection. Similar to isotropic texture mapping, the major axis of the projected tangent space ellipse is used to determine the surfel mipmap level. The surfel color is computed by linear interpolation between the closest two mipmap levels with prefilter radii r^k_{pre} and r^{k+1}_{pre}, respectively.

7.5 Shading

The illumination model is usually applied before visibility testing. However, deferred shading after visibility testing avoids unnecessary work. Grossman and Dally [12] perform shading calculations in object space to avoid transformation of normals to camera space. However, we already transform the normals to camera space during visibility splatting (Section 7.3). With the transformed normals at hand, we use cube reflectance and environment maps [28] to calculate a per-surfel Phong illumination model. Shading with per-surfel normals results in high quality specular highlights.

7.6 Image Reconstruction and Antialiasing

Reconstructing a continuous surface from projected surfels is fundamentally a scattered data interpolation problem. In contrast to other approaches, such as splatting [29], we separate visibility calculations from image reconstruction [22]. Z-buffer pixels with holes are marked during visibility splatting. These hole pixels are not used during image reconstruction because they do not contain any visible samples. Figure 10 shows the z-buffer after rendering of an object and the image reconstruction process.

Figure 10: *Image reconstruction.*

The simplest and fastest approach, shown in Figure 10a, is to choose the size of an output pixel s_o to be the same as the z-buffer pixel size s_z. Surfels are mapped to pixel centers using nearest neighbor interpolation, shown with color squares in the figure. Holes are marked with a black X. Recall that during forward warping each surfel stores i_{max}, an estimate of the maximum distance between adjacent projected surfels of a block. i_{max} is a good estimate for the minimum radius of a pixel filter that contains at least one surfel. To interpolate the holes, we use a radially symmetric Gauss filter with a radius r_{rec} slightly larger than i_{max} positioned at hole pixel centers. Alternatively, to fill the holes we implemented the pull-push algorithm used by Grossman and Dally [12] and described by Gortler et al.[9].

A high quality alternative is to use supersampling, shown in Figure 10b. The output image pixel size s_o is any multiple of the z-buffer pixel size s_z. Dotted lines in the figure indicate image-buffer subpixels. Rendering for supersampling proceeds exactly the same as before. During image reconstruction we put a Gaussian filter at the centers of all output pixels to filter the subpixel colors. The radius of the filter is $r_{rec} = \max(r'_{rec}, i_{max})$. Thus r_{rec} is at least as large as $r'_{rec} = \frac{\sqrt{2}}{2}s_o$, but it can be increased if i_{max} indicates a low density of surfels in the output image.

It is instructive to see how the color of an output pixel is determined for regular rendering and for supersampling in the absence of holes. For regular rendering, the pixel color is found by nearest neighbor interpolation from the closest visible surfel in the z-buffer. The color of that surfel is computed by linear interpolation between two surfel mipmap levels. Thus the output pixel color is calculated from two prefiltered texture samples. In the case of supersampling, one output pixel contains the filtered colors of one surfel per z-buffer subpixel. Thus, up to eight prefiltered texture samples may contribute to an output pixel for 2×2 supersampling. This produces image quality similar to trilinear mipmapping.

Levoy and Whitted [18] and Chang et al. [4] use an algorithm very similar to Carpenter's A-Buffer [2] with per-pixel bins and compositing of surfel colors. However, to compute the correct per pixel coverage in the A-buffer requires projecting all visible surfels. Max [20] uses an output LDI and an A-buffer for high quality anti-aliasing, but he reports rendering times of 5 minutes per frame. Our method with hierarchical density estimation, visibility splatting, and surfel mipmap texture filtering offers more flexible speed-quality tradeoffs.

8 Implementation and Results

We implemented sampling using the Blue Moon Rendering Tools (BMRT) ray tracer [10]. We use a sampling resolution of 512^2 for the LDC for 480^2 expected output resolution. At each intersection point, a Renderman shader performs view-independent calculations, such as texture filtering, displacement mapping, and bump mapping, and prints the resulting surfels to a file. Pre-processing for a typical object with 6 LOD surfel mipmaps takes about one hour.

A fundamental limitation of LDC sampling is that thin structures that are smaller than the sampling grid cannot be correctly represented and rendered. For example, spokes, thin wires, or hair are hard to capture. The rendering artifacts are more pronounced after 3-to-1 reduction because additional surfels are deleted. However, we had no problems sampling geometry as thin as the legs and wings of the wasp shown in Figure 1 and Figure 12.

The surfel attributes acquired during sampling include a surface normal, specular color, shininess, and three texture mipmap levels. Material properties are stored as an index into a table. Our system does currently not support transparency. Instead of storing a normal we store an index to a quantized normal table for reflection and environment map shading [28]. Table 1 shows the minimum storage requirements per surfel. We currently store RGB colors as 32-bit integers for a total of 20 Bytes per surfel.

Data	Storage
3 texture mipmap levels	3×32 bits
Index into normal table	16 bit
LDI depth value	32 bit
Index into material table	16 bit
Total per sample:	20 Bytes

Table 1: *Typical storage requirements per surfel.*

Table 2 lists the surfel objects that we used for performance analysis with their geometric model size, number of surfels, and file size

Figure 11: *Tilted checker plane. Reconstruction filter: a) Nearest neighbor. b) Gaussian filter. c) Supersampling.*

before and after 3-to-1 reduction. All models use three LODs and three surfel mipmap levels. The size of the LDC tree is about a factor of 1.3 larger than the LDC acquired during sampling. This

Data	# Polys	3 LDIs	3-to-1 Reduced
Salamander	81 k	112 k / 5 MB	70 k / 3 MB
Wasp	128 k	369 k / 15 MB	204 k / 8 MB
Cab	155 k	744 k / 28 MB	539 k / 20 MB

Table 2: *Geometric model sizes and storage requirements (# surfels / file size) for full and 3-to-1 reduced LDC trees.*

overhead is due to the octree data structure, mainly because of the pointers from the lower resolution blocks to surfels of the sampled LDC. We currently do not optimize or compress the LDC tree.

Figure 1 shows different renderings of surfel objects, including environment mapping and displacement mapping. Figure 12 shows an example of hole detection and image reconstruction. Visibility splatting performs remarkably well in detecting holes. However, holes start to appear in the output image for extreme closeups when there are less than approximately one surfel per 30 square pixels.

Figure 12: *Hole detection and image reconstruction. a) Surfel object with holes. b) Hole detection (hole pixels in green). c) Image reconstruction with a Gaussian filter.*

To compare image quality of different reconstruction filters, we rendered the surfel checker plane shown in Figure 11. There is an increasing number of surfels per pixel towards the top of the image, while holes appear towards the bottom for nearest neighbor reconstruction. However, a checker plane also demonstrates limitations of the surfel representation. Because textures are applied during sampling, periodic texture patterns are stored explicitly with the object instead of by reference. In addition, flat surfaces are much more efficiently rendered using image space rasterization, where attributes can be interpolated across triangle spans.

Table 3 shows rendering performance broken down into percentages per major rendering tasks. The frame rates were measured on a 700 MHz Pentium III system with 256 MB of SDRAM using an unoptimized C version of our program. All performance numbers are averaged over one minute of an animation that arbitrarily rotates

Data	WRP	VIS	SHD	REC	CLR	fps
Output image: 256×256						
Salamander	39%	3%	28%	17%	13%	11.2
Wasp	61%	4%	21%	8%	8%	6.0
Cab	91%	2%	5%	1%	1%	2.5
Output image: 480×480						
Salamander	14%	18%	31%	22%	16%	4.6
Wasp 3to1	29%	17%	29%	15%	9%	2.7
Wasp 3LDI	48%	13%	22%	11%	6%	2.0
Wasp SS	15%	22%	28%	18%	16%	1.3
Cab	74%	7%	11%	5%	3%	1.4
Output image: 1024×1024						
Salamander	5%	14%	26%	32%	23%	1.3
Wasp	13%	19%	25%	26%	17%	1.0
Cab	16%	36%	24%	16%	8%	0.6

Table 3: *Rendering times with breakdown for warping (WRP), visibility splatting (VIS), Phong shading (SHD), image reconstruction (REC), and framebuffer clear (CLR). Reconstruction with pull-push filter. All models, except Wasp 3LDI, are 3-to-1 reduced. Wasp SS indicates 2x2 supersampling.*

the object centered at the origin. The animation was run at three different image resolutions to measure the effects of magnification and holes.

Similar to image-based rendering, the performance drops almost linearly with increasing output resolution. For 256^2 or object minification, the rendering is dominated by warping, especially for objects with many surfels. For 1024^2, or large object magnification, visibility splatting and reconstruction dominate due to the increasing number of surface holes. The performance difference between a full LDC tree (Wasp 3LDI) and a reduced LDC tree (Wasp 3to1) is mainly in the warping stage because fewer surfels have to be projected. Performance decreases linearly with supersampling, as shown for 2x2 supersampling at 480^2 resolution (Wasp SS). The same object at 1024^2 output resolution with no supersampling performs almost identically, except for slower image reconstruction due to the increased number of hole pixels.

To compare our performance to standard polygon rendering, we rendered the wasp with 128k polygons and 2.3 MB for nine textures using a software-only Windows NT OpenGL viewing program. We used GL_LINEAR_MIPMAP_NEAREST for texture filtering to achieve similar quality as with our renderer. The average performance was 3 fps using the Microsoft OpenGL implementation (opengl32.lib) and 1.7 fps using Mesa OpenGL. Our unoptimized surfel renderer achieves 2.7 fps for the same model, which compares favorably with Mesa OpenGL. We believe that further optimization will greatly improve our performance.

Choosing the block size b for the LDC tree nodes has an influence on block culling and warping performance. We found that a block size of $b = 16$ is optimal for a wide range of objects. However, the frame rates remain practically the same for different choices of b due to the fact that warping accounts for only a fraction of the overall rendering time.

Because we use a z-buffer we can render overlapping surfel objects and integrate them with traditional polygon graphics, such as OpenGL. However, the current system supports only rigid body animations. Deformable objects are difficult to represent with surfels and the current LDC tree data structure. In addition, if the surfels do not approximate the object surface well, for example after 3-to-1 reduction or in areas of high curvature, some surface holes may appear during rendering.

9 Future Extensions

A major strength of surfel rendering is that in principal we can convert any kind of synthetic or scanned object to surfels. We would like to extend our sampling approach to include volume data, point clouds, and LDIs of non-synthetic objects. We believe that substantial compression of the LDC tree can be achieved using run length encoding or wavelet-based compression techniques. The performance of our software renderer can be substantially improved by using Pentium III SSE instructions. Using an occlusion compatible traversal of the LDC tree [21], one could implement order-independent transparency and true volume rendering.

Our major goal is the design of a hardware architecture for surfel rendering. Block warping is very simple, involving only two conditionals for z-buffer tests [11]. There are no clipping calculations. All framebuffer operations, such as visibility splatting and image reconstruction, can be implemented using standard rasterization and framebuffer techniques. The rendering pipeline uses no inverse calculations, such as looking up textures from texture maps, and runtime texture filtering is very simple. There is a high degree of data locality because the system loads shape and shade simultaneously and we expect high cache performance. It is also possible to enhance an existing OpenGL rendering pipeline to efficiently support surfel rendering.

10 Conclusions

Surfel rendering is ideal for models with very high shape and shade complexity. As we move rasterization and texturing from the core rendering pipeline to the preprocessing step, the rendering cost per pixel is dramatically reduced. Rendering performance is essentially determined by warping, shading, and image reconstruction — operations that can easily exploit vectorization, parallelism, and pipelining.

Our surfel rendering pipeline offers several speed-quality tradeoffs. By decoupling image reconstruction and texture filtering we achieve much higher image quality than comparable point sample approaches. We introduce visibility splatting, which is very effective at detecting holes and increases image reconstruction performance. Antialiasing with supersampling is naturally integrated in our system. Our results demonstrate that surfel rendering is capable of high image quality at interactive frame rates. Increasing processor performance and possible hardware support will bring it into the realm of real-time performance.

11 Acknowledgments

We would like to thank Ron Perry and Ray Jones for many helpful discussions, Collin Oosterbaan and Frits Post for their contributions to an earlier version of the system, and Adam Moravanszky and Simon Schirm for developing a surfel demo application. Thanks also to Matt Brown, Mark Callahan, and Klaus Müller for contributing code, and to Larry Gritz for his help with BMRT [10]. Finally, thanks to Alyn Rockwood, Sarah Frisken, and the reviewers for their constructive comments, and to Jennifer Roderick for proofreading the paper.

References

[1] Animatek. Caviar Technology. Web page. http://www.animatek.com/.

[2] L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics*, volume 18 of *SIGGRAPH '84 Proceedings*, pages 103–108. July 1984.

[3] E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. thesis, University of Utah, Salt Lake City, December 1974.

[4] C.F. Chang, G. Bishop, and A. Lastra. LDI Tree: A Hierarchical Representation for Image-Based Rendering. In *Computer Graphics*, SIGGRAPH '99 Proceedings, pages 291–298. Los Angeles, CA, August 1999.

[5] S. E. Chen. Quicktime VR – An Image-Based Approach to Virtual Environment Navigation. In *Computer Graphics*, SIGGRAPH '95 Proceedings, pages 29–38. Los Angeles, CA, August 1995.

[6] W. Dally, L. McMillan, G. Bishop, and H. Fuchs. The Delta Tree: An Object-Centered Approach to Image-Based Rendering. Technical Report AIM-1604, AI Lab, MIT, May 1996.

[7] M. Deering. Data Complexity for Virtual Reality: Where do all the Triangles Go? In *IEEE Virtual Reality Annual International Symposium (VRAIS)*, pages 357–363. Seattle, WA, September 1993.

[8] D. Ebert, F. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing & Modeling - A Procedural Approach*. AP Professional, second edition, 1994.

[9] S. Gortler, R. Grzeszczuk, R. Szeliski, and M. Cohen. The Lumigraph. In *Computer Graphics*, SIGGRAPH '96 Proceedings, pages 43–54. New Orleans, LS, August 1996.

[10] L. Gritz. Blue Moon Rendering Tools. Web page. http://www.bmrt.org/.

[11] J. P. Grossman. *Point Sample Rendering*. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, August 1998.

[12] J. P. Grossman and W. Dally. Point Sample Rendering. In *Rendering Techniques '98*, pages 181–192. Springer, Wien, Vienna, Austria, July 1998.

[13] P. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics & Applications*, 6(11):56–67, November 1986.

[14] P. Heckbert. *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Science, June 17 1989.

[15] G. T. Herman. Discrete Multidimensional Jordan Surfaces. *CVGIP: Graphical Modeling and Image Processing*, 54(6):507–515, November 1992.

[16] A. Kaufman, D. Cohen, and R. Yagel. Volume Graphics. *Computer*, 26(7):51–64, July 1993.

[17] M. Levoy and P. Hanrahan. Light Field Rendering. In *Computer Graphics*, SIGGRAPH '96 Proceedings, pages 31–42. New Orleans, LS, August 1996.

[18] M. Levoy and T. Whitted. The Use of Points as Display Primitives. Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, 1985.

[19] D. Lischinski and A. Rappoport. Image-Based Rendering for Non-Diffuse Synthetic Scenes. In *Rendering Techniques '98*, pages 301–314. Springer, Wien, Vienna, Austria, June 1998.

[20] N. Max. Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers. In *Rendering Techniques '96*, pages 165–174. Springer, Wien, Porto, Portugal, June 1996.

[21] L. McMillan and G. Bishop. Plenoptic Modeling: An Image-Based Rendering System. In *Computer Graphics*, SIGGRAPH '95 Proceedings, pages 39–46. Los Angeles, CA, August 1995.

[22] V. Popescu and A. Lastra. High Quality 3D Image Warping by Separating Visibility from Reconstruction. Technical Report TR99-002, University of North Carolina, January 15 1999.

[23] W. T. Reeves. Particle Systems – A Technique for Modeling a Class of Fuzzy Objects. In *Computer Graphics*, volume 17 of *SIGGRAPH '83 Proceedings*, pages 359–376. July 1983.

[24] G. Schaufler. Per-Object Image Warping with Layered Impostors. In *Rendering Techniques '98*, pages 145–156. Springer, Wien, Vienna, Austria, June 1998.

[25] J. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered Depth Images. In *Computer Graphics*, SIGGRAPH '98 Proceedings, pages 231–242. Orlando, FL, July 1998.

[26] A. R. Smith. Smooth Operator. *The Economist*, pages 73–74, March 6 1999. Science and Technology Section.

[27] J. Torborg and J. Kajiya. Talisman: Commodity Real-Time 3D Graphics for the PC. In *Computer Graphics*, SIGGRAPH '96 Proceedings, pages 353–364. New Orleans, LS, August 1996.

[28] D. Voorhies and J. Foran. Reflection Vector Shading Hardware. In *Computer Graphics*, Proceedings of SIGGRAPH 94, pages 163–166. July 1994.

[29] L. Westover. Footprint Evaluation for Volume Rendering. In *Computer Graphics*, Proceedings of SIGGRAPH 90, pages 367–376. August 1990.

Surface Representations and Signal Processing

Markus Gross
Department of Computer Science
ETH Zürich
Joint work with A. Hubeli, M. Pauly, C. Sigg, K. Meyer
graphics.ethz.ch

ETH Eidgenössische Technische Hochschule Zürich

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Outline

Introduction

Triangle Signal Processing

- Fairing and Filtering of Nonmanifold Models
- Mesh Edge Detection

Point Signal Processing

- Spectral Processing of Point Clouds

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Surface Signal Processing Applications #1

Smoothing and enhancement

Dataset:
Courtesy the Digital Michelangelo Project
Stanford University

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Surface Signal Processing Applications #2

Resampling and level of detail

"electronic lense"

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Surface Signal Processing Applications #3

Feature detection

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Surface Representations

Hubeli, Gross: A Survey of Surface Representations for Geometric Modeling. TR No. 335, ETH Zürich, 2000.

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Triangle Signal Processing

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Mesh Smoothing (Fairing)

- **Fairing** is the process of removing high frequency components from geometry
- **Fairing** is an extension of lowpass filtering in signal processing
- **Fairing** relates to the solution of partial differential equations (PDEs)
- **Fairing** is easy to implement

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Mesh Frequencies

- Generalized frequencies = Eigenvectors of the Laplacian
- Let e_i and λ_i be the solutions of the Eigenproblem

$$\Delta \mathbf{x} = \lambda \cdot \mathbf{x}$$

- With the 2D Laplacian

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

- Solve by discretization L of the Laplacian

$$\mathbf{L}\mathbf{x} = \lambda \cdot \mathbf{x} \longrightarrow \mathbf{e}_i, \lambda_i$$

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Mesh Frequencies

- Set of mesh vertices **x** as a linear combination of eigenvectors of the Laplacian

$$\mathbf{x} = \sum_{k=1}^{N} \mathbf{a}_k \mathbf{e}_k$$

- Coefficients computed by inner products

$$\mathbf{a}_k = \langle \mathbf{x}, \mathbf{e}_k \rangle$$

Note: Eigenmode analysis as generalized Fourier Transform

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Example: Bunny's "Eigenmeshes"

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Lowpass Filtering

Discarding the high frequencies

$$\mathbf{x} = \sum_{k=1}^{K} \mathbf{a}_k \mathbf{e}_k + \sum_{k=K+1}^{N} \mathbf{a}_k \mathbf{e}_k$$

lowpass highpass

Numerical stability for high vertex counts

Gross, Hubeli: Eigenmeshes.
TR No. 338, ETH Zürich, 2000.

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Example

Removing high frequencies from a mesh

Surface Fairing

- Lowpass filtering corresponds to diffusion
- Iterative solution of a diffusion equation
- Let x_i be the mesh vertices
- Forward iteration of the diffusion equation yields

$$\mathbf{x}_i^{(\text{new})} = \mathbf{x}_i + \lambda \cdot \mathbf{L}\mathbf{x}_i$$

- Problem: How to discretize the Laplacian?

$$\mathbf{L}\mathbf{x}_i \approx -\frac{1}{K}\sum_{k \in N(\mathbf{x}_i)}^{K} \mathbf{x}_i - \mathbf{x}_k$$

The Bunny

Advanced

- Multilevel fairing
- Volume preservation
- Geometric Laplacians
- Curvature flow
- Anisotropic diffusion

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Manifold Fairing

Example model: manifold fairing

Hubeli, Gross: Fairing of Non-Manifolds for Visualization. IEEE Visualization 2000.

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Non-Manifold Fairing

Example model: non-manifold fairing

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Definitions

Two-manifold

Non-manifold

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Model Representation

Extended boundary representation

- An *n*-cell is defined by its *(n-1)*-D boundaries
  - A 2D surface is defined by its 1D boundaries
  - A 1D line is defined by its 0D boundaries
- Topology is separated from the geometry
  - An *n*-cell can have multiple geometric realizations
  - Fairing can be applied across disconnected *n*-cells

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Non-manifold Fairing

Problem

- Compute the Laplacian at singularities

Our solution

- An *n*-cell is smoothed by moving its *n*-vertices
- The *m*-vertices ($m < n$) in the *n*-cell are smoothed by moving their *n*-vertex neighbors

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

The Operator in 1D

Example: fairing of a one-cell

Close-up of the one-cell on a zero-vertex

- Zero-vertex
- One-vertices

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

The Operator in 1D

Example: fairing of a one-cell

Evaluate the Laplacian in one-neighborhood

- Zero-vertex
- One-vertices
- Laplacian vector

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

The Operator in 1D

Example: fairing of a one-cell

Freeze the zero-vertex

- Zero-vertex
- One-vertices
- Displacement vector

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

The Operator in 1D

Example: fairing of a one-cell

One-cell after the smoothing step

- Zero-vertex
- One-vertices

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Non-manifold Fairing Operator

- Assumption on the operator

$$\Delta \mathbf{x}_i = \sum_{k=1}^{n} c_{j_k,i} \cdot \mathbf{x}_{j_k} - \mathbf{x}_i$$

- Increase the support for every vertex

$$\begin{bmatrix} \Delta \mathbf{x}_i \\ \Delta \mathbf{x}_{j_1} \\ \vdots \\ \Delta \mathbf{x}_{j_n} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad \mathbf{x}_{j_1}, \cdots, \mathbf{x}_{j_n} \in N_1(\mathbf{x}_i)$$

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Non-manifold Fairing Operator

- Constraints lead to a system of equations

$$\begin{bmatrix} 1 \\ -c_{j_1,i} \\ \vdots \\ -c_{j_n,i} \end{bmatrix} \mathbf{x}_i^{(l+1)} = \begin{bmatrix} \Delta \mathbf{x}_i \\ \Delta \mathbf{x}_{j_1} \\ \vdots \\ \Delta \mathbf{x}_{j_n} \end{bmatrix} + \begin{bmatrix} 1 \\ -c_{j_1,i} \\ \vdots \\ -c_{j_n,i} \end{bmatrix} \mathbf{x}_i^{(l)}$$

- Least squares solution of the system

$$\mathbf{x}_i^{(l+1)} = \mathbf{x}_i^{(l)} + \frac{\Delta \mathbf{x}_i - \sum_{k=1}^{n} c_{j_k,i} \Delta \mathbf{x}_{j_k}}{1 + \sum_{k=1}^{n} c_{j_k,i}^2}$$

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Results

- Feature preservation
- Multilevel smoothing

A geological model

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Automatic Feature Extraction

3 stage procedure

- I. Edge detection
- II. Patching
- III. Skeletonizing

Hubeli, Meyer, Gross: Mesh Edge Detection.
TR, ETH Zürich, 2001.

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Detection of Mesh Features

Best-fit polynomials

- Construct a parameter plane
- Project vertices on a parameter plane
- Fit a curve through the points
- Compute the curvature of a curve at edge e

$$w(e) = P''(e)$$

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Results

The golf club model

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Results

The mannequin model

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Signal Processing of Point Clouds

Dataset:
Courtesy the Digital Michelangelo Project
Stanford University

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Goals

- Extend Fourier transform to 2-manifold surfaces
  Versatile alternative surface representation
- Spectral filtering
  Noise-removal, microstructure analysis, enhancement
- Adaptive resampling
  Point decimation, continuous LOD
- Efficiency
  Processing of large point-sampled models

Pauly, Gross: Spectral Processing of Point Sampled Geometry. Siggraph 2001.

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Fourier Transform

Benefits:

- Sharp concept of frequency
- Extensive theory in signal processing
- Error control

$$\mathbf{X}_n = \sum_{k=1}^{N} \mathbf{x}_k e^{-j2\pi nk}$$

Limitations:

- Euclidean domain - global parameterization
- Regular sampling
- No spatial localization

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Spectral Processing Pipeline
Create Patch Layout
Patch Blending
Input Data
Raw Patches
Processed Patches
Final Surface
SDA
DFT
Spectral Analysis
Inverse DFT
Re-sampling
Regular Grid
Fourier Spectrum
Modified Spectrum
Filtered Patch

Patch Layout Generation
Clustering ⇨ Optimization
Samples ⇨ Clusters ⇨ Patches
SIGGRAPH 2001
EXPLORE INTERACTION AND DIGITAL IMAGES

Scattered Data Approximation

Hierarchical Approach: Splatting ⇨ Push ⇨ Pull

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Spectral Analysis

- 2D Discrete Fourier Transform (FFT)
- Direct manipulation of coefficients
- Linear filtering (lowpass, highpass)
- Advanced spectral estimation (Wiener)
- Sampling theorem (decimation)
- Parseval's theorem (error)

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Spectral Analysis

Dataset:
Courtesy the Digital Michelangelo Project
Stanford University

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Reconstruction

- Filtering can lead to discontinuities at patch boundaries
- Create patch overlap and blend adjacent patches
- Blending function: splatting ⇨ convolution filtering

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Resampling

Dataset:
Courtesy the Digital Michelangelo Project
Stanford University

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Local Analysis - STFT

Dataset:
Courtesy the Digital Michelangelo Project
Stanford University

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Restoration – Wiener Filtering

Original noise+blur | Gaussian Filter | Restoration Filter | Patch Layout

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Closing Remarks

- Processing of „geometric signals" is highly important in Computer Graphics
- Generalize methods to irregular geometry
- More sophisticated surface representations are required
- Point sampled geometry is clearly an alternative for very large datasets

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Spectral Processing of Point-Sampled Geometry

Mark Pauly Markus Gross

ETH Zurich

Figure 1: Spectral processing pipeline. Processing stages are depicted as rectangles, rounded boxes represent input/output data of each stage. Gray background color indicates the preprocessing phase.

Abstract

We present a new framework for processing point-sampled objects using spectral methods. By establishing a concept of local frequencies on geometry, we introduce a versatile spectral representation that provides a rich repository of signal processing algorithms. Based on an adaptive tesselation of the model surface into regularly resampled displacement fields, our method computes a set of windowed Fourier transforms creating a spectral decomposition of the model. Direct analysis and manipulation of the spectral coefficients supports effective filtering, resampling, power spectrum analysis and local error control. Our algorithms operate directly on points and normals, requiring no vertex connectivity information. They are computationally efficient, robust and amenable to hardware acceleration. We demonstrate the performance of our framework on a selection of example applications including noise removal, enhancement, restoration and subsampling.

Keywords: Signal processing, spectral filtering, subsampling, Fourier transform, point-based representations

1 Introduction

Today's range sensing devices are capable of producing highly detailed surface models that contain hundreds of millions of sample points. Due to a variety of physical effects and limitations of the model acquisition procedure, raw range datasets are prone to various kinds of noise and distortions, requiring sophisticated processing methods to improve the model quality. In spite of the recent advances made in mesh optimization, traditional mesh processing algorithms approach their limits, since triangle primitives implicitly store information about local surface topology including vertex valence or adjacency. This leads to a substantial additional overhead in computation time and memory costs. With increasing model size we thus experience a shift from triangle mesh representations towards purely point-based surface descriptions. For instance, recent work concentrated on point-based rendering pipelines [17, 18] where point samples without connectivity are proposed as rendering primitives. Surprisingly, however, little work has been done so far on direct processing or manipulation of point-sampled geometry. In this paper we present a new framework for spectral analysis and processing of point-sampled objects. The method operates directly on irregular point sets with normals and does not require any a priori connectivity information. Our framework extends so-called windowed Fourier transforms - a concept being well known from signal processing - to geometry.

The Fourier transform is a powerful and widely used tool for data analysis and manipulation. In particular, image processing techniques successfully exploit frequency representations to implement a variety of advanced spectral processing algorithms comprising noise removal, enhancement, feature detection and extraction, up/down-sampling, etc. [6]. Extending this approach to general geometric models is difficult due to a number of intrinsic limitations of the conventional Fourier transform: First, it requires a global parameterization on which the basis functions are defined. Second, most FT algorithms require a regular sampling pattern [16]. These prerequisites are usually not satisfied by common discrete geometry, rendering the standard Fourier transform inoperable. A further limitation of traditional Fourier representations is the lack of spatial localization making it impractical for local data analysis. We will show how these limitations can be overcome and present a generalization of the windowed FT to general 2-manifolds. The basic idea behind our framework is to preprocess the raw irregular point cloud into a model representation that describes the object surface with a set of regularly resampled height fields. These surface patches form "windows" in which we compute a discrete Fourier transform to obtain a set of local frequency spectra. Although being confined to individual surface patches, our windowed FT provides a powerful and versatile mechanism for both local and global processing. The concept of frequency on point-sampled geometry gives us access to the vast space of sophisticated spectral methods resulting from tens of years of research in signal processing.

In this paper we will focus on two classes of such methods: Spectral filtering and resampling. We will point out how sophisticated filtering operations can be implemented elegantly by analyzing and modifying the coefficients of the frequency spectrum. Possible applications include noise removal, analysis of the surface microstructure and enhancement. Further we present a fast algorithm for adaptively resampling the point geometry, using the spectral representation to determine optimal sampling rates. This method is particularly useful for reducing the complexity of overly dense point-sampled models. By using FFT and other signal processing algorithms our framework is efficient in computation and memory costs, amenable to hardware acceleration and allows us to process hundreds of millions of points on contemporary PCs.

1.1 Previous Work

Extending the concept of frequency onto geometry has gained increasing attention over the last years. Conceptually, this generalization can be accomplished by the eigenfunctions of the Laplacian. Taubin [19] pioneered spectral methods for irregular

meshes using a discrete Laplacian to implement iterative Gaussian smoothing for triangle meshes. This method has later been improved by Desbrun et al [4] who tackled the difficulty of discretizing a geometric Laplacian by introducing curvature flow for noise removal. Kobbelt [12] presented a novel concept for multiresolution variational fairing and modeling, where high mesh frequencies are attenuated by iteratively solving discretized Laplacian equations. While being based on signal processing methodology, these algorithms do not compute an explicit spectral representation of the object surface. As a consequence, typical filters such as Gaussian smoothing have to be implemented in the spatial domain. In contrast, our method robustly generates a set of local Fourier spectra that can be explicitly analyzed and manipulated. This supports much more powerful filtering, e.g. least-squares optimal or inverse, feature enhancement and Fourier sampling. Specifically, we can examine the power spectrum of the surface signal to estimate optimal filter parameters or determine the noise level present in the data.

Guskov et al [8] introduced signal processing methods using subdivision and pyramid algorithms. While they achieve qualitatively remarkable effects such as band-pass filtering and enhancement, their notion of frequency is based on detail vectors between different levels of a mesh hierarchy. Our scheme uses the Fourier transform, which efficiently computes a projection into the space of eigenfunctions of the Laplacian. Within this framework concepts like natural vibration modes or spatial frequency are solidly founded in the theory of differential calculus. This allows us to exploit many results from the extensive work on Fourier theory including Sampling or Parseval's theorems. With the former we obtain a profound means to determine optimal sampling rates, while the latter supports local error control. Lately, Karni and Gotsman [11] introduced a method for spectral compression of triangle meshes that is based on a fixed partitioning of the mesh into submeshes. Effective compression is achieved by a direct decomposition of these patches into the eigenfunctions of the Laplacian. While their notion of frequency is strictly local, the explicit eigenvector computations are expensive and potentially unstable, constituting serious limits for the efficient processing of large patch sizes.

All of the above methods focus on triangle meshes, relying heavily on connectivity information between vertices. In contrast, our method is purely point-based, requiring only vertex position and associated normals. This allows direct processing of scanned data without the need to construct polygonal meshes, making it particularly suitable for the very large models obtained with modern range scanners [14].

1.2 Algorithm Overview

Figure 1 gives a high-level overview of our spectral processing pipeline. In the first stage we split the point-sampled model into a number of overlapping patches. A patch is defined as a collection of sample points that represents a connected region of the underlying surface. The tesselation is done in such a way that the surface represented by each patch can be expressed as a displacement field over a planar domain. The so generated patch layout forms the basis of our windowed Fourier transform and the following stages operate locally on individual patches. First the patch surface is resampled on a regular grid using a fast scattered data approximation (SDA). Then we apply a Discrete Fourier Transform (DFT) to obtain the spectral representation of the patch surface. Using appropriate spectral filters we can directly manipulate the Fourier spectrum to achieve a variety of effects such as de-noising or enhancement. A subsequent inverse DFT reconstructs the filtered patch surface in the spatial domain. We can then also utilize the spectral information to adaptively resample the patch surface. At the end of the pipeline is the reconstruction stage, where the processed patches are stitched together to yield the final object surface. This requires careful attention at the patch boundaries, where we create a smooth transition by blending the overlapping parts of adjacent patch surfaces. As indicated in Figure 1, the processing pipeline can be split into two phases: Patch layout generation, SDA and DFT can be separated into a *preprocessing* step. We also precompute the parameter mapping between adjacent patches and the blending function used in the reconstruction. This leaves spectral analysis, inverse DFT, resampling and reconstruction as the actual processing stages. We will now describe the individual stages of the processing pipeline in more detail, following the order depicted in Figure 1.

2 Creating the Patch Layout

In this section we present a new method for creating a continuous surface representation from an unordered set of sample points and associated normals. We assume that the sample points represent a smooth two-manifold of arbitrary topology and possibly multiple connected components. Further we require the sampling to be dense enough in the sense that adjacent points in 3-space with similar normal orientation belong to the same local neighborhood of the surface [1].

The goal is to describe the object surface with a set of patches that can be represented as scalar height fields. To achieve this we grow patches by accumulating adjacent sample points subject to a *normal cone condition*. This criterion states that the aperture angle of the cone spanned by the normals of a patch's sample points is less than π. Bounding the normal cone width guarantees that no foldovers can occur, i.e. that we can bijectively map the patch surface to a height field representation over a planar domain. In practice we choose $\pi/2$ as maximum normal cone width, as this provides a more uniform parameter mapping and thus makes the following scattered data approximation more robust. We compute the normal cone with an adapted version of Gärtner's miniball algorithm [3]. It determines the smallest enclosing sphere of a set of normal vectors interpreted as points on the unit sphere. The vector through the center of the miniball gives the normal cone center and from its radius we can determine the aperture angle (see Figure 2).

Figure 2: The miniball algorithm provides an accurate estimation of the cone spanned by a set of normal vectors (red).

Our algorithm for generating the patch layout proceeds in two stages: The first stage creates an initial fine-grain patch layout by clustering adjacent sample points, while the second stage merges adjacent clusters into patches using an optimization approach (see also Figure 4). During this iterative growth we ensure at all times that the normal cone condition is satisfied.

Clustering. We first arrange the sample points in a binary space partition (BSP) tree by recursively splitting the sampling set along the longest axis of its bounding box. The BSP structure implicitly encodes the 3D adjacency information, requiring approx. 10% of the input model size in additional memory overhead. We choose the leaves of the tree, which contain exactly one sample point, as our initial clusters. Now we successively merge clusters with a common parent in the BSP tree, since these are neighbors in 3-space. However, as Figure 3 illustrates, a cluster has potentially many other neighbors and allowing only sibling clusters to be merged is too restrictive to lead to a useful patch layout. Therefore we stop the clustering stage as soon as the clusters reach a suitable size (typically 25-100 sample points, depending on model size). We will call the patches created by clustering *leaf patches*, as they are leaves of the final BSP tree.

Patch Merging. At the beginning of the second stage we have to compute local neighborhood information, i.e. for each leaf patch we need to determine a list of all adjacent leaf patches. A leaf patch is confined by six BSP split planes, each of which corresponds to an internal node of the tree. In a first step we collect for each split plane all leaf patches that border on either side of the plane. Then we project the bounding boxes of these leaf patches onto the split plane and check for overlaps of the projections. If an overlap occurs, we mark the leaf patches as neighbors (see Figure 3).

Figure 3: Neighborhood information for leaf patches (2D for illustration). Thick lines (resp. black dots) indicate the BSP split planes that confine the green patch. Patch 1 and 6 are neighbors because their projections onto the split plane overlap. Note that neighbors can be distributed over the whole BSP tree.

Using the adjacency information of the leaf patches, we can now apply a more sophisticated merging technique. The idea is to use an optimization approach that merges patches according to a local quality metric Φ. Let P_i and P_j be neighboring patches such that $P = (P_i, P_j)$ is a potential merge candidate pair. Then $\Phi(P)$ gives a relative measure of the quality of the patch layout obtained after merging P_i and P_j, with small values of Φ indicating a high quality. By iteratively merging the pair with the highest quality gain we can locally optimize the patch layout. Merge candidate pairs are arranged in a priority queue that is ordered by increasing Φ and initialized with all pairs of neighboring leaf patches. Now we successively remove the pair with the highest priority (i.e. lowest Φ) from the queue and merge the two patches if their union satisfies the normal cone condition. Then we update the priorities and neighborhood information of all affected pairs accordingly. Φ is determined using the following formula:

$$\Phi(P) = \Phi_{Size}(P) \cdot \Phi_{NC}(P) \cdot \Phi_B(P) \cdot \Phi_{Reg}(P) . \qquad (1)$$

Each of the terms of Equation 1 seeks to optimize a specific quality feature of the final patch layout. Since the individual quality measures are difficult to normalize, we combine them in a product to yield Φ.

- Φ_{Size} assigns a high priority to small patches and thus reduces undesirable fragmentation:

  $$\Phi_{Size}(P_i, P_j) = |P_i| \cdot |P_j| ,$$

  where $|P_k|$ is the number of samples in patch P_k.

- Φ_{NC} penalizes the increase in normal cone width of the merged patch $P_i \cup P_j$:

  $$\Phi_{NC}(P_i, P_j) = \alpha(P_i \cup P_j) - max\{\alpha(P_i), \alpha(P_j)\} ,$$

  where $\alpha(P_k)$ is the aperture angle of the normal cone of P_k. This leads to a better adaptation of the patch layout to the local curvature of the underlying surface, since flat regions are quickly merged into large patches, while highly curved regions will be covered by smaller patches.

- Φ_B is introduced to control the boundary of the patches:

  $$\Phi_B(P_i, P_j) = \frac{L(P_i \cup P_j)}{B(P_i) + B(P_j) - B(P_i \cup P_j)} ,$$

  where $L(P_k)$ counts the number of leaf patches of P_k, while $B(P_k)$ counts only those leaf patches that lie on its boundary[1]. Thus Φ_B seeks to minimize the length of the patch boundary relative to the patch area. This will favour roughly circular-shaped patches, which is beneficial for the later SDA and DFT processing stages.

- Φ_{Reg} is used to regularize the patch distribution:

  $$\Phi_{Reg}(P_i, P_j) = \frac{E_s(P_i \cup P_j)}{E_s(P_i) + E_s(P_j)} ,$$

  where $E_s(P_k) = \sum_{l \in N(k)} \sigma \|\boldsymbol{c}_k - \boldsymbol{c}_l\|$ is a spring energy term.

  It is derived by placing a spring with tension σ on each edge from the center $\boldsymbol{c}_k = (c_x, c_y, c_z)$ of P_k to the center of all neighboring patches $P_l, l \in N(k)$.

The merging process terminates as soon as no more patches can be merged without violating the normal cone condition. To have additional control over the granularity of the patch layout, the user can specify a maximum patch size in terms of number of sample points or spatial extent. One could also assign different weights to each of the individual quality measures by using additional exponents in Equation 1. In practice we found, however, that equal weights generally lead to satisfactory results.

Figure 4 illustrates the two stages of the patch layout generation for the simple example of a sphere. Figure 13, 15 and 16 show the final patch layout for more complex point-sampled models. Observe how the distribution and shape of the patches adapts to the geometry, i.e. in regions of high curvature we have more and smaller patches than in flat parts of the surface.

Figure 4: The patch layout is created by first merging sample points into clusters and then merging clusters into patches.

3 Scattered Data Approximation

The patch generation algorithm does not require nor create any connectivity information of individual samples. At this point a patch is simply a set of irregular sample points without any additional knowledge about the spatial relations between them. The goal of the next stage of the processing pipeline is to create a continuous surface representation that describes the patch surface as a scalar displacement field sampled at regular intervals.

Functional Mapping. The first step in doing so is to define the local coordinate frame of the height field representation. We call the plane specified by the center of a patch's normal cone its *base plane* (see Figure 2). It defines a coordinate transformation T that maps a sample $\boldsymbol{p} = (x, y, z)$ given in world coordinates to $\boldsymbol{p}' = T\boldsymbol{p} = (u, v, h)$, where h is the displacement from the base plane at parameter values (u, v). Then we compute the smallest enclosing box of all (u, v) pairs on the base plane. This allows us to optimally align the rectangular sampling grid with the sample points (see Figure 5).

Overlap. As mentioned before, we need to let patches overlap to handle boundary effects during the reconstruction stage. This is achieved by increasing the size of the parameter rectangle and including all sample points from neighboring patches that map into the enlarged parameter domain (see Figure 5, right). We check for each boundary point, if it satisfies the normal cone condition. Here it has proven useful to increase the maximum normal cone width for boundary points by $\pi/8$, allowing more information from the underlying surface to be included in the

[1] internally we store a patch as a list of leaf patches that constitute the patch, hence L and B can easily be evaluated.

Figure 5: Left: Smallest enclosing box of the interior sample points in the parameter plane. Right: extended parameter domain with regular sampling grid. The red dots indicate boundary points that have been included from neighboring patches.

overlap region. For the applications of this paper we found an overlap size of 10% of the interior parameter box sufficient for creating a smooth transition at the boundaries during reconstruction.

Figure 6: Scattered data approximation for a patch on the forehead of the Happy Buddha. Different colors illustrate the displacement from the base plane with dark blue pixels indicating holes. The latter are quickly filled during the pull phase by halving the resolution in each dimension. The final patch is created in the push phase, where the arrows indicate which grids are blended.

Regular Sampling. As Figure 5 illustrates, the sampling pattern on the base plane is in general *irregular*. Standard spectral transforms such as Cosine or Fourier transforms require regularly sampled input data, however. Therefore we apply a fast, hierarchical scattered data approximation, which projects the displacement field onto a regular grid. We use linear B-Spline basis functions centered at each grid point, such that the support of each basis touches the center of its eight neighboring basis functions. Linear B-Splines allow for efficient evaluation due to their compact support, and interpolate the original samples provided the sampling rate is sufficiently high. We utilize the scattered data approximation method presented by Gortler et al. [7] for image based rendering and refer to there for details. As shown in Figure 6, this algorithm proceeds in three phases:

- *Splatting* computes weighted averages of the sample points to create an initial approximation of the coefficients of the basis functions. Due to the irregularity of the sample points this first approximation may still contain holes, i.e. undefined regions, that need to be filled.
- *Pull* iteratively generates lower resolution approximations through hierarchical convolution filtering.
- *Push* fills the holes in the final patch by successively blending approximations at different resolutions.

We set the grid size $N \cdot M$ proportional to the number n of interior and boundary points of the patch: $N = \lceil s_1 K \rceil$, $M = \lceil s_2 K \rceil$, where $K = \kappa\sqrt{n/(s_1 s_2)}$ with oversampling factor κ (see Figure 5). For all our models we chose $\kappa = 1$, which leads to an approximation error[1] of less than 0.01%. Note that substantially smaller grid sizes introduce some noticeable low pass-filtering due to the averaging of the splatting phase.

As explained above, our patch layout describes a surface by a set of scalar-valued displacement coefficients. A similar approach was taken for displaced subdivision surfaces [13] and normal meshes [9] that achieve staggering mesh compression rates. Both methods, however, require the costly computation of a coarse triangle mesh to obtain the base domain for the displacements. In addition, nontrivial parameterizations are mandatory to keep track of coefficients. This creates a substantial computational overhead, making both representations less suitable for our purposes. The patch layout scheme is much more simple and neither requires triangle meshes nor mesh simplification. The described procedures operate directly on point clouds, making them fast and efficient even for very large datasets (see also Table 1).

4 Discrete Fourier Transform

The surface representation created by the SDA describes a point-sampled model with a set of overlapping patches, each of which satisfies the Fourier requirements of regular sampling distribution and Euclidean domain. We can thus apply a discrete Fourier transform (DFT) using a 2D box window function[2] to obtain a spectral decomposition of the surface model. In order to better understand what follows, we give a brief introduction of the DFT, mentioning only those properties that we directly exploit in our algorithms. For more details we refer to textbooks such as [2].

The two-dimensional DFT is essentially a basis transform into the space of eigenfunctions of the Laplacian. Given an input signal $\mathbf{x}$ defined on a regular grid of size $N \cdot M$, the coefficients of the DFT $\mathbf{X} = F(\mathbf{x})$ can be written as

$$\mathbf{X}_{k,l} = \sum_{n=1}^{N} \sum_{m=1}^{M} \mathbf{x}_{n,m} e^{-j\frac{2\pi}{N}kn - j\frac{2\pi}{M}lm}, \tag{2}$$

where $j = \sqrt{-1}$ and $(k, l) \in ([1, N], [1, M])$ are the discrete frequencies. Using a 2D Fast Fourier transform (FFT), we can compute the DFT in $O(NM\log(NM))$ operations, instead of $O(N^2M^2)$ operations required for the direct evaluation of Equation 2 [5]. Fundamental for the implementation of the spectral filters described below is the convolution theorem. It relates a convolution $\mathbf{x} \otimes \mathbf{y}$ of two signals $\mathbf{x}$ and $\mathbf{y}$ in the spatial domain with a multiplication in the spectral domain via the Fourier transform:

$$F(\mathbf{x} \otimes \mathbf{y}) = F(\mathbf{x}) \cdot F(\mathbf{y}). \tag{3}$$

Instead of doing a computationally expensive (filtering) convolution in the spatial domain, we can thus perform a cheap multiplication in the frequency domain using the DFT and its inverse.

Power Spectrum. The power spectrum P is the Fourier transform of the autocorrelation function of a signal $\mathbf{x}$:

$$P(\mathbf{x}) = F(\mathbf{x} \otimes \mathbf{x}^*), \tag{4}$$

where the asterisk denotes the complex conjugate. Power spectrum estimation is a widely used tool in data analysis. As illustrated in Figure 7, it allows to estimate the signal-to-noise ratio

[1] measured as the RMS error between original sampling points and reconstructed points from the SDA surface.
[2] other windows, e.g. Gaussian or Hanning, can be used too.

Figure 7: Power spectrum estimation. Normalized logarithmic plot of the power spectrum of a typical patch surface (left). The annotation at each circle indicates the relative amount of power contained within the circle. On the right an (idealized) illustration for signal-to-noise ratio estimation.

and can thus be used to optimize filter characteristics such as cut-off frequency or pass- and stop-band. Using the convolution theorem, we can directly compute P from the spectral coefficients, i.e. $P(\mathbf{x}) = |\mathbf{X}|^2$ with $\mathbf{X} = F(\mathbf{x})$.

Error Estimation. Another important result from Fourier theory is Parseval's theorem

$$\sum_n \sum_m \mathbf{x}_{n,m}^2 = \frac{1}{NM} \sum_k \sum_l |\mathbf{X}_{k,l}|^2 , \tag{5}$$

which relates the signal energy in spatial and frequency domains. We can utilize this property for estimating the error introduced by filtering the spectral coefficients: Suppose $\mathbf{Y} = F(\mathbf{y})$ is a filtered version of the spectrum $\mathbf{X} = F(\mathbf{x})$. Then the L_2-norm of the difference $\mathbf{x} - \mathbf{y}$ is given by

$$\|\mathbf{x} - \mathbf{y}\|_2 = \sqrt{\frac{1}{NM} \sum_k \sum_l |\mathbf{X}_{k,l} - \mathbf{Y}_{k,l}|^2} \tag{6}$$

Thus we have explicit control over the error introduced when modifying the spectral coefficients.

5 Spectral Analysis

The frequency spectrum obtained by the DFT provides us with a spectral representation of the patch surface. The basis functions in the spectral domain represent natural vibration modes of the surface, thus relating specific surface features to certain frequency intervals. Low frequencies, for instance, represent the overall geometric shape, while high frequencies account for small geometric detail and noise. With these semantics we can perform elaborate filtering operations by manipulating the frequency spectrum. Figure 8 shows various such filters with the corresponding transfer functions. Low-pass filtering eliminates high frequencies and thus leads to surface smoothing. Observe that the ideal low-pass filter with its sharp cut-off frequency produces the well-known ringing artefacts [6], which are clearly visible as surface ripples in the image. This phenomenon can easily be explained with the convolution theorem: Multiplying the frequency spectrum with the box function of the ideal filter is equivalent to convolving the original surface with a sinc function (see Figure 9, left image). When using a Gaussian transfer function for surface smoothing, no ringing artefacts occur, since the corresponding filter kernel in the spatial domain is also a Gaussian (Figure 9, right). The lower left image of Figure 8 shows a band-stop filter that attenuates middle frequencies. This leads to overall surface smoothing while still retaining the microstructure of the surface material. We can also enhance certain features of the surface by scaling the frequency spectrum appropriately (Figure 8, lower right).

Signal Restoration. Real imaging systems often introduce some undesirable low-pass filtering since the physical apparatus does not have a perfect delta-function response. This blurring can be reduced with an inverse filter that amplifies high frequencies. Inverse filtering, however, tends to instabilities and is extremely sensitive to noise. To restore the object surface in the

Figure 8: Spectral filters applied to St. Matthew data set. The 2D transfer function is obtained by rotating the shown 1D function around the vertical axis.

presence of blur *and* noise we apply a least-squares optimal filter or *Wiener* filter [10]. Suppose we have a blurred and noisy

Figure 9: Convolution filter kernels in the spatial domain.

input signal $\mathbf{x}'$ and we want to reconstruct the underlying original signal $\mathbf{x}$. Applying the spectral filter function ϕ to the Fourier transform of $\mathbf{x}'$ yields the filtered signal $\mathbf{y} = F^{-1}(\phi \cdot F(\mathbf{x}'))$. The goal is to determine ϕ such that

$$\int |\mathbf{y}(t) - \mathbf{x}(t)|^2 dt = \int |\mathbf{Y}(f) - \mathbf{X}(f)|^2 df \tag{7}$$

is minimized. As shown in [10] this can be achieved by power spectrum analysis, yielding

$$\phi(f) = \frac{|\mathbf{S}(f)|^2}{|\mathbf{S}(f)|^2 + |\mathbf{N}(f)|^2} \cdot \frac{1}{|\mathbf{R}(f)|^2} , \tag{8}$$

where $|\mathbf{S}|^2$, $|\mathbf{N}|^2$ and $|\mathbf{R}|^2$ are the (estimated) power spectra of the blurred signal, noise and imaging system response, respectively (see Figure 7, right). Note that effective Wiener filtering relies on an accurate estimation of these quantities, which often requires some knowledge of the system's impulse response (see also Figure 13).

6 Resampling

After manipulating the frequency spectrum, an inverse DFT takes us back into the spatial domain. If we only want to filter the input model without affecting the sampling pattern and density, we sample the filtered patch surface at the parameter values of the original sample points. However, for many applications it is desirable to have some mechanism for adaptively refining a surface through upsampling or reducing the model size through subsampling. The latter is particularly important when dealing with very large datasets, which often cannot be handled well in their full resolution.

Fourier Sampling. The Fourier spectrum provides us with an elegant way to estimate the optimal sampling rate when subsampling the patch surface. Suppose we have a bandlimited signal $\mathbf{x}$ with Nyquist frequency f_c, i.e. all coefficients associated with frequencies greater than f_c are zero. Then the sampling theorem of Fourier theory states that we can reconstruct $\mathbf{x}$ exactly if the sampling interval is less or equal to $1/(2f_c)$. Thus to (uniformly) subsample the patch surface we proceed as follows: First we low-pass filter the frequency spectrum to obtain a bandlimited signal. Using the power spectrum and error estimation described in Section 4, we can adjust the filter parameters to match the desired maximum error. Then we apply the sampling theorem to compute the optimal sampling interval for the filtered signal. Thus we can control the sampling rate by specifying the maximum error tolerance.

Sampling Points and Normals. To determine a patch surface point at arbitrary parameter values, our current implementation uses bilinear interpolation and computes the corresponding normals with first order divided differences. Higher order schemes can easily be implemented as well. Alternatively, we could use the subdivision scheme presented in [13], where the scalar displacements are interpreted as subdivision coefficients.

7 Reconstruction

At this stage of the processing pipeline we need to reassemble the object surface by stitching together the processed patches. Some care needs to be taken here, since individual processing of patches can lead to discontinuities at the patch boundaries. To create a smooth transition between patches we blend the patch surfaces in their regions of overlap. The blending is done by computing a convex combination of corresponding points of neighboring patches using weights given by a precomputed blending function.

Parameter Mapping. To blend points from neighboring patches we need to define a mapping between the different parameter domains in the regions of overlap. Suppose we have an interior point $\boldsymbol{p}_1 = (u_1, v_1, h_1)$ in patch P_1 and that the overlap of patch P_2 also covers $\boldsymbol{p}_1$. The corresponding parameter values (u_2, v_2) can be determined by first mapping $\boldsymbol{p}_1$ to world space using the inverse mapping transform of P_1. This gives us the point $\boldsymbol{q}_1$ which is then projected onto the base plane of P_2. Now we sample P_2 at (u_2, v_2) to obtain $\boldsymbol{p}_2$, which is mapped to world coordinates to yield $\boldsymbol{q}_2$. The blended sample point $\boldsymbol{q}$ is then computed as the convex combination $\boldsymbol{q} = (\omega_1 \boldsymbol{q}_1 + \omega_2 \boldsymbol{q}_2)/(\omega_1 + \omega_2)$, where ω_1 and ω_2 are the weights given by the blending functions at (u_1, v_1) and (u_2, v_2), respectively. Multiple patch overlaps are handled analogously. To improve performance we can store the parameter mapping in (multi-layered) texture maps using bilinear interpolation to compute the parameter correspondence of intermediate points.

Blending function. The blending function for a patch is generated by first splatting all interior samples (see Figure 5) onto a regular grid. This grid is aligned to the sample points in the same manner as the SDA grid, but can be of different resolution. Subsequent convolution filtering with a Gaussian kernel creates a smooth decay to zero at the patch boundary (see Figure 10). Thus the more we approach the rim of the overlap region of the patch, the smaller will the influence of the sample point be in the convex combination of the blended sample. Splatting can be done using conventional graphics hardware with splat size equivalent to the size of the convolution matrix. The latter is chosen to match the size of the overlap as defined in Section 3, which for all our test cases was sufficient to guarantee hole-free reconstruction. Note that blending function and parameter correspondence are generated in the preprocessing phase, i.e. operate on the original sample points prior to spectral filtering.

Figure 10: Blending function for the patch of Figure 6. The left image shows the splatted interior sample points with black dots indicating the sample positions. The right image shows the blending function after convolution filtering.

Blending Normals. A smooth boundary transition of normals is achieved analogously to the convex blending used for geometric position. Substantial changes of the shape of the patch surfaces, however, may cause this simple method to fail. Consider the situation of Figure 11, where the patch processing has created a significant gap between the two surfaces. While the blending of position works fine, the blended normals do not adequately describe the tangent plane of the surface. We detect such cases using a simple conservative heuristic that takes into account the positions of the initial and blended points. The correct normal can then be approximated by sampling a small number of points in the vicinity of the considered sample and fitting a least-squares tangent plane through these points. While computationally more expensive, this normal estimation is rarely required. In all our test cases less than 1% of all normals have been computed in this way, rendering the additional overhead negligible.

Figure 11: In case of substantial filtering, the blended normals (green) can differ significantly from the correct normals (red).

Blending the sampling rate. The resampling strategy described in Section 6 uses the sampling theorem to determine the sampling rate for each patch. Since adjacent patches can differ significantly in their spectral representation, this may lead to sharp changes of the sample density at the patch boundaries. For most applications, however, a smooth transition of the sampling rate is preferable. To achieve this, we blend the sampling rate analogously to the blending of geometric positions and normals.

Figure 12: Blending the sampling rate at the patch boundary. Left: Continuous sampling rate. Middle: Discretized area weighted sampling rate. Right: Resulting sampling pattern.

This gives us a continuous function describing the sampling rate, which is then discretized on a regular grid. Each grid value serves as an index into a list of precomputed sampling patterns,

generated using Mitchell's algorithm for Poisson disk sampling [15]. Thus we achieve a gradual change of sampling density at patch boundaries (see Figure 12).

8 Results & Discussion

The filtering and subsampling methods described in Sections 5 and 6 operate locally on individual patches. To achieve global effects we apply the same filter, resp. the same relative error bound for subsampling, to each patch after appropriate scaling of the frequency spectrum.

Figure 13 shows a Gaussian and a restoration filter applied to a laser range scan of a human head. In our current implementation the parameters of the Wiener filter have to be adjusted interactively by investigating the power spectra of a small number of patches to determine the signal-to-noise ratio. Observe how the Wiener filter preserves geometric features that are smoothed away by the Gaussian.

Interactive local editing operations on the head of the St. Matthew statue are illustrated in Figure 15. The user can draw a curve on the surface to mark a region of interest (red and blue circles). The patches are split adaptively at this curve and spectral processing is only applied to those patches within the specified area. Note how the patch blending automatically creates a smooth transition between the filtered and non-filtered areas.

Figures 14 and 16 show subsampling for Michelangelo's David. The original data set contains 4,128,614 vertices, which have been reduced to 287,165 in the subsampled version, corresponding to approx. 98% of patch signal power. While the sampling of the original model is fairly uniform, the spectral subsampling creates a nonuniform sampling distribution that locally adapts to the geometry. Strictly speaking, the notion of error as established by Equations 5 and 6 only holds for the patch interior. The local frequency information in the overlap region - and hence the error - is influenced by the blending function, which in turn results from the convolution process depicted in Figure 10. Our experimental investigations showed, however, that using the same relative maximum error for each patch leads to a bounded global error and enables intuitive global control. A more thorough analysis of the error behavior at the patch boundary is a main focus of future research.

Model	Head	St.Matthew	David
#vertices	460,800	3,382,866	4,128,614
#patches	256	596	2,966
Computation time (sec.)			
Clustering	1.7	13.5	17.2
Patch Merging	4.8	68.7	61.4
SDA	4.1	32.1	46.3
DFT	0.3	2.9	3.4
Total Preprocess	**10.9**	**117.2**	**128.3**
Spectral Analysis	<0.1	0.2	0.2
Inverse DFT	0.3	2.9	3.4
Reconstruction			
full model	4.6	32.7	57.7
subsampled to 10%	(1.2)	(10.4)	(15.1)
Total	**15.8 (12.4)**	**153 (130.7)**	**189.6 (147)**

Table 1: Timings for the different stages of the processing pipeline (cf. Figure 1) on a 1.1GHz AMD Athlon with 1.5 GByte main memory.

Performance. Table 1 shows some timing data for our spectral processing pipeline. Note that the bulk of the computation time is spent in the preprocessing stage. Due to its scalar representation, our surface description (comprising SDA and blending grids and parameter mapping) requires less than 40% of the memory of the input model (points and normals) even though no specific compression scheme is applied.

Robustness. An important issue deserving discussion is the effect of a specific patch layout on the final reconstructed surface. Naturally, we want the spectral processing to be invariant under different patchings. While our patching scheme is robust against moderate parameter variations, drastic modifications consequently lead to differences in patch size and shape. Nevertheless, for all examples shown in this paper, we found no perceivable difference when experimentally applying different patch layouts. Of course, if filtering becomes excessive this no longer holds true. If all spectral coefficients are set to zero, for instance, then the patch surfaces will degenerate to the base planes, clearly exhibiting a dependence on the patch layout and the blending function.

Texture and Scalar Attributes. In addition to the geometric information, our pipeline allows to process any attribute data associated with the sample points, such as color or reflectance properties. By including appropriate terms in Equation 1, these attributes could also be used to control the patch layout scheme.

9 Conclusions & Future Work

We have introduced a spectral processing pipeline that extends standard Fourier techniques to general point-sampled geometry. Our framework supports sophisticated surface filtering and Fourier-based resampling, is very efficient in both memory and computation time and thus allows processing of very large geometric models.

Directions for future research include: global error analysis, out-of-core implementation of the processing pipeline, geometry compression, feature detection and extraction, and editing and animation.

Acknowledgements. Our thanks to Marc Levoy and the Digital Michelangelo Project people for providing the data sets of the David and St. Matthew statues. Also many thanks to Simon Rusinkiewicz for making Qsplat publicly available.

References

[1] Amenta, N., Bern, M., Kamvysselis, M. A New Voronoi-Based Surface Reconstruction Algorithm. SIGGRAPH 98
[2] Bracewell, R.N. The Fourier Transform and Its Applications. McGraw-Hill, New York, 2nd rev. ed., 1986
[3] Gärtner, B. Fast and Robust Smallest Enclosing Balls. Proc. 7th Annual European Symposium on Algorithms (ESA), Lecture Notes in Computer Science 1643, Springer-Verlag, 1999
[4] Desbrun, M., Meyer, M. Schröder, P., Barr, A.H. Implicit Fairing of Irregular Meshes Using Diffusion and Curvature Flow. SIGGRAPH 99
[5] Dudgeon, D.E., Mersereau, R.M. Multidimensional Digital Signal Processing, Prentice-Hall, 1984
[6] Gonzalez, R.C., Woods, R.E. Digital Image Processing. Addision-Wesley, 1993
[7] Gortler, S.J., Grzeszczuk, R., Szeliski, R., Cohen, M.F. The Lumigrap. SIGGRAPH 96
[8] Guskov, I., Sweldens, W., Schröder, P. Multiresolution Signal Processing for Meshes. SIGGRAPH 99
[9] Guskov, I., Vidimce, K., Sweldens, W., Schröder, P. Normal Meshes. SIGGRAPH 00
[10] Jain, A.K. Fundamentals of Digital Image Processing, Prentice Hall, 1989
[11] Karni, Z. Gotsman, C. Spectral Compression of Mesh Geometry. SIGGRAPH 00
[12] Kobbelt, L. Discrete Fairing. Proc. of the 7th IMA Conference on the Mathematics of Surfaces '97, 1997
[13] Lee, A., Moreton, H., Hoppe, H. Displaced Subdivision Surfaces. SIGGRAPH 00
[14] 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. SIGGRAPH 00
[15] Mitchell, D.P. Generating antialiased images at low sampling densities. SIGGRAPH 87
[16] Papoulis, A. Signal Analysis, McGraw Hill, 1977
[17] Pfister,H., Zwicker, M., van Baar, J., Gross, M. Surfels: Surface Elements as Rendering Primitives. SIGGRAPH 00
[18] Rusinkiewicz, S., Levoy, M. QSplat: A Multiresolution Point Rendering System for Large Meshes. SIGGRAPH 00
[19] Taubin, G. A Signal Processing Approach to Fair Surface Design. SIGGRAPH 95

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

Figure 13: Restoration of a blurred and noisy surface model (a), filtered with a Gaussian (b) and a feature-preserving Wiener filter (c). The underlying patch layout is shown in image (d).

(a) (b) (c)

Figure 14: The subsampled head (a) and the zooms of the belly button (b, original) and (c, subsampled) clearly show the nonuniform sampling distributions with more samples concentrated at regions of high curvature.

Figure 15: Local smoothing (red circle) and enhancement (blue circle) with adaptive patch layout.

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

Figure 16: Michelangelo's David. Qsplat [18] renderings of the original model (a) (4,128,614 vertices) and the subsampled model (b) (287,165 vertices). Image (c) shows the sampling distribution of the latter, while image (d) illustrates the patch layout.

Adaptively Sampled Distance Fields (ADFs)

Representing Shape for Computer Graphics

Sarah F. Frisken and Ronald N. Perry
Mitsubishi Electric Research Laboratories

Outline

- Overview of ADFs
  - definition
  - advantages
  - instantiations
- Algorithms for octree-based ADFs
  - specifics of octree-based ADFs
  - generating, rendering, and triangulating ADFs
- Applications
  - sculpting, scanning, meshing, modeling, machining ...

Distance Fields

- A distance field is a scalar field that
  - specifies the distance to a shape ...
  - where the distance may be signed to distinguish between the inside and outside of the shape
- Distance
  - can be defined very generally (e.g., non-Euclidean)
  - minimum Euclidean distance is used for most of this presentation (with the exception of the volumetric molecules)

Distance Fields

2D shape with sampled distances to the surface

Regularly sampled distance values

2D distance field

2D Distance Field

R shape

Distance field of R

2D Distance Field

3D visualization of distance field of R

Shape

- By *shape* we mean more than just the 3D geometry of physical objects. Shape can have arbitrary dimension and be derived from simulated or measured data.

Color printer

Color gamut

Conceptual Advantages of Distance Fields

- Represent more than the surface
  - object interior and the space in which the object sits
- Gains in efficiency and quality because
  - distance fields vary "smoothly"
  - are defined throughout space
- Gradient of the distance field yields
  - surface normal for points on the surface
  - direction to closest surface point for points off the surface

Practical Advantages of Distance Fields

- Smooth surface reconstruction
  - continuous reconstruction of a smooth field
- Trivial inside/outside and proximity testing
  - using sign and magnitude of the distance field
- Fast and simple Boolean operations
  - intersection: $\text{dist}(A \cap B) = \min(\text{dist}(A), \text{dist}(B))$
  - union: $\text{dist}(A \cup B) = \max(\text{dist}(A), \text{dist}(B))$
- Fast and simple surface offsetting
  - offset by d: $\text{dist}(A_{offset}) = \text{dist}(A) + d$
- Enables geometric queries such as closest point
  - using gradient and magnitude of the distance field

Sampled Distance Fields

- Similar to sampled images, insufficient sampling of distance fields results in aliasing
- Because fine detail requires dense sampling, excessive memory is required with *regularly* sampled distance fields when *any* fine detail is present

Adaptively Sampled Distance Fields

- Detail-directed sampling
  - high sampling rates only where needed
- Spatial data structure
  - fast localization for efficient processing
- ADFs consist of
  - adaptively sampled distance values …
  - organized in a spatial data structure …
  - with a method for reconstructing the distance field from the sampled distance values

ADF Instantiations

- Spatial data structures
  - octrees
  - wavelets
  - multi-resolution tetrahedral meshes …
- Reconstruction functions
  - trilinear interpolation
  - B-spline wavelet synthesis
  - barycentric interpolation ...

Quadtree

2D Spatial Data Structures – An Example

Wavelets

2D Spatial Data Structures – An Example

Multi-resolution Triangulation

2D Spatial Data Structures – An Example

A Gallery of Examples – A Carved Vase

Illustrates smooth surface reconstruction, fine carving, and representation of algebraic complexity

A Gallery of Examples – A Carved Slab

Illustrates sharp corners and precise cuts

A Gallery of Examples – A Volume Rendered Molecule

Illustrates volume rendering of ADFs, semi-transparency, thick surfaces, and distance-based turbulence

A Gallery of Examples – A 2D Crescent

ADFs provide:

- spatial hierarchy
- distance field
- object surface
- object interior
- object exterior
- surface normal (gradient at surface)
- direction to closest surface point (gradient off surface)

ADFs consolidate the data needed to represent complex objects

ADFs - A Unifying Representation

- Represent surfaces, volumes, and implicit functions
- Represent sharp edges, organic surfaces, thin-membranes, and semi-transparent substances
- Consolidate multiple structures for complex objects (e.g., for collision detection, LOD construction, and dynamic meshing)
- Can store auxiliary data in cells or at cell vertices (e.g., color and texture)

Algorithms for Octree-based ADFs

- Specifics of octree-based ADFs
- Generating ADFs
- Rendering ADFs
- Triangulating ADFs

Octree-based ADFs

- A distance value is stored for each cell corner in the octree
- Distances and gradients are estimated from the stored values using trilinear reconstruction

Reconstruction

A single trilinear field can represent highly curved surfaces

Comparison of 3-color Quadtrees and ADFs

23,573 cells (3-color)

1713 cells (ADF)

Bottom-up Generation

Fully populate Recursively coalesce

Top-down Generation

Initialize root cell Recursively subdivide

Tiled Generation

- Reduced memory requirements
- Better memory coherency
- Reduced computation

```
tiledGeneration(genParams, distanceFunc)
  // cells: block of storage for cells
  // dists: block of storage for final distance values
  // tileVol: temporary volume for computed and
  // reconstructed distance values
  // bitFlagVol: volume of bit flags to indicate
  // validity of distance values in tileVol
  // cell: current candidate for tiled subdivision
  // tileDepth: L (requires (2^(L+1))^3 volume - the L+1
   // level is used to compute cell errors for level L)
  // maxADFLevel: preset max level of ADF (e.g., 12)

  maxLevel = tileDepth
  cell = getNextCell(cells)
  initializeCell(cell, NULL) (i.e., root cell)

  while (cell)
    setAllBitFlagVolInvalid(bitFlagVol)
    if (cell.level == maxLevel)
      maxLevel = min(maxADFLevel, maxLevel + tileDepth)
    recurSubdivToMaxLevel(cell,maxLevel,maxADFLevel)
    addValidDistsToDistsArray(tileVol, dists)
    cell = getNextCandidateForSubdiv(cells)
```

```
initializeCell(cell, parent)
  initCellFields(cell, parent, bbox, level)
  for (error = 0, pt = cell, face, and edge centers)
    if (isBitFlagVolValidAtPt(pt))
      comp = getTileComputedDistAtPt(pt)
      recon = getTileReconstructedDistAtPt(pt)
    else
      comp = computeDistAtPt(pt)
      recon = reconstructDistAtPt(cell, pt)
      setBitFlagVolValidAtPt(pt)
    error = max(error, abs(comp - recon))
  setCellError(error)
```

```
recurSubdivToMaxLevel(cell, maxLevel, maxADFLevel)
  // Trivially exclude INTERIOR and EXTERIOR cells
  // from further subdivision
  pt = getCellCenter(cell)
  if (abs(getTileComputedDistAtPt(pt)) >
      getCellHalfDiagonal(cell))
    // cell.type is INTERIOR or EXTERIOR
    setCellTypeFromCellDistValues(cell)
    return

  // Stop subdividing when error criterion is met
  if (cell.error < maxError)
    // cell.type is INTERIOR, EXTERIOR, or BOUNDARY
    setCellTypeFromCellDistValues(cell)
    return

  // Stop subdividing when maxLevel is reached
  if (cell.level >= maxLevel)
    // cell.type is INTERIOR, EXTERIOR, or BOUNDARY
    setCellTypeFromCellDistValues(cell)
    if (cell.level < maxADFLevel)
      // Tag cell as candidate for next layer
      setCandidateForSubdiv(cell)
    return

  // Recursively subdivide all children
  for (each of the cell's 8 children)
    child = getNextCell(cells)
    initializeCell(child, cell)
    recurSubdivToMaxLevel(child, maxLevel, maxADFLevel)

  // cell.type is INTERIOR, EXTERIOR, or BOUNDARY
  setCellTypeFromChildrenCellTypes(cell)

  // Coalesce INTERIOR and EXTERIOR cells
  if (cell.type != BOUNDARY) coalesceCell(cell)
```

Tiled Generation Pseudocode

Tiled Generation – Overview

- Recursively subdivide root cell to a level *L*
- Cells at level *L* requiring further subdivision are appended to a list of candidate cells, *C-list*
- These candidate cells are recursively subdivided between levels *L* and *2L*, where new candidate cells are produced and appended to *C-list*
- Repeat layered production of candidate cells (*2L* to *3L*, etc.) until *C-list* is empty

Tiled Generation – Candidate Cells

- A cell becomes a candidate for further subdivision when all of the following are true:
  - it is a leaf cell of level *L*, or *2L*, or *3L*, etc.
  - it can not be trivially determined to be an interior or exterior cell
  - it does not satisfy a specified error criterion
  - its level is below a specified maximum ADF level

Tests for Candidate Cells

Test to trivially determine if a cell is interior or exterior

19 test points to determine cell error

Tiled Generation – Tiling

- For each candidate cell, computed and reconstructed distances are produced *only* as needed during subdivision
- These distances are stored in a ***tile***, a regularly sampled volume
- The tile resides in cache memory and its size determines *L*
- A volume of bit flags keeps track of valid distances in the tile to ensure that distances are computed only once

Tiled Generation – Tiling

- For coherency, cells and final distances are stored in two separate contiguous memory blocks
- After a candidate cell has been processed, valid distances in the tile are appended to the block of final distances
- Special care is taken at tile boundaries to ensure that distances are *never* duplicated for neighboring cells

Tiled Generation – Cache Efficiency

- Tile sizes can be tuned to the CPU cache architecture
- For current Pentium systems, a tile size of 16^3 has worked most effectively
- Using a separate bit flag volume further enhances cache effectiveness and provides fast invalidation of tile distances prior to processing each candidate cell

Rendering

- Ray casting
- Adaptive ray casting
- Point-based rendering
- Triangles

Ray Casting

Ray-surface Intersection with a Cubic Solver

- See Parker *et al.*, "Interactive Ray Tracing for Volume Visualization"

Ray Casting

Ray-surface Intersection with a Linear Solver

- Assume that distances vary linearly along the ray
- Determine the zero-crossing within the cell given distances at the points where the ray enters and exits the cell

$$\hat{P}_0 = P_{in} \cdot (1-t) + P_{out}\, t$$

Ray Casting

Crackless Surface Rendering with the Linear Solver

- Set the distance at the entry point of a cell equal to the distance computed for the exit point of the previous cell

$$d_{out}(A) \stackrel{?}{=} d_{in}(B)$$

Ray Casting

Volume Rendering

- Colors and opacities are accumulated at equally spaced samples along each ray

Adaptive Ray Casting

- The image region to be rendered is divided into a hierarchy of image tiles
- The subdivision of each tile is guided by a perceptually-based predicate
- Pixels within image tiles of size greater than 1x1 are bilinearly interpolated to produce the image
- Rays are cast into the ADF at tile corners and intersected with the surface using the linear solver

Adaptive Ray Casting

- The predicate individually weights the contrast in the red, green, and blue channels and the variance in depth-from-camera across the tile
  - See Mitchell , SIGGRAPH'87, and Bolin and Meyer, SIGGRAPH'98
- Results in a typical 6:1 reduction in rendering time over non-adaptive ray casting

Adaptive Ray Casting

Adaptively ray cast ADF

Rays cast to render part of the left image

Point-based Rendering

- Determine the number of points to generate in each boundary leaf cell
  - Compute an estimate of the object's surface area within each boundary leaf cell *areaCell* and the total estimated surface area of the object,
    areaObject = Σ *areaCell*
  - Set the number of points in each cell *nPtsCell* proportional to *areaCell / areaObject*

- For each boundary leaf cell in the ADF
  - Generate *nPtsCell* random points in the cell
  - Move each point to the object's surface using the distance and gradient at the point

Point-based Rendering
Pseudocode

```
generatePoints(adf, points, nPts, maxPtsToGen)
  // Estimate object's surface area within each boundary leaf
  // cell and the total object's surface area
  for (areaObject = 0, level = 0 to maxADFLevel)
    nCellsAtLevel = getNumBoundaryLeafCellsAtLevel(adf, level)
    areaCell[level] = sqr(cellSize(level))
    areaObject += nCellsAtLevel * areaCell[level]

  // nPtsCell is proportional to areaCell / areaObject
  for (level = 0 to maxADFLevel)
    nPtsAtLevel[level] = maxPtsToGen * areaCell[level] / areaObject

  // For each boundary leaf cell, generate cell points
  // and move each point to the surface
  for (nPts = 0, cell = each boundary leaf cell of adf)
    nPtsCell = nPtsAtLevel[cell.level]
    while (nPtsCell--)
      pt = generateRandomPositionInCell(cell)
      d = reconstructDistAtPt(cell, pt)
      n = reconstructNormalizedGradtAtPt(cell, pt)
      pt += d * n
      n = reconstructNormalizedGradtAtPt(cell, pt)
      setPointAttributes(pt, n, points, nPts++)
```

Point-based Rendering

An ADF rendered as points at two different scales

Triangle Rendering

- ADFs can also be rendered by triangulating the surface and using graphics hardware to rasterize the triangles
- Triangulation is fast
  - 200,000 triangles in 0.37 seconds, Pentium IV
  - 2,000 triangles in < 0.01 seconds
- The triangulation produces models that are orientable and closed

Triangulation

- ***Seed*** – Each boundary leaf cell of the ADF is assigned a vertex that is initially placed at the cell's center
- ***Join*** – Vertices of neighboring cells are joined to form triangles
- ***Relax*** – Vertices are moved to the surface using the distance field
- ***Improve*** – Vertices are moved over the surface towards their average neighbors' position to improve triangle quality

Triangulation

- Vertices are ***joined*** to form triangles using the following observations
  - A triangle joins the vertices of 3 neighboring cells that share a common edge (hence triangles are associated with cell edges)
  - A triangle is associated with an edge only if that edge has a zero crossing of the distance field
  - The orientation of the triangle can be derived from the orientation of the edge it crosses
  - In order to avoid making redundant triangles, we consider 6 of the 12 possible edges for each cell

Triangulation – Surface Cracks

Most triangulation algorithms for adaptive grids suffer from this type of crack; our algorithm ***does not***

As with other algorithms, this type of crack occurs ***very rarely*** but we can prevent it with a simple pre-conditioning step

Triangulation – Pre-conditioning

- In 3D, the pre-conditioning step compares the number of zero-crossings of the iso-surface for each face of each boundary leaf cell to the total number of zero-crossings for faces of the cell's face-adjacent neighbors that are shared with the cell
- When the number of zero-crossings are not equal for any face, the cell is subdivided using distance values from its face-adjacent neighbors until the number of zero-crossings match

```
triangulateADF(adf)
  // vertices: storage for vertices
  // triangles: storage for triangles

  // Initialize triangles vertices at cell centers
  // and associate each vertex with its cell

  for (cell = each boundary leaf cell of adf)
    v = getNextVertex(vertices)
    associateVertexWithCell(cell, v)
    v.position = getCellCenter(cell)

  // Make triangles. Each cell edge joins two cell
  // faces face1 and face2 which are ordered to ensure
  // a consistent triangle orientation (see EdgeFace
  // table below). For a given cell edge and face,
  // getFaceNeighborVertex returns either the vertex of
  // the cell's face-adjacent neighbor if the
  // face-adjacent neighbor is the same size or larger
  // than the cell, OR, the vertex of the unique child
  // cell (uniqueness is guaranteed by a
  // pre-conditioning step) of the face-adjacent
  // neighbor that is both adjacent to the face and
  // has a zero-crossing on the edge

  for (cell = each boundary leaf cell of adf)
    for (edge = cell's up-right, down-left, up-front,
    down-back, front-right, and back-left edges)
      if (surfaceCrossesEdge(edge))
        face1 = EdgeFace[edge].face1
        face2 = EdgeFace[edge].face2
        v0 = getCellsAssociatedVertex(cell)
        v1 = getFaceNeighborVertex(face1, edge)
        v2 = getFaceNeighborVertex(face2, edge)
        t = getNextTriangle(triangles)
        if (edgeOrientation(edge) > 0)
          t.v0 = v0, t.v1 = v1, t.v2 = v2
        else
          t.v0 = v0, t.v1 = v2, t.v2 = v1

  // Relax each vertex to the surface and then along
  // the tangent plane at the relaxed position towards
  // the average neighbor position

  for (each vertex)
    v = getVertexPosition(vertex)
    u = getAveragePositionOfNeighborVertices(vertex)
    cell = getVertexCell(vertex)
    d = reconstructDistAtPt(cell, v)
    n = reconstructNormalizedGradtAtPt(cell, v)
    v += d * n
    v += (u - v) - n · (u - v)


// ----------------------------------------
// EdgeFace table:
//   edge                face1        face2
//   ----------          -----        -----
//   up-right            up           right
//   down-left           down         left
//   up-front            up           front
//   down-back           down         back
//   front-right         front        right
//   back-left           back         left
```

Triangulation – Level-of-Detail

- The octree is traversed and vertices are ***seeded*** into boundary cells whose maximum error satisfies a user-specified threshold
- Cells below these cells in the hierarchy are ignored
- The error threshold can be varied continuously enabling fine control over the number of triangles generated
- Time to produce an LOD model is proportional to the number of vertices in the output mesh

Triangulation – Level-of-Detail

Applications

- Sculpting
- 3D scanning
- Dynamic meshing
- Physically-based modeling
- Color management
- Volumetric effects
- Machining

Sculpting

"Kizamu: A System for Sculpting Digital Characters"

- ADFs can represent both smooth surfaces and sharp corners without excessive memory
- Carving is direct, intuitive, and fast
- Does not require control point manipulation or trimming
- The distance field can be used to position and orient the sculpting tool or to constrain carving

3D Scanning

- Use of distance fields provides more robust, water-tight surfaces
- ADFs result in significant savings in memory and distance computations
- Resultant models can be directly sculpted to correct the scanned data
- Fast new triangulation method produces optimal triangle meshes from the ADF

Dynamic Meshing

Level-of-Detail and View Dependent Triangulation

- ADF octree provides hierarchical structure for generating LOD models
- View-dependent meshing uses ADF hierarchy, cell size, and cell gradients
- ADF cell error enables fine control over triangle count in LOD meshes
- Real-time ADF triangulation algorithm produces meshes that are orientable and closed

Physically-based Modeling

- ADFs provide a compact representation of complex surfaces
- ADF spatial hierarchy and trivial inside/outside tests enable fast collision detection
- Distance field provides penetration depths for computing impact forces
- Distance field allows computation of material-dependent contact deformation

Color Management

Representing Color Gamuts

- ADF distance field enables a fast, simple out-of-gamut test
- ADFs provide a compact representation of complex gamut shapes
- Gamut test is very accurate near the gamut surface
- Distance and gradient indicate how far out of gamut a color lies and the direction to the nearest in-gamut color

Volumetric Effects

- Offset surfaces can be used to render thick, translucent surfaces
- Volume texture can be added within the thick surface
- Distance values away from the surface can be used for special effects (e.g., turbulent haze)
- Octree and distance field allow space-leaping and other methods to speed up volume rendering

Machining

- ADFs represent surfaces, object interiors, and the material to be removed
- ADFs represent smooth surfaces and very fine detail
- Trivial inside/outside and proximity tests are useful for designing tool paths
- Gradients can be used to select tool orientation
- Offset surfaces can be used for rough cutting in coarse-to-fine milling

For More Information

At Siggraph 2001

- Paper presentation:
  - *"Kizamu: A System for Sculpting Digital Characters",* Wednesday, 15 August, 10:30 am
- Sketches:
  - *"Dynamic Meshing Using Adaptively Sampled Distance Fields",* Wednesday, 15 August, 4:30 pm
  - *"A Computationally Efficient Framework for Modeling Soft Body Impact",* Thursday, 18 August, 8:30 am
  - *"Computing 3D Geometry Directly from Range Images",* Friday, 17 August, 2:20 pm

For More Information

In Your Course Notes

- A nearly final version of the Kizamu paper, SIGGRAPH 2001 and MERL Technical Report TR2001-08
- *"A New Representation for Device Color Gamuts",* MERL Technical Report TR2001-09
- *"Computing 3D Geometry Directly from Range Images",* SIGGRAPH 2001 Technical Sketch and MERL Technical Report TR2001-10
- *"A Computationally Efficient Framework for Modeling Soft Body Impact",* SIGGRAPH 2001 Technical Sketch and MERL Technical Report TR2001-11
- *"A New Framework For Non-Photorealistic Rendering",* MERL Technical Report TR2001-12
- *"Dynamic Meshing Using Adaptively Sampled Distance Fields",* SIGGRAPH 2001 Technical Sketch and MERL Technical Report TR2001-13
- *"Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics",* SIGGRAPH 2000 and MERL Technical Report TR2000-15
- *"Using Distance Maps for Accurate Surface Representation in Sampled Volumes",* IEEE VolVis Symp. 1998 and MERL Technical Report TR99-25

The End

Image-Based Modeling and Rendering:

Explicit or Implicit Shape?

Leonard McMillan
LCS Computer Graphics Group
Massachusetts Institute of Technology

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Goals of IBR

To infer new images from old ones...

Different Schools of Thought

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Images with structure

- Prerendering analysis
- Small runtime footprint
- Active image sensors
- Backward compatible

Images without structure

- Can use raw images
- Minimal analysis and assumptions
- Large storage requirements
- Incompatible/Radical

IBR Principles

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Consider images as a collection of rays,

... rather than a collection of pixels.

The Plenoptic Function

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Given enough sample rays, can we interpolate nearby ones?

IBR is a different approach to computer graphics.

Where to Begin?

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Next Generation Cameras

- Known Internal calibration
- Know where they are
- Photometric
- High-dynamic range

Image Courtesy of MIT City Scanning Project, Seth Teller, Satyan Coorg, JP Mellor, George Chou, Doug De Couto, Neel Master, Barb Cutler, Eric Amram, Mike Bosse, Matt Antone, Stefano Totaro, and Manish Jethwa.

Warping Points with Depth

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Advantages

- Simple warping transformation

$$\vec{x}_2 = \frac{1}{z_1}\vec{e}_2 + H_{21}\vec{x}_1$$

- Small footprint in memory
- Occlusion compatible rendering order
- Compatible with traditional graphics methods

Disadvantages

- View-independent shading
- Reconstruction errors
- Disocclusion

Approximate Geometry

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Many images with an approximate model

- Façade
- View-dependent texture mapping
- Image-based visual hulls

Approximate shape for inter-object interactions

- Occlusions
- Walk-around

Textures for detail

- View-dependent shading
- Small geometric features

Acquiring Depth Images

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Economical laser scanners

Images and video courtesy of the University of North Carolina at Chapel Hill, "Office of the Future" Project, Wei-Chao Chen, Henry Fuchs, Lars Nyland, Herman Towles and Greg Welch.

Warping Prospects

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Modeling

- Active sensing
- Passive sensing
- Dynamic scenes

Rendering

- View-dependent shading
- Disocclusion (LDIs, MCOPs)
- Reconstruction

Hardware Acceleration

What is a Visual Hull?

Why use a Visual Hull?

They rely on the simplest CV algorithms

They can be computed robustly

They can be computed efficiently

Acquisition

Several cameras with overlapping views

- Geometric & Photometric calibration
- Synchronization

Image-based Visual Hulls

Volume-like

Self-consistent

Discrete-continuous

Image-Based Computation

Observation

Incremental computation along scanlines

Step 1: Binning

Sort silhouette edges into bins

Step 2: Scanning

IBVH Advantages

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

- Approximately constant computation per pixel per camera
- Parallelizes
- Consistent with input silhouettes

Shading Algorithm

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Use view dependent texture mapping at each IBVH sample

Problem: VISIBILITY

IBVH Visibility Algorithm

Determining Visibility in 2D

Shading Visual Hulls

View-dependent illumination

Visibility

IBVH Results

An IBVH-Based 3D Scanner

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Simultaneous capture of IBVH shape and reflected radiance

Low-cost

Fast acquisition

Active Back Lighting

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Provides improved segmentation

Actual System

IBVH Scans

Image-based visual hulls built from 108 (3 x 36) images

Dealing with Concavities

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Concave surface regions never appear on a silhouette. Thus, an IBVH can not capture such shapes...

View Dependent Shading

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

However, the captured images can be used as a surface light field defined over the visual hull. Thus, providing accurate renderings despite the geometric inaccuracies.

IBVH Object Models

We have captured 100's of models with our IBVH-based 3D scanning system. Including, highly specular, fuzzy, and translucent objects.

How many Images?

The shape estimate of the visual hull converges rapidly. Subsequent silhouettes provide only minor improvements.

How many Images?

Adding more images dramatically improves the rendering quality of highly specular and transparent surfaces, as well as improving the rendering of concavities.

Visual Hull Prospects

Capture

- Passive - multiple synchronized calibrated cameras
- Limited working volume
- Approximate model (even in the limit)

Rendering

- Fast - can be computed at a constant cost/pixel
- Limited view-dependent shading
- Texture registration on approximate geometry

Acceleration

- Scalable to a large number of cameras

Structured Light fields

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Focal planes as approximate structure

Interpolating Rays

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Less structure / More images

Structure "on-the-fly"

Dynamic Reparameterization

Light Field Acquisition

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Motion Platforms

- Precise positioning
- Calibrated digital camera
- Expensive (> $10K)
- Very Slow (~20 mins)

Light field cameras

- Less precise
- Calibration per aperture
- Inexpensive (~ $100)
- Slow (> 3 mins)

Focal Planes as Structure

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Structure is discovered by user interaction rather than recovered by computer vision

Intuitive "camera-like" interface

Reconstruction (Interpolation) controlled by variable focus and variable aperture

Dynamic Reparameterization

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Light fields with variable focal planes and apertures

Prospects of Light Fields

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Modeling

- Many images/cameras (expensive setup)
- Minimal assumptions about the scene
- Compression = Structure?

Rendering

- Supports view-dependent shading
- Very fast - large memory requirements
- Less structure - more aliasing

Hardware Acceleration

- Can use existing texture-mapping H/W

Unstructured Light Fields

Reference images along unconstrained camera paths

Acquiring Unstructured LFs

Can use a wide range of source images

- Images from regular arrays or panoramas
- Tracked or calibrated cameras
- Hand-held camcorder

Position/Pose recovery

- Post-process source images by tracking features
- Photometric equalization

Rendering Unstructured LFs

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

View-dependent parameterization via dynamic triangulation

H/W texture mapping

Results

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Static scenes

Dynamic scenes

Unstructured LF Prospects

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Modeling

- Wide range of source materials
- Camera tracking is essential

Rendering

- Supports view-dependent shading
- Very very fast - large memory requirements
- Many artifacts to overcome (scattered data reconstruction)

Hardware Acceleration

- Can use existing texture-mapping H/W

Conclusions

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

- Representation Trade-offs
  - Explicit (Model's + Texture)
  - Implicit (Images alone)
- How much Shape?
  - As much as we can get?
  - As much as we can get by with?

Image-Based Visual Hulls

Wojciech Matusik*
Laboratory for Computer Science
Massachusetts Institute of Technology

Chris Buehler*
Laboratory for Computer Science
Massachusetts Institute of Technology

Ramesh Raskar‡
Department of Computer Science
University of North Carolina - Chapel Hill

Steven J. Gortler†
Division of Engineering and Applied Sciences
Harvard University

Leonard McMillan*
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

In this paper, we describe an efficient image-based approach to computing and shading visual hulls from silhouette image data. Our algorithm takes advantage of epipolar geometry and incremental computation to achieve a constant rendering cost per rendered pixel. It does not suffer from the computation complexity, limited resolution, or quantization artifacts of previous volumetric approaches. We demonstrate the use of this algorithm in a real-time virtualized reality application running off a small number of video streams.

Keywords: Computer Vision, Image-Based Rendering, Constructive Solid Geometry, Misc. Rendering Algorithms.

1 Introduction

Visualizing and navigating within virtual environments composed of both real and synthetic objects has been a long-standing goal of computer graphics. The term "Virtualized Reality™", as popularized by Kanade [23], describes a setting where a real-world scene is "captured" by a collection of cameras and then viewed through a virtual camera, as if the scene was a synthetic computer graphics environment. In practice, this goal has been difficult to achieve. Previous attempts have employed a wide range of computer vision algorithms to extract an explicit geometric model of the desired scene.

Unfortunately, many computer vision algorithms (e.g. stereo vision, optical flow, and shape from shading) are too slow for real-time use. Consequently, most virtualized reality systems employ off-line post-processing of acquired video sequences. Furthermore, many computer vision algorithms make unrealistic simplifying assumptions (e.g. all surfaces are diffuse) or impose impractical restrictions (e.g. objects must have sufficient non-periodic textures) for robust operation. We present a new algorithm for synthesizing virtual renderings of real-world scenes in real time. Not only is our technique fast, it also makes few simplifying assumptions and has few restrictions.

*(wojciech | cbuehler | mcmillan)@graphics.lcs.mit.edu
†sjg@cs.harvard.edu
‡raskar@cs.unc.edu

Figure 1 - The intersection of silhouette cones defines an approximate geometric representation of an object called the visual hull. A visual hull has several desirable properties: it contains the actual object, and it has consistent silhouettes.

Our algorithm is based on an approximate geometric representation of the depicted scene known as the visual hull (see Figure 1). A visual hull is constructed by using the visible silhouette information from a series of reference images to determine a conservative shell that progressively encloses the actual object. Based on the principle of *calculatus eliminatus* [28], the visual hull in some sense carves away regions of space where the object "is not".

The visual hull representation can be constructed by a series of 3D constructive solid geometry (CSG) intersections. Previous robust implementations of this algorithm have used fully enumerated volumetric representations or octrees. These methods typically have large memory requirements and thus, tend to be restricted to low-resolution representations.

In this paper, we show that one can efficiently render the exact visual hull without constructing an auxiliary geometric or volumetric representation. The algorithm we describe is "image based" in that all steps of the rendering process are computed in "image space" coordinates of the reference images.

We also use the reference images as textures when shading the visual hull. To determine reference images that can be used, we compute which reference cameras have an unoccluded view of each point on the visual hull. We present an image-based visibility algorithm based on epipolar geometry and McMillan's occlusion compatible ordering [18] that allows us to shade the visual hull in roughly constant time per output pixel.

Using our *image-based visual hull* (IBVH) algorithm, we have created a system that processes live video streams and renders the observed scene from a virtual camera's viewpoint in real time. The resulting representation can also be combined with traditional computer graphics objects.

2 Background and Previous Work

Kanade's virtualized reality system [20] [23] [13] is perhaps closest in spirit to the rendering system that we envision. Their initial implementations have used a collection of cameras in conjunction with multi-baseline stereo techniques to extract models of dynamic scenes. These methods require significant off-line processing, but they are exploring special-purpose hardware for this task. Recently, they have begun exploring volume-carving methods, which are closer to the approach that we use [26] [30].

Pollard's and Hayes' [21] immersive video objects allow rendering of real-time scenes by morphing live video streams to simulate three-dimensional camera motion. Their representation also uses silhouettes, but in a different manner. They match silhouette edges across pairs of views, and use these correspondences to compute morphs to novel views. This approach has some limitations, since silhouette edges are generally not consistent between views.

Visual Hull. Many researchers have used silhouette information to distinguish regions of 3D space where an object is and is not present [22] [8] [19]. The ultimate result of this carving is a shape called the object's *visual hull* [14]. A visual hull always contains the object. Moreover, it is an equal or tighter fit than the object's convex hull. Our algorithm computes a view-dependent, sampled version of an object's visual hull each rendered frame.

Suppose that some original 3D object is viewed from a set of reference views R. Each reference view r has the silhouette s_r with interior pixels covered by the object. For view r one creates the cone-like volume vh_r defined by all the rays starting at the image's point of view p_r and passing through these interior points on its image plane. It is guaranteed that the actual object must be contained in vh_r. This statement is true for all r; thus, the object must be contained in the volume $vh_R=\bigcap_{r\in R} vh_r$. As the size of R goes to infinity, and includes all possible views, vh_R converges to an approximate shape known as the visual hull vh_∞ of the original geometry. The visual hull is not guaranteed to be the same as the original object since concave surface regions can never be distinguished using silhouette information alone.

In practice, one must construct approximate visual hulls using only a finite number of views. Given the set of views R, the approximation vh_R is the best conservative geometric description that one can achieve based on silhouette information alone (see Figure 1). If a conservative estimate is not required, then alternative representations are achievable by fitting higher order surface approximations to the observed data [2].

Volume Carving. Computing high-resolution visual hulls can be a tricky matter. The intersection of the volumes vh_r requires some form of CSG. If the silhouettes are described with a polygonal mesh, then the CSG can be done using polyhedral CSG, but this is very hard to do in a robust manner.

A more common method used to convert silhouette contours into visual hulls is volume carving [22] [8] [29] [19] [5] [27]. This method removes unoccupied regions from an explicit volumetric representation. All voxels falling outside of the projected silhouette cone of a given view are eliminated from the volume. This process is repeated for each reference image. The resulting volume is a quantized representation of the visual hull according to the given volumetric grid and the reference image set. A major advantage of our view-dependent method is that it minimizes artifacts resulting from this quantization.

CSG Rendering. A number of algorithms have been developed for the fast rendering of CSG models, but most are ill suited for our task. The algorithm described by Rappoport [24], requires that each solid be first decomposed to a union of convex primitives. This decomposition can prove expensive for complicated silhouettes. Similarly, the algorithm described in [11] requires a rendering pass for each layer of depth complexity. Our method does not require preprocessing the silhouette cones. In fact, there is no explicit data structure used to represent the silhouette volumes other than the reference images.

Figure 2 – Computing the IBVH involves three steps. First, the desired ray is projected onto a reference image. Next, the intervals where the projected ray crosses the silhouette are determined. Finally, these intervals are lifted back onto the desired ray where they can be intersected with intervals from other reference images.

Using ray tracing, one can render an object defined by a tree of CSG operations without explicitly computing the resulting solid [25]. This is done by considering each ray independently and computing the interval along the ray occupied by each object. The CSG operations can then be applied in 1D over the sets of intervals. This approach requires computing a 3D ray-solid intersection. In our system, the solids in question are a special class of cone-like shapes with a constant cross section in projection. This special form allows us to compute the equivalent of 3D ray intersections in 2D using the reference images.

Image-Based Rendering. Many different image-based rendering techniques have been proposed in recent years [3] [4] [15] [6] [12]. One advantage of image-based rendering techniques is their stunning realism, which is largely derived from the acquired images they use. However, a common limitation of these methods is an inability to model dynamic scenes. This is mainly due to data acquisition difficulties and preprocessing requirements.

3 Visual-Hull Computation

Our approach to computing the visual hull has two distinct characteristics: it is computed in the image space of the reference images and the resulting representation is viewpoint dependent. The advantage of performing geometric computations in image space is that it eliminates the resampling and quantization artifacts that plague volumetric approaches. We limit our sampling to the pixels of the desired image, resulting in a view-dependent visual-hull representation. In fact, our IBVH representation is equivalent to computing exact 3D silhouette cone intersections and rendering the result with traditional rendering methods.

Our technique for computing the visual hull is analogous to finding CSG intersections using a ray-casting approach [25]. Given a desired view, we compute each viewing ray's intersection with the visual hull. Since computing a visual hull involves only

intersection operations, we can perform the CSG calculations in any order. Furthermore, in the visual hull context, every CSG primitive is a generalized cone (a projective extrusion of a 2D image silhouette). Because the cone has a fixed (scaled) cross section, the 3D ray intersections can be reduced to cheaper 2D ray intersections. As shown in Figure 2 we perform the following steps: 1) We project a 3D viewing ray into a reference image. 2) We perform the 1D intersection of the projected ray with the 2D silhouette. These 1D intersections result in a list of intervals along the ray that are interior to the cone's cross-section. 3) Each 1D interval is then lifted back into 3D using a simple projective mapping, and then intersected with the results of the ray-cone intersections from other reference images. A naïve algorithm for computing these IBVH ray intersections follows:

```
IBVHisect (intervalImage &d, refImList R){
  for each referenceImage r in R
    computeSilhouetteEdges (r)
  for each pixel p in desiredImage d do
    p.intervals =  {0..inf}
  for each referenceImage r in R
    for each scanline s in d
      for each pixel p in s
        ray3D  ry3 = compute3Dray(p,d.camInfo)
        lineSegment2D l2 = project3Dray(ry3,r.camInfo)
        intervals int2D = calcIntervals(l2,r.silEdges)
        intervals int3D = liftIntervals(int2D,r.camInfo,ry3)
        p.intervals = p.intervals ISECT int3D
}
```

To analyze the efficiency of this algorithm, let n be the number of pixels in a scanline. The number of pixels in the image d is $O(n^2)$. Let k be the number of reference images. Then, the above algorithm has an asymptotic running time $O(ikn^2)$, where i is the time complexity of the `calcintervals` routine. If we test for the intersection of each projected ray with each of the e edges of the silhouette, the running time of `calcintervals` is $O(e)$. For large classes of scenes, we can describe the average number of edges on the boundary of a silhouette to be $O(ln)$ where l is the average number of times that a projected ray intersects the silhouette[1]. Thus, the running time of `IBVHisect` to compute all of the 2D intersections for a desired view is $O(lkn^3)$.

The performance of this naïve algorithm can be improved by taking advantage of incremental computations that are enabled by the epipolar geometry relating the reference and desired images. These improvements will allow us to reduce the amortized cost of 1D ray intersections to $O(l)$ per desired pixel, resulting in an implementation of `IBVHisect` that takes $O(lkn^2)$.

Given two camera views, a reference view r and a desired view d, we consider the set of planes that share the line connecting the cameras' centers. These planes are called *epipolar planes*. Each epipolar plane projects to a line in each of the two images, called an *epipolar line*. In each image, all such lines intersect at a common point, called the *epipole*, which is the projection of one of the camera's center onto the other camera's view plane [9].

As a scanline of the desired view is traversed, each pixel projects to an epipolar line segment in r. These line segments emanates from the epipole e_{dr}, the image of d's center of projection onto r's image plane (see Figure 3), and trace out a "pencil" of epipolar lines in r. The slopes of these epipolar line segments will either increase or decrease monotonically depending on the direction of traversal (Green arc in Figure 3). We take advantage of this monotonicity to compute silhouette intersections for the whole scanline incrementally.

[1] We assume reference images also have $O(n^2)$ pixels.

Figure 3 – The pixels of a scanline in the desired image trace out a pencil of line segments in the reference image. An ordered traversal of the scanline will sweep out these segments such that their angle about the epipole varies monotonically.

The silhouette contour of each reference view is represented as a list of edges enclosing the silhouette's boundary pixels. These edges are generated using a 2D variant of the marching cubes approach [16]. Next, we sort the $O(nl)$ contour vertices in increasing order by the slope of the line connecting each vertex to the epipole. These sorted vertex slopes are divided into $O(nl)$ bins. Bin B_i has an angular extent spanning between the slopes of the ith and $i+1$st slope in the sorted list. In each bin B_i we place all edges that are intersected by epipolar lines with an angle falling within the bin's extent. During `IBVHisect` as we traverse the pixels along a scanline in the desired view, the projected corresponding view rays fan across the epipolar pencil in the reference view with increasing slope. Concurrently, we scan through the list of bins. The appropriate bin for each epipolar line is found and it is intersected with the edges in that bin. This procedure is analogous to merging two sorted lists, which can be done in a time proportional to the length of the lists ($O(nl)$ in our case).

For each scanline in the desired image we evaluate n viewing rays. For each viewing ray we compute its intersection with edges in a single bin. Each bin contains on average $O(l)$ silhouette edges. Thus, this step takes O(l) time per ray. Simultaneously we traverse the sorted set of O(nl) bins as we traverse the scanline. Therefore, one scanline is computed in O(nl) time. Over n scanlines in k reference images[2], this gives a running time of $O(lkn^2)$. Pseudocode for the improved algorithm follows.

```
IBVHisect (intervalImage &d, refImList R){
  for each referenceImage r in R
    computeSilhouetteEdges (r)
  for each pixel p in desiredImage d do
    p.intervals =  {0..inf}
  for each referenceImage r in R
    bins b = constructBins(r.caminfo, r.silEdges, d.caminfo)
    for each scanline s in d
      incDec order = traversalOrder(r.caminfo,d.caminfo,s)
      resetBinPositon(b)
      for each pixel p in s according to order
        ray3D  ry3 = compute3Dray(p,d.camInfo)
        lineSegment2D l2 = project3Dray(ry3,r.camInfo)
        slope m = ComputeSlope(l2,r.caminfo,d.caminfo)
        updateBinPosition(b,m)
        intervals int2D = calcIntervals(l2,b.currentbin)
        intervals int3D = liftIntervals(int2D,r.camInfo,ry3)
        p.intervals = p.intervals ISECT int3D
}
```

It is tempting to apply further optimizations to take greater advantage of epipolar constraints. In particular, one might con-

[2] Sorting the contour vertices takes $O(nl \log(nl))$ and binning takes $O(nl^2)$. Sorting and binning over k reference views takes $O(knl \log(nl))$ and $O(knl^2)$ correspondingly. In our setting, $l << n$ so we view the total complexity as $O(lkn^2)$.

sider rectifying each reference image with the desired image prior to the ray-silhouette intersections. This would eliminate the need to sort, bin, and traverse the silhouette edge lists. However, a call to `liftInterval` would still be required for each pixel, giving the same asymptotic performance as the algorithm presented. The disadvantage of rectification is the artifacts introduced by the two resampling stages that it requires. The first resampling is applied to the reference silhouette to map it to the rectified frame. The second is needed to unrectify the computed intervals of the desired view. In the typical stereo case, the artifacts of rectification are minimal because of the closeness of the cameras and the similarity of their pose. But, when computing visual hulls the reference cameras are positioned more freely. In fact, it is not unreasonable for the epipole of a reference camera to fall within the field of view of the desired camera. In such a configuration, rectification is degenerate.

4 Visual-Hull Shading

The IBVH is shaded using the reference images as textures. In order to capture as many view-dependent effects as possible a view-dependent texturing strategy is used. At each pixel, the reference-image textures are ranked from "best" to "worst" according to the angle between the desired viewing ray and rays to each of the reference images from the closest visual hull point along the desired ray. We prefer those reference views with the smallest angle [7]. However, we must avoid texturing surface points with an image whose line-of-sight is blocked by some other point on the visual hull, regardless of how well aligned that view might be to the desired line-of-sight. Therefore, visibility must be considered during the shading process.

When the visibility of an object is determined using its visual hull instead of its actual geometry, the resulting test is conservative– erring on the side of declaring potentially visible points as non-visible. We compute visibility using the visual hull approximation, VH_R, as determined by IBVHisect. This visual hull is represented as intervals along rays of the desired image *d*. Pseudocode for our shading algorithm is given below.

```
IBVHshade(intervalImage &d, refImList R){
  for each pixel p in d do
    p.best = BIGNUM
  for each referenceImage r in R do
    for each pixel p in d do
       ray3D ry3 = compute3Dray(p,d.camInfo)
       point3 pt3 = front(p.intervals,ry3)
       double s  = angleSimilarity(pt3,ry3,r.camInfo)
       if isVisible(pt3,r,d)
         if (s < p.best)
           point2 pt2 = project(pt3,r.camInfo)
           p.color = sample_color(pt2,r)
           p.best = s
}
```

The `front` procedure finds the front most geometric point of the IBVH seen along the ray. The `IBVHshade` algorithm has time complexity $O(vkn^2)$, where *v* is the cost for computing visibility of a pixel.

Once more we can take advantage of the epipolar geometry in order to incrementally determine the visibility of points on the visual hull. This reduces the amortized cost of computing visibility to $O(l)$ per desired pixel, thus giving an implementation of `IBVHshade` that takes $O(lkn^2)$.

Consider the visibility problem in flatland as shown in Figure 4. For a pixel *p*, we wish to determine if the front-most point on the visual hull is occluded with respect to a particular reference image by any other pixel interval in *d*.

Figure 4 – In order to compute the visibility of an IBVH sample with respect to a given reference image, a series of IBVH intervals are projected back onto the reference image in an occlusion-compatible order. The front-most point of the interval is visible if it lies outside of the unions of all preceding intervals.

Efficient calculation can proceed as follows. For each reference view *r*, we traverse the desired-view pixels in front-to-back order with respect to *r* (left-to-right in Figure 4). During traversal, we accumulate coverage intervals by projecting the IBVH pixel intervals into the reference view, and forming their union. For each front most point, `pt3`, we check to see if its projection in the reference view is already covered by the coverage intervals computed thus far. If it is covered, then `pt3` is occluded from *r* by the IBVH. Otherwise, `pt3` is not occluded from *r* by either the IBVH or the actual (unknown) geometry.

```
visibility2D(intervalFlatlandImage &d, referenceImage r){
  intervals  coverage = <empty>
  for each pixel p in d do \\front to back in r
    ray2D ry2 = compute2Dray(p,d.camInfo)
    point2 pt2 = front(p.intervals,ry2);
    point1D p1 = project(pt2,r.camInfo)
    if contained(p1,coverage)
      p.visible[r] = false
    else
      p.visible[r] = true
    intervals tmp  =
                 prjctIntrvls(p.intervals,ry2,r.camInfo)
    coverage = coverage UNION tmp
}
```

This algorithm runs in $O(nl)$, since each pixel is visited once, and containment test and unions can be computed in $O(l)$ time.

Figure 5 – Ideally, the visibility of points in 3D could be computed by applying the 2D algorithm along epipolar planes.

In the continuous case, 3D visibility calculations can be reduced to a set of 2D calculations within epipolar planes (Figure 5), since all visibility interactions occur within such planes. However, the extension of the discrete 2D algorithm to a complete discrete 3D solution is not trivial, as most of the discrete pixels in our images do not exactly share epipolar planes. Consequently, one must be careful in implementing conservative 3D visibility.

Figure 6 – An epipolar wedge includes all pixels between two epipolar lines that might potentially occlude each other.

A guaranteed conservative, actually over-conservative, visibility solution can be computed as follows. We define an epipolar wedge that starts from the epipole e_{rd} in the desired view, and extends to a pixel-width interval on the image boundary. Depending on the relative camera views, we traverse the wedge either towards or away from the epipole [18]. All pixels that are touched by a wedge can be computed with two "nearest-grid DDA" lines. For each pixel in the wedge, we compute its visibility with respect to other pixels in the wedge using the 2D-visibility algorithm previously discussed. If a pixel is declared *visible*, then no geometry within the wedge could have occluded this pixel in the reference view. Since a pixel may be included in more than one wedge, the AND of its visibility test in all relevant wedges determines its final visibility. There are $O(n)$ wedges. The unions of their extents cover the whole image. Each wedge has $O(n)$ pixels traversed, so visibility can be computed in $O(lkn^2)$.

The visibility test described above can be excessively conservative, particularly when combined with the inherent tendency of the visual hull to block surface regions that are not occluded by the actual geometry. It is possible to achieve better visual results by choosing a different visibility criterion. Consider a sample-sized patch on the visual hull to be visible when *any ray* from the epipole has an unobscured view of *any portion* of the patch. Under this definition, the *AND*ing step of the conservative algorithm is replaced with an *OR*. This modification provides us with a certain amount of "hole-filling" in regions that would otherwise be considered occluded, and it provides a small performance advantage. We have used this non-conservative visibility criterion, for the results presented here, and there are few noticeable artifacts.

The total time complexity of our IBVH algorithms is $O(lkn^2)$, which allows for efficient rendering of IBVH objects. These algorithms are well suited to distributed and parallel implementations. We have demonstrated this efficiency with a system that computes IBVHs in real time from live video sequences.

Figure 7 – Four segmented reference images from our system.

5 System Implementation

Our system uses four calibrated Sony DFW500 FireWire video cameras. We distribute the computation across five computers, four that process video and one that assembles the IBVH (see Figure 7). Each camera is attached to a 600 MHz desktop PC that captures the video frames and performs the following processing steps. First, it corrects for radial lens distortion using a lookup table. Then it segments out the foreground object using background-subtraction [1] [10]. Finally, the silhouette and texture information are compressed and sent over a 100Mb/s network to a central server for IBVH processing.

Our server is a quad-processor 550 MHz PC. We interleave the incoming frame information between the 4 processors to increase throughput. The server runs the IBVH intersection and shading algorithms. The resulting IBVH objects can be depth-buffer composited with an OpenGL background to produce a full scene. In the examples shown a model of our graphics lab made with the Canoma modeling system was used as a background.

Figure 8 – A plot of the execution times for each step of the IBVH rendering algorithm on a single processor. A typical IBVH might cover approximately 8000 pixels in a 640×480 image and it would execute at greater than 8 frames per second on our 4 CPU machine.

In Figure 8, the performances of the different stages in the IBVH algorithm are given. For these tests, 6 input images with resolutions of 256×256 were used. The average number of times that a projected ray crosses a silhouette is *6.5*. Foreground segmentation (done on client) takes about 85 ms. We adjusted the field of view of the desired camera, to vary the number of pixels occupied by the object. This graph demonstrates the linear growth of our algorithm with respect to the number of output pixels.

6 Conclusions and Future Work

We have described a new image-based visual-hull rendering algorithm and a real-time system that uses it. The algorithm is efficient from both theoretical and practical standpoints, and the resulting system delivers promising results.

The choice of the visual hull for representing scene elements has some limitations. In general, the visual hull of an object does not match the object's exact geometry. In particular, it cannot represent concave surface regions. This shortcoming is often considered fatal when an accurate geometric model is the ultimate goal. In our applications, the visual hull is used largely as an imposter surface onto which textures are mapped. As such, the visual hull provides a useful model whose combination of accurate silhouettes and textures provides surprisingly effective renderings that are difficult to distinguish from a more exact model. Our system also requires accurate segmentations of each image into foreground and background elements. Methods for accomplishing such segmentations include chromakeying and image differencing. These techniques are subject to variations in cameras, lighting, and background materials.

We plan to investigate techniques for blending between textures to produce smoother transitions. Although we get impressive results using just 4 cameras, we plan to scale our system up to larger numbers of cameras. Much of the algorithm parallelizes in a straightforward manner. With k computers, we expect to achieve $O(n^2\, l \log k)$ time using a binary-tree based structure.

Figure 9 - Example IBVH images. The upper images show depth maps of the computed visual hulls. The lower images show shaded renderings from the same viewpoint. The hull segment connecting the two legs results from a segmentation error caused by a shadow.

7 Acknowledgements

We would like to thank Kari Anne Kjølaas, Annie Chio, Tom Buehler, and Ramy Sadek for their help with this project. We also thank DARPA and Intel for supporting this research effort. NSF Infrastructure and NSF CAREER grants provided further aid.

8 References

[1] Bichsel, M. "Segmenting Simply Connected Moving Objects in a Static Scene." *IEEE PAMI* 16, 11 (November 1994), 1138-1142.

[2] Boyer, E., and M. Berger. "3D Surface Reconstruction Using Occluding Contours." *IJCV* 22, 3 (1997), 219-233.

[3] Chen, S. E. and L. Williams. "View Interpolation for Image Synthesis." *SIGGRAPH 93*, 279-288.

[4] Chen, S. E. "Quicktime VR – An Image-Based Approach to Virtual Environment Navigation." *SIGGRAPH 95*, 29-38.

[5] Curless, B., and M. Levoy. "A Volumetric Method for Building Complex Models from Range Images." *SIGGRAPH 96*, 303-312.

[6] Debevec, P., C. Taylor, and J. Malik, "Modeling and Rendering Architecture from Photographs." *SIGGRAPH 96*, 11-20.

[7] Debevec, P.E., Y. Yu, and G. D. Borshukov, "Efficient View-Dependent Image-based Rendering with Projective Texture Mapping." *Proc. of EGRW* 1998 (June 1998).

[8] Debevec, P. *Modeling and Rendering Architecture from Photographs.* Ph.D. Thesis, University of California at Berkeley, Computer Science Division, Berkeley, CA, 1996.

[9] Faugeras, O. *Three-dimensional Computer Vision: A Geometric Viewpoint.* MIT Press, 1993.

[10] Friedman, N. and S. Russel. "Image Segmentation in Video Sequences." *Proc 13th Conference on Uncertainty in Artifical Intelligence* (1997).

[11] Goldfeather, J., J. Hultquist, and H. Fuchs. "Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System." *SIGGRAPH 86*, 107-116.

[12] Gortler, S. J., R. Grzeszczuk, R. Szeliski, and M. F. Cohen. "The Lumigraph." *SIGGRAPH 96*, 43-54.

[13] Kanade, T., P. W. Rander, and P. J. Narayanan. "Virtualized Reality: Constructing Virtual Worlds from Real Scenes." *IEEE Multimedia* 4, 1 (March 1997), 34-47.

[14] Laurentini, A. "The Visual Hull Concept for Silhouette Based Image Understanding." *IEEE PAMI* 16,2 (1994), 150-162.

[15] Levoy, M. and P. Hanrahan. "Light Field Rendering." *SIGGRAPH 96*, 31-42.

[16] Lorensen, W.E., and H. E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm." *SIGGRAPH 87*, 163-169.

[17] McMillan, L., and G. Bishop. "Plenoptic Modeling: An Image-Based Rendering System." *SIGGRAPH 95*, 39-46.

[18] McMillan, L. *An Image-Based Approach to Three-Dimensional Computer Graphics*, Ph.D. Thesis, University of North Carolina at Chapel Hill, Dept. of Computer Science, 1997.

[19] Moezzi, S., D.Y. Kuramura, and R. Jain. "Reality Modeling and Visualization from Multiple Video Sequences." *IEEE CG&A* 16, 6 (November 1996), 58-63.

[20] Narayanan, P., P. Rander, and T. Kanade. "Constructing Virtual Worlds using Dense Stereo." *Proc. ICCV* 1998, 3-10.

[21] Pollard, S. and S. Hayes. "View Synthesis by Edge Transfer with Applications to the Generation of Immersive Video Objects." *Proc. of VRST*, November 1998, 91-98.

[22] Potmesil, M. "Generating Octree Models of 3D Objects from their Silhouettes in a Sequence of Images." *CVGIP* 40 (1987), 1-29.

[23] Rander, P. W., P. J. Narayanan and T. Kanade, "Virtualized Reality: Constructing Time Varying Virtual Worlds from Real World Events." *Proc. IEEE Visualization* 1997, 277-552.

[24] Rappoport, A., and S. Spitz. "Interactive Boolean Operations for Conceptual Design of 3D solids." *SIGGRAPH 97*, 269-278.

[25] Roth, S. D. "Ray Casting for Modeling Solids." *Computer Graphics and Image Processing*, 18 (February 1982), 109-144.

[26] Saito, H. and T. Kanade. "Shape Reconstruction in Projective Grid Space from a Large Number of Images." *Proc. of CVPR*, (1999).

[27] Seitz, S. and C. R. Dyer. "Photorealistic Scene Reconstruction by Voxel Coloring." *Proc. of CVPR* (1997), 1067-1073.

[28] Seuss, D. "The Cat in the Hat," *CBS Television Special* (1971).

[29] Szeliski, R. "Rapid Octree Construction from Image Sequences." *CVGIP: Image Understanding* 58, 1 (July 1993), 23-32.

[30] Vedula, S., P. Rander, H. Saito, and T. Kanade. "Modeling, Combining, and Rendering Dynamic Real-World Events from Image Sequences." *Proc. 4th Intl. Conf. on Virtual Systems and Multimedia* (Nov 1998).

Creating and Rendering Image-Based Visual Hulls

Chris Buehler, Wojciech Matusik, Leonard McMillan
MIT, LCS Computer Graphics Group

Steven J. Gortler
Harvard University

Abstract

In this paper, we present efficient algorithms for creating and rendering image-based visual hulls. These algorithms are motivated by our desire to render real-time views of dynamic, real-world scenes. We first describe the visual hull, an abstract geometric entity useful for describing the volumes of objects as determined by their silhouettes. We then introduce the image-based visual hull, an efficient representation of an object's visual hull. We demonstrate two desirable properties of the image-based visual hull. First, it can be computed efficiently (i.e., in real-time) from multiple silhouette images. Second, it can be quickly rendered from novel viewpoints. These two properties motivate our use of the image-based visual hull in a real-time rendering system that we are currently developing .

Introduction

Computer graphics has long been concerned with the rendering of *static synthetic* scenes, or scenes composed of non-moving computer-created models. In time, attention turned to the rendering of *dynamic synthetic* scenes, as exemplified by virtual reality systems, most modern computer games, and the recent computer-animated movies. More recently, many researchers have embraced an image-based rendering approach in which scenes are represented by simple images that may be synthetic or acquired from the real world (say, with a digital camera). In this spirit, work has been done in rendering *static acquired* scenes, non-moving scenes acquired from real-world imagery (e.g., QuicktimeVR). However, relatively little work has been done in the case of *dynamic acquired* scenes. It is the goal of our work to develop an appropriate representation and rendering system for such scenes.

Figure 1. A hypothetical arrangement for acquiring dynamic scenes.

Using our system, a user can control a virtual camera within a moving scene that is acquired in real-time. Such a system has many potential uses. A commonly cited example is the virtual sports camera: users viewing a sporting event would be able to view the event from any angle, perhaps to focus on their favorite player or to see the action better. We are also targeting our current system at other tasks: teleconferencing and virtual sets. In a teleconferencing setting, our system would allow participants to navigate the virtual conference room or change their gaze while viewing the other participants moving in real-time. Applied to synthetic sets, our system would enable a director to see his actors perform in real-time in a dynamic three-dimensional virtual set.

A dynamic, acquired rendering system can be designed analogously to a static, acquired one. Static scenes (or objects) are typically acquired from many still photographs taken at different locations. Many photos are acquired, and often the same camera is used to take them. To extend this scenario to the dynamic case, we substitute video sequences for still photographs and place multiple, synchronized video cameras around the scene to acquire these sequences (see Figure 1). The dynamic setup is more restrictive than the static case: the number of input images is limited by the physical number of video cameras, and the cameras can only be placed in locations that do not impede the activity in the scene.

In both the static and dynamic cases, the acquired images are generally processed in some way—details vary from system to system—after which new images of the scene (or object) can be produced from arbitrary camera locations. In the dynamic case, a distinction can be made between *real-time* systems, those that process video and synthesize views at interactive rates, and *off-line* systems, those that require more extensive processing or rendering before viewing. In this paper, we are concerned with real-time systems.

There are a number of challenges inherent in real-time systems. The first is processing all the video frames at interactive rates. Obvious approaches for extracting useful information from multiple video streams, such as multi-baseline stereo algorithms, run too slow on current general-purpose hardware for a real-time system. The second challenge is rendering new views such that a virtual image exhibits as much visual fidelity as an image from one of the real cameras. For example, voxel-based systems often display noticeable artifacts in their images as a result of the low-resolution voxel data structure.

Our real-time system for rendering dynamic, acquired objects is designed to meet these challenges. We utilize between five and ten synchronized, digital video cameras to acquire continuous video streams. To achieve interactive rates, we process the video streams using efficient silhouette-based techniques to create a approximate on-the-fly models (called the *visual hull*) of the dynamic scene objects. We then create novel views of these dynamic objects using image-based rendering techniques, which are fast and preserve much of the detail of the original video sequences.

Related Work

Kanade's virtualized reality system [Kanade97] is perhaps closest in spirit to the dynamic acquired rendering system that we envision, although it is not currently a real-time system. They use a collection of cameras in conjunction with multi-baseline stereo techniques to extract models of dynamic scenes. Currently their method still requires significant off-line preprocessing time to perform the stereo correlation, but they are exploring special purpose hardware for this task, an option we wish to avoid. Recently, they have begun using silhouette methods such as the ones we use to improve the quality of their stereo reconstruction [Vedula98].

Pollard and Hayes [Pollard98] attempt to solve the problem of rendering real-time acquired data with their immersive video objects. Immersive video objects are annotated video streams that can be morphed in real-time to simulate three-dimensional camera motion. Their representation also utilizes object silhouettes, but in a different manner. They match silhouette edges across multiple views, and use these correspondences to compute a morph to a novel view. This approach has some problems, however, as silhouette edges are generally not consistent between views. These inconsistencies require their cameras to be placed close together, limiting the usefulness of the system.

Static Silhouette Methods

Silhouette contours have been used by computer vision researchers build approximate geometric models of static objects and scenes. These techniques are attractive because of the ease of extracting and working

with silhouettes.

Typically these object models are computed by using silhouettes to "carve" away regions of empty space. Potmesil describes a method for computing a voxel representation of objects from sequences of silhouettes [Potmesil87]. He uses an octree data structure to represent a binary volume of space, and does not attack the problem of reconstructing novel views of his objects.

Szeliski has implemented a similar idea [Szeliski92]. He uses a turntable to rotate objects in front of a real camera. After automatically extracting object silhouettes, he computes an octree-based voxel representation of the object by projecting octree nodes into the silhouette images.

Laurentini, recognizing the interest in silhouette methods, has introduced a formalism for analyzing object reconstruction from silhouettes [Laurentini94]. Central to his theory is the concept of the visual hull, which, is the best approximation to an object's shape that one can build from simple silhouettes. His framework is useful for understanding the limitations of silhouette methods, something that has not been quantified in earlier work.

Other volumetric carving methods, related to silhouette techniques, have also been suggested. These include volumetric reconstruction from active laser-range data [Curless96] and volumetric reconstruction based on photometric sample correspondences [Sietz97]. These techniques could be used to improve upon the approximate object models that are obtained from silhouettes. However, currently, they are not as well suited to real-time implementation.

Image-Based Rendering

Image-based rendering has been proposed as a practical alternative to the traditional modeling/rendering framework. In image based rendering, one starts with images and directly produces new images from this data. This avoids the traditional (i.e., polygonal) modeling process, and often leads to rendering algorithms whose running time is independent of the scene's geometric and photometric complexity.

Chen's QuicktimeVR [Chen95] is one of the first commercial static, acquired rendering systems. This system relies heavily on image-based rendering techniques to produce photo-realistic panoramic images of real scenes. Although successful, the system has some limitations: the scenes are static and the viewpoint is fixed.

McMillan's plenoptic modeling system [McMillan95] is QuicktimeVR-like, although it does allow a translating viewpoint. The rendering engine is based on three-dimensional image warping, a now commonplace image-based rendering technique. Dynamic scenes are not supported as the panoramic input images require much more off-line processing than the simple QuicktimeVR images.

Light field methods [Gortler96, Levoy96] represent scenes as a large database of images. Processing requirements are modest making real-time implementation feasible, if not for the large number of cameras required (on the order of hundreds). The cameras must also be placed close together, resulting in a small effective navigation volume for the system.

Paper Organization

In the next section we describe the visual hull, the approximate geometric representation that we use in our system. We demonstrate how it is related to object silhouettes, and why silhouette-based analysis techniques are well suited to this sort of system. We also point out some of the problems with using the visual hull as an object approximation.

In the second section, we describe various algorithms for computing visual hulls using a image-based representation. The first algorithm is slow, but conceptually simple, while the second algorithm is faster and more sophisticated. We present advantages and disadvantages and runtime analyses.

The third section discusses various rendering algorithms for image-based visual hulls. We have investigated at least four algorithms, each with strengths and weaknesses. In this paper, we discuss three of the algorithms.

Silhouettes and the Visual Hull

Silhouette methods are well suited to real-time analysis of object shape. First, computing object silhouettes is fast and relatively robust. Second, multiple silhouettes of an object give a strong indication

of that object's shape.

Computing Silhouettes

An object silhouette is essentially a binary segmentation of an image in which pixels are labeled "foreground" (belonging to the silhouette) or "background." In this paper, background pixels are typically drawn in white and foreground pixels non-white.

One common technique for computing silhouettes is chromakeying, or bluescreen matting [Smith96]. In this technique, the actual scene background is a single uniform color that is unlikely to appear in foreground objects. Foreground objects can then be segmented from the background by using color comparisons. Chromakey techniques are widely used in television weather forecasts and for cinematic special effects, which demonstrates their speed and quality. However, chromakey techniques do not admit arbitrary backgrounds, which is a severe limitation.

More general is a technique called background subtraction or image differencing [Bichsel94, Friedman97]. In background subtraction, a statistical model of a background scene is accumulated from many images. Changes in the scene, such as a figure walking into view, can then be detected by computing the difference between the new frame and the retained model. Differences that fall outside the allowed margins of the model are classified as foreground objects. There are many variations on the above two algorithms, but almost all of them are fast and robust enough to be used in a real-time system.

Shape from Silhouettes: The Visual Hull

It seems intuitive that the shape of an object can be recovered from many silhouettes. However, it is also clear that not all shapes can be recovered from silhouettes alone. For example, the concave region inside a bowl will never be evident in any silhouette, so any method based solely on silhouettes will fail to reconstruct it completely [Koenderink90].

Laurentini has introduced the concept of the visual hull for understanding the shapes of objects that can be reconstructed from their silhouettes [Laurentini94]. Loosely, the visual hull of an object is the closest approximation to that object that can be obtained from silhouettes alone.

The visual hull of an object depends both on the object itself and on a particular viewing region. A viewing region is a set of points in space from which silhouettes of an object are seen. The viewing region might be the set of all points enclosing the object, or, in a more practical case, a finite set of camera positions arranged around the object.

Formally, the visual hull of object S with respect to viewing region R, denoted $VH(S, R)$, is a volume in space such that for each point $P \in VH(S,R)$ and each viewpoint $V \in R$, the half-line from V through P contains at least one point of S [Laurentini94]. This definition simply states that the visual hull consists of all points in space whose images lie within all silhouettes viewed from the viewing region. Stated another way, the visual hull is the maximal object that has the same silhouettes as the original object, as viewed from the viewing region.

It is useful to think of an alternative, constructive definition of the visual hull with respect to a viewing region. Given a point V in the viewing region R, the silhouette of the object as seen from V defines a generalized cone in space with its apex at V (see Figure 2). The intersection of the cones from every point in R results in the visual hull with respect to R.

Figure 2. The intersection of the three silhouette cones defines the visual hull as seen from the viewing region. In this case, the viewing region contains only the apexes of the three silhouette cones.

This definition is useful because it implies a practical way to compute a visual hull. Almost all useful visual hull construction algorithms use some sort of volume intersection technique, as discussed in later sections.

Limitations of the Visual Hull

In the following discussion, we will assume that the viewing region for the visual hull is the set of all "reasonable" vantage points: those points outside the convex hull of the object. Using this special viewing region results in the closest possible approximation to the actual object. This viewing region is also assumed whenever reference is made to a visual hull whose viewing region is not implied by context.

The visual hull is a superset that contains the actual object's shape. It cannot represent concave surface regions (e.g., the inside of a bowl), in general, or even convex or hyperbolic points that are below the rim of a concavity (e.g., a marble inside a bowl). However, the visual hull is a tighter fit to the object than a convex hull, which only includes object regions that are globally convex. The visual hull of a convex object is the same as the object. However, the visual hull of an object composed of multiple, disjoint convex objects may not be the same as the actual objects, see Figure 3.

Figure 3. The visual hull of these two gray circles (black and gray regions) is slightly larger than the circles themselves. It is delimited by the bi-tangent lines drawn in the figure.

When the viewing region of the visual hull does not completely surround the object, the visual hull

becomes a coarser approximation and may even be larger than the convex hull. The visual hull becomes even worse for finite viewing regions, and may exhibit undesirable artifacts such as phantom volumes (Figure 4).

Figure 4. Intersecting the two silhouette cones results in "phantom" volumes, shown in gray on the left. A third silhouette can resolve the problem in this case (right).

In spite of these limitations, the visual hull is still a useful entity for approximating an object's shape in a dynamic rendering system. Object concavities can largely be camouflaged by object motion or hidden with surface texturing. Viewing regions that do not surround the object can be used as long as the virtual camera is confined to locations within the viewing region, as the visual hull is guaranteed to reproduce correctly all silhouettes seen from within the viewing region. Artifacts arising from using a finite viewing region (i.e., a finite amount of cameras) can be lessened by sampling a desired viewing region with appropriately placed viewpoints.

An Image-Based Visual Hull

One could attempt to compute a visual hull geometrically, but this approach, based on the intersection of multiple polytopes, is difficult to implement robustly and the resulting representation is composed of a great number of polygons if the silhouette contour is complex.

As a result, most visual hulls have been computed volumetrically by successively carving away all voxels outside of the projected silhouette cone. However, volumetric approaches suffer from problems with resolution. First, volumetric data structures are generally very memory intensive. This limitation is reduced somewhat by the fact that visual hull is a binary volume, and it is thus well suited to octree-type representations. However, it is still difficult to retain the full precision of the original silhouette images using a standard volumetric representation. If arbitrary configurations of input images are allowed then the intersection of the projected regions from them can have an arbitrarily high spatial frequency content. Thus no uniform spatial sampling is sufficient for exactly representing the final volume. Of course, reasonable approximations can be made by requiring the resulting volume to project to a silhouette contour that is within some error bound relative to the original.

In our approach, we prefer to use an image-based representation of the visual hull, which alleviates some of the problems with a standard voxel approach. In the graphics community, the term "image-based" has had many interpretations. In the strictest sense, an image-based representation consists solely of images (possibly along with matrices describing camera configurations). Along these lines, an image-based representation of the visual hull is simply the set of silhouettes themselves (along with the associated viewpoints). By definition, such a representation preserves the full resolution of the input images and contains no more or no less information than that provided by the silhouettes.

More generally, an "image-based" representation is often identified with a two-dimensional, sampled representation. For example, a standard color image is a rectangular grid of color samples, and a depth image is a grid of depth samples. Note that the samples are not considered connected in any way; they simply exist at regular intervals. The bulk of this paper is concerned with this second form of image-based visual hull.

This second type of image-based visual hull is constructed with respect to some viewpoint V in the viewing region of the visual hull. We can imagine that a camera at this viewpoint sees a silhouette image, which is discretized into a grid of pixels (i.e., samples). For each pixel in this silhouette image a list of occupancy intervals is stored. If a pixel does not belong to the silhouette (i.e., it is background), then the list is empty. Otherwise, the list contains intervals of space that are occupied by the visual hull of the object. These intervals, extruded over the solid angle subtended by the pixel, represent the region of the visual hull that projects to that pixel. The union of all such slices gives the visual hull as sampled from that viewpoint. In Figure 5, we show a slice of an image-based visual hull. The lines represent viewing rays along one column of the image, and the dark line segments denote occupied regions of space.

Figure 5. A single slice of an image-based visual hull. A full image-based visual hull contains many such slices, forming a volume in space.

Advantages of the Image-Based Representation

The image-based representation has a number of advantages in terms of storage requirements, computational efficiency, and ease of rendering.

The occupancy intervals can be stored as pairs of real numbers (where the numbers represent the minimum and maximum depths of the interval), similar to a run-length encoded volume. Thus, while the volume is discretized in two dimensions, the third dimension is continuous, allowing for higher resolution volumes than a voxel approach. Note also that this representation can be used for an arbitrary volume; it is not specialized for a visual hull. Similar data structures have been used by [VanHook86] and [Lacroute94] in traditional volume rendering settings.

Computing a visual hull using the image-based representation is much simpler than previous approaches.

As we will show in the next section, the three-dimensional generalized cone intersections and the volumetric carving operations of other methods are replaced with simple interval intersections in our method. These interval intersections are fast and robust, allowing for a real-time calculation of the visual hull.

Rendering the visual hull is also facilitated by the image-based representation. As we show in later sections, this representation can be rendered using only slight modifications to the standard three-dimensional image warp algorithm. This approach minimizes image resampling, as we only resample during rendering, and produces renderings of quality comparable to the input video images.

Mathematical Preliminaries

We first introduce the mathematical notation and concepts that we use in the rest of the paper. Dotted capital letters (e.g., $\dot{C}$) represent points in three-dimensional space, while lowercase over-bar letters (e.g., $\bar{x}$) represent homogeneous image (pixel) coordinates. Matrices (all are 3 x 3) are written in bold capital letters (e.g., **P**), while scalars are lowercase (e.g., t). We denote equality up to a scale factor with a dotted equals sign, $\doteq$.

One View

The basic quantity that we manipulate is a *view*, which is an image along with the viewpoint from which it was seen. We characterize a view [$\mathbf{P}$, $\dot{C}$] by a center of projection $\dot{C}$ (i.e., the viewpoint) and an inverse projection matrix $\mathbf{P}$ that transforms homogeneous image coordinates $\bar{x}$ to rays in three-dimensional world space according to the following equation:

$$\dot{X}(t) = \dot{C} + t\mathbf{P}\bar{x} ,$$

where $\dot{X}(t)$ represents three-dimensional world points parameterized by the distance (or range) t along a ray. Conceptually, these rays originate at $\dot{C}$ and pass through the pixel $\bar{x}$ = $[u, v, 1]^{\mathrm{T}}$ in the imaging plane.

Often it is computationally more convenient to work with the reciprocal of the range parameter t. We call this quantity the *generalized disparity*, defined as

$$\delta = \frac{1}{t} .$$

Two Views

Two views [$\mathbf{P}_1$, $\dot{C}_1$] and [$\mathbf{P}_2$, $\dot{C}_2$] with different centers of projection (i.e., $\dot{C}_1 \neq \dot{C}_2$) are related by a so-called epipolar geometry. This geometry describes how a ray through a pixel in one view is seen as an *epipolar line* in the other view. Mathematically, this relationship between pixel coordinates in one view and epipolar lines in a second view is expressed by the *fundamental matrix* $\mathbf{F}_{21}$ between the two views [Faugeras93]. That is,

$$\bar{x}_2{}^T \mathbf{F}_{21} \bar{x}_1 = 0 ,$$

where the quantity $\mathbf{F}_{21}\bar{x}_1$ gives the coefficients of a line equation in the second image. Given two views [$\mathbf{P}_1$, $\dot{C}_1$] and [$\mathbf{P}_2$, $\dot{C}_2$], their fundamental matrix can be computed as

$$\mathbf{F}_{21} = \mathbf{E}_2 \mathbf{P}_2^{-1} \mathbf{P}_1 .$$

Matrix $\mathbf{E}_2$ is a matrix representation of the cross product defined such that

$$\mathbf{E}_2 v = \bar{e}_2 \times v ,$$

where v is an arbitrary vector and vector $\bar{e}_2$ is the *epipole*, or the projection of the first view's center of projection onto the second view's image plane. This epipole is computed as

$$\bar{e}_2 = \mathbf{P}_2^{-1}(\dot{C}_1 - \dot{C}_2) ,$$

and the epipole of the first view with respect to the second is computed similarly.

Often we want to calculate a desired view from a known view. Given two views [$\mathbf{P}_1$, $\dot{C}_1$] and [$\mathbf{P}_2$, $\dot{C}_2$], where the first one is known and the second one is desired, we can transform pixels from the known view to pixels in the desired view using a three-dimensional image warping equation [McMillan96]:

$$\bar{x}_2 \doteq \mathbf{P}_2^{-1}\mathbf{P}_1\bar{x}_1 + \delta_1 \mathbf{P}_2^{-1}(\dot{C}_1 - \dot{C}_2) . \tag{1}$$

This equation gives pixel coordinates $\bar{x}_2$ in the desired view of the point defined by the pixel $\bar{x}_1$ and the disparity δ_1 in the first view. Thus, computing a desired view from a single known view requires auxiliary disparity information, which is often stored in the form of a depth image associated with the known view.

In computing image-based visual hulls, we are often interested in recovering the range (or disparity)

parameter t_2 given corresponding image points in two views. We solve this problem by computing the range parameters of the points of closest approach of the two rays defined by the corresponding pixels in two images as follows:

$$t_1 = \frac{\det\left[\dot{C}_2 - \dot{C}_1 \quad \mathbf{P}_2\bar{x}_2 \quad \mathbf{P}_1\bar{x}_1 \times \mathbf{P}_2\bar{x}_2\right]}{\left\|\mathbf{P}_1\bar{x}_1 \times \mathbf{P}_2\bar{x}_2\right\|^2}.$$

The parameter t_2 can be computed similarly.

Three Views

It has been shown [Shashua97] that three views are related by a mathematical entity called the trilinear tensor. Similar to the fundamental matrix for two views, the trilinear tensor describes the relationship between points and lines in the three views. A complete description of the trilinear tensor is beyond the scope of this paper, however, we do present four equations derived from the tensor which relate the coordinates of a pixel $\bar{p}'' = [x'', y'', 1]$ in a third view to the coordinates of pixels in two other views ($\bar{p} = [x, y, 1]$ and $\bar{p}' = [x', y', 1]$):

$$\begin{aligned}
x''\alpha_i^{13}\bar{p}^i - x''x'\alpha_i^{33}\bar{p}^i + x'\alpha_i^{31}\bar{p}^i - \alpha_i^{11}\bar{p}^i &= 0\,, \\
y''\alpha_i^{13}\bar{p}^i - y''x'\alpha_i^{33}\bar{p}^i + x'\alpha_i^{32}\bar{p}^i - \alpha_i^{12}\bar{p}^i &= 0\,, \\
x''\alpha_i^{23}\bar{p}^i - x''y'\alpha_i^{33}\bar{p}^i + y'\alpha_i^{31}\bar{p}^i - \alpha_i^{21}\bar{p}^i &= 0, \\
y''\alpha_i^{23}\bar{p}^i - y''y'\alpha_i^{33}\bar{p}^i + y'\alpha_i^{32}\bar{p}^i - \alpha_i^{22}\bar{p}^i &= 0\,.
\end{aligned}$$

In the above equations, α_i^{jk} (i,j,k = 1,2,3) is the 27 element trilinear tensor, and the notation $\alpha_i^{nm}\bar{p}^i$ denotes a dot-product of a row of the tensor with $\bar{p}$. The elements of α_i^{jk} are obtained from the three views [$\mathbf{P}_1$, $\dot{C}_1$], [$\mathbf{P}_2$, $\dot{C}_2$], and [$\mathbf{P}_3$, $\dot{C}_3$] according to the formulas given in [Shashua97].

The important quality of these equations, with regard to image synthesis, is that the third pixel's location is completely constrained by the locations of the two other pixels; no auxiliary depth image is needed. As we will demonstrate, these equations can be exploited when rendering novel views given two or more known views.

Creating Image-Based Visual Hulls

In the following sections, we describe algorithms for computing image-based visual hulls from a finite number of silhouette images. In all of these algorithms, the input is assumed to be a set of k silhouettes (i.e., binary images), their associated viewpoints, and a viewpoint from which the visual hull is to be constructed. The algorithms output a sampled image of the visual hull, in which each pixel of the image contains a list of occupied intervals of space.

To ease algorithm analysis, the input silhouettes are assumed to be square m x m arrays of pixels. The output resolution of the image-based visual hull is n x n pixels.

The Basic Algorithm

We implement the same basic idea in all of our visual hull construction algorithms. We cast a ray into space for each pixel in the desired view of the visual hull. We intersect this ray with the k silhouette cones defined by the k silhouette views and record the intersections as pairs of enter/exit points (i.e., intervals). This process results in k lists of intervals, which are then intersected together to form a single list. This final list, representing the intersection of the viewing ray with the visual hull, is stored in our data structure.

The key aspect of all our algorithms is that all of the ray/cone intersection calculations are done in two dimensions rather than three. Recall that each silhouette cone is defined by a two-dimensional silhouette

image and a center of projection. Instead of projecting these cones into three-dimensional space and then computing ray intersections, we can project the three-dimensional ray into the two-dimensional space of the silhouette image and perform intersections there. The ray simply projects to a line (in fact, the epipolar line as discussed in a previous section), and the resulting two-dimensional calculations are much more tractable.

The above observations lead directly to an algorithm for computing the image-based visual hull:

```
for each pixel p = [x,y,1] in VHULL
    initialize VHULL[x][y] = [depth_min, depth_max]

for each silhouette image SIL_i
    compute fundamental matrix F_i
    for each pixel p = [x,y,1] in VHULL
        compute epipolar line coefficients F_i p
        trace epipolar line in image SIL_i
        record list of silhouette contour intersection points [p_i,k]
        interval_list = []
        for each pair of intersection points p_i,2l and p_i,2l+1
            compute depth_i,l,min  and depth_i,l,max measured w.r.t. VHULL
            interval_list = interval_list ∪ [depth_i,l,min, depth_i,l,max]
        endfor
    VHULL[x][y] = VHULL[x][y] ∩ interval_list
    endfor
endfor
```

The algorithm is illustrated in Figure 6. Six silhouettes from a synthetic dinosaur model are shown, and the desired image-based visual hull is computed from the viewpoint of the upper left silhouette (the primary view). Three pixels are labeled in this primary view. The corresponding epipolar line for each pixel is shown in the remaining five (secondary) images. The algorithm processes one secondary image at a time. First it detects each interval where the line crosses through the silhouette of the object. At each of these silhouette contour crossings the length along the ray of the primary image is computed using the equation for the point of closest approach. A list of these intervals is computed for each secondary image. Finally, the interval lists are merged by computing their intersections across all secondary images. This process is repeated for every pixel in the primary image.

Figure 6. The image-based visual hull is computed from the viewpoint shown in the upper left. The epipolar lines corresponding to the three labeled pixels are shown in the five other silhouettes.

Analysis

The basic algorithm, while conceptually simple, is not a particularly efficient way to compute image-based visual hulls. The asymptotic running time is $O(km^2n)$, as the algorithm traces a line of length $O(n)$ in k images for each of m^2 pixels in the primary view. This analysis ignores the number of intervals traced and the cost of intersecting them. This omission is justified as there are typically far fewer intervals than the number of pixels in one dimension of a secondary image, and certainly not more than this number. When the primary and secondary images are of the same dimensions, a common case, then the running time is $O(kn^3)$. Thus, we generally consider this an n-cubed algorithm.

The algorithm also suffers from some quantization problems. The digital epipolar lines traced by the algorithm are generally not identical to the ideal epipolar lines. This discrepancy may cause the silhouette intersection points to be slightly off. In practice, such quantization problems have been largely unnoticeable.

Line-Cache Algorithm

The best running time we might expect from a visual hull construction algorithm is $O(km^2)$. This lower bound arises from the fact that we need to fill in interval lists for m^2 pixels, and we need to process k views. One might imagine a faster algorithm, based on a hierarchical decomposition (e.g., a quadtree) of the visual hull image, but here we will assume we want to create m^2 individual interval lists. A hierarchical decomposition, if desired, can then be applied to any of our algorithms.

The line-cache algorithm is an algorithm for computing the image-based visual hull that achieves the $O(km^2)$ running time. The increased efficiency is due to a simple observation: multiple three-dimensional rays from the primary image project to the same two dimensional line in the secondary images. This fact can be understood from the epipolar geometry between two views. A viewing ray from the primary image and the viewpoint of a secondary image are contained within a plane in space. This plane projects to an epipolar line in the secondary image. Any other viewing ray from the primary image which also lies in this plane projects to the same epipolar line in the secondary image.

The observation can also be demonstrated with a counting argument. It takes roughly $O(n)$ lines of length $O(n)$ to fill a discrete (pixelized) two-dimensional space of size $O(n^2)$. Thus, if we project $O(n^2)$ lines of length $O(n)$ into this space, we can expect that $O(n)$ lines will map to the same line. Of course, this argument is really only valid in a discrete setting, which is the setting in which we compute our image-based visual hulls.

Using the above observation, we amend our basic algorithm in the following way. When we attempt to compute the two dimensional line/silhouette intersection, we first check in an "epipolar line cache" data structure to see if the intersection intervals have already been computed. If so, we used the cached results. Otherwise, we compute the line intersections and store the resulting interval list in the line cache.

Figure 7. We determine line cache indices by the farthest intersection of the epipolar line with the image boundary. Lines that do not intersect this boundary need not be cached.

The only real issue to deal with in this algorithm is how to index the cache. That is, how do we determine that two lines are the same? There are many ways to do this; in our implementation we compute the intersection of the epipolar line with the farthest image boundary (see Figure 7). We use this intersection coordinate as the index to our cache. This indexing style allows us to vary the performance of our cache by changing the resolution of our coordinate system. For example, computing intersections to the nearest half-pixel gives a larger cache that better represents lines, but may result in fewer cache hits. Using the nearest double-pixel results in a smaller cache and more hits, but may group lines that are too dissimilar in the same cache location.

The line-cache algorithm is as follows:

```
for each pixel p = [x,y,1] in VHULL
    initialize VHULL[x][y] = [depth_min, depth_max]

for each silhouette image SIL_i
    for each cache index
        initialize CACHE_i[index] = EMPTY
    endfor
    compute fundamental matrix F_i
    for each pixel p = [x,y,1] in VHULL
        compute epipolar line coefficients F_i p
        compute line cache index = compute_index(F_i p)
        if(CACHE_i[index] = EMPTY)
            trace epipolar line in image SIL_i
            record list of silhouette contour intersection points [p_i,k]
            CACHE_i[index] = [p_i,k]
        else
            [p_i,k] = CACHE_i[index]
        endif
        interval_list = []
        for each pair of intersection points p_i,2l and p_i,2l+1
            compute depth_i,l,min and depth_i,l,max measured w.r.t. VHULL
            interval_list = interval_list ∪ [[depth_i,l,min, depth_i,l,max]]
        endfor
    VHULL[x][y] = VHULL[x][y] ∩ interval_list
    endfor
endfor
```

Analysis

We will consider a worst case running time for the line-cache algorithm in which all cache lines are accessed. The size of each cache is O(n), and for each cache entry a line of length O(n) is traversed, leading to a total time of O(kn^2) spent computing all cache entries. The algorithm spends time O(km^2) retrieving interval lists from the caches. Thus, the runtime is O(kn^2) if $n > m$, and O(km^2) otherwise. In practice, we find that 90% of the cache entries are accessed, so this worst case analysis is applicable.

The line-cache algorithm gains its speed by making some tradeoffs in the quality of the resulting visual hull. In addition to the quantization errors from the basic algorithm, the line cache algorithm introduces errors by mapping slightly different epipolar lines to the same cache location. In practice, such errors are small, although they may be noticeable near depth discontinuity edges.

Rendering Image-Based Visual Hulls

The rendering problem is to produce a novel image of the *original object* as seen from some desired view, given an image-based visual hull of the object along with its original source views (i.e., the camera pose and images before segmentation). Since we have already shown that the visual hull is an approximation to the object's true shape, it will generally be impossible to create the exact image of the object from the new view. Thus, the goal of our rendering algorithms is to reproduce as closely as possible the true object's shape and color with information from the visual hull (shape) and the original camera images (color).

We are interested in a number of additional sub-goals for our rendering algorithms. First, they should be fast enough so that they will be applicable in our dynamic, real-time system. Second, they should offer

high quality imagery in the sense that rendered images should be reasonably indistinguishable from the original camera images.

The inputs to each algorithm are assumed to be an image-based visual hull (n x n pixels), k original camera images (n x n pixels), and a desired view. The output is an m x m pixel image as seen from the desired view.

In all comparisons, we use the synthetic dinosaur images as inputs. The visual hull is computed from six 256 x 256 images. We generate novel renderings from three different viewpoints to exercise the strengths and weaknesses of the different algorithms. All six input dinosaur images are shown in Figure 8.

Figure 8. The six input dinosaur images (textures and silhouettes) used to create and render the image-based visual hull examples in this paper.

Texture Extrusion

The texture extrusion rendering method requires the image-based visual hull to be computed from the same viewpoint as one of the original camera images. In this special case, the pixels in the camera image are in one-to-one correspondence with the pixels in the visual hull image. In other words, each list of occupancy intervals in the visual hull image has a color assigned to it from the corresponding pixel in the camera image.

This special arrangement suggests a simple rendering technique: we can draw the occupancy intervals as seen from the new view, and we can color them with the colors assigned from the camera image. Such a rendering technique amounts to extruding the two-dimensional color image (or texture) along viewing rays to create a three-dimensional textured volume.

The basic requirement to use this technique is an ability to render a list of occupancy intervals from arbitrary viewpoints. The occupancy intervals are essentially long, thin cones in space. Calculating their projected shape exactly in the desired view would be prohibitively expensive for a real-time rendering algorithm. However, for viewpoints that are close to the viewpoint of the visual hull, the occupancy intervals can be approximated by simple line segments. Drawing these line segments can be done very

quickly since it is possible calculate the end points of the line segments efficiently.

The line segment endpoints can be incrementally computed using the three-dimensional warping equation (Equation 1). Recall that the image-based visual hull data structure stores a list of disparity values $[\delta_{1,\min}, \delta_{1,\max}, \ldots, \delta_{k,\min}, \delta_{k,\max}]$ for each pixel $\overline{p} = [x, y, 1]$, much like a Layered Depth Image [Shade98]. As is done when rendering Layered Depth Images, we exploit the fact that the warping equation reduces to a simple function of disparity for a fixed pixel $\overline{p}$:

$$\overline{x}_2(\delta) \doteq \overline{a} + \delta\overline{e}\,, \tag{2}$$

where $\overline{a} = \mathbf{P}_2^{-1}\mathbf{P}_1\overline{p}$ and $\overline{e} = \mathbf{P}_2^{-1}(\dot{C}_1 - \dot{C}_2)$, which are constant for a given $\overline{p}$.

While a Layered Depth Image only stores depth values for front-facing surfaces, we store pairs of depth values that delimit occupied regions of space. Thus, to calculate the endpoints for the line segments, we evaulate this simple expression for each disparity pair $(\delta_{\min}, \delta_{\max})$ in the occupancy interval list. Given the endpoints, we draw the line segments using a fast digital line drawing routine. The complete texture extrusion algorithm is as follows:

```
compute H = P2^-1 P1
compute e = P2^-1 (C1 - C2)
for each pixel p = [x,y,1] in VHULL
    compute a = Hp
    for each interval [d1,min, d1,max] in VHULL[x][y]
        compute line segment endpoints [x1,min, y1,min] = a + d1,min e
        and [x1,max, y1,max] = a + d1,max e using the incremental
        three-dimensional warp equation (Equation 2)
        draw_line(x1,min, y1,min, x1,max, y1,max, VHULL[x][y].color)
    endfor
endfor
```

Analysis

The texture extrusion algorithm runs in time complexity O(n^2m), as it draws a line of length O(m) for each of n^2 interval lists in the visual hull data structure. Although this may not seem fast, in practice it is fast enough for real-time rendering (~ 20 frames/sec). Texture extrusion also produces reasonably good looking images for viewpoints close to the viewpoint of the visual hull. Figure 9a demonstrates a novel viewpoint close to the original one. The visual hull in this case was computed from the viewpoint of the upper left-hand image in Figure 8.

Texture extrusion fails, however, when the desired viewpoint is far from the viewpoint at which the visual hull was sampled. This failure is primarily due to two factors. First, when the viewpoint is moved too far to one side, the extruded colors no longer approximate the true color of the object (see Figure 9b). This problem is unavoidable, as a single camera image can not see the entire object at one time. Second, when the viewpoint is moved very close to the object, the approximation of drawing line segments for the occupancy intervals is no longer valid and the images "explode" (see Figure 9c).

(a) (b) (c)

Figure 9. Images rendered from three novel viewpoints using texture extrusion.

Texture Projection

The texture projection algorithm extends the texture extrusion algorithm to handle a wider range of viewpoints. It corrects the second viewpoint problem, that of incorrect colors for distant viewpoints, by combining colors from multiple textures into a single rendering.

Texture projection is a simple extension to the texture extrusion algorithm. In texture extrusion, a single texture is essentially projected through the volume of the visual hull. Regions of the visual hull that are seen from the texture's viewpoint are colored correctly, while other regions are colored incorrectly. In texture projection, we project multiple textures onto the surface of the visual hull. Regions of the visual hull that are not seen by one texture can be colored with information from another texture.

We implement texture projection by a small modification to the texture extrusion algorithm. Instead of drawing each line segment with a constant color, we projectively texture map the line segment with colors from another texture. The projective texture mapping is done using the trilinear tensor equations. The tensor between the three views—the visual hull's view, the texture's view, and the desired view—allow us to compute texture coordinates in the texture's view given coordinates in the visual hull's view and the desired view. Pseudocode for the algorithm is give below. In the pseudocode [$\mathbf{P}_1$, $\dot{C}_1$] refers to the visual hull's view, [$\mathbf{P}_2$, $\dot{C}_2$] denotes the desired view, and [$\mathbf{P}_k$, $\dot{C}_k$] is one of the texture views.

```
compute H = P2^-1 P1
compute e = P2^-1 (C1 - C2)
for each pixel p = [x,y,1] in VHULL
    compute a = Hp
    for each interval [d1,min, d1,max] in VHULL[x][y]
        compute line segment endpoints [x1,min, y1,min] = a + d1,min e
            and [x1,max, y1,max] = a + d1,max e using the incremental
            three-dimensional warp equation (Equation 1)
        k = select_texture(x, y, l)
        draw_line_proj_tex(x, y, x1,min, y1,min, x1,max, y1,max, P1, C1, P2, C2, Pk, Ck)
    endfor
endfor
```

The auxiliary function `draw_line_proj_tex` implements projective texture mapping using the trilinear tensor computed from [$\mathbf{P}_1$, $\dot{C}_1$], [$\mathbf{P}_2$, $\dot{C}_2$], and [$\mathbf{P}_k$, $\dot{C}_k$]. The function `select_texture` selects the texture to be mapped to the indicated visual hull interval. Many mappings are possible; we implemented a particularly simple strategy in our real-time implementation. We choose the texture with the minimum angle between the visual hull interval and the texture's viewpoint.

Analysis

The texture projection algorithm has the same asymptotic running time as the texture extrusion algorithm, $O(n^2m)$. However, because of the cost of the texture mapping, the hidden constant is much larger, which makes the algorithm slower in practice. The quality of the images is generally better, and the algorithm is useful for larger changes in the viewpoint (see Figures 10a and 10b). However, texture projection does suffer from the same zooming problem as the texture extrusion algorithm (see Figure 10c).

(a) (b) (c)

Figure 10. Images rendered from three novel viewpoints using texture projection.

Ray-Casting

Both the texture extrusion and the texture projection algorithms suffer from the same problem with viewpoints that are too close to the object: the image tends to break apart. This problem is directly related to the fact that both algorithms are *forward mapped*. They transform points from the visual hull to pixels in the desired view, and they may miss pixels along the way. Similar problems exist in other areas of computer graphics, and they are typically solved by using a *backward mapped* algorithm. In such an algorithm, pixels in the desired view are transformed to points in the visual hull. In this manner, every pixel in the desired view can be mapped to some point in the visual hull and colored appropriately.

To implement a backward mapped algorithm for rendering visual hulls, we would like to know for every pixel in the desired view whether or not the ray through that pixel intersects the visual hull. To compute this, we can cast a ray for every pixel in the desired view and test it for intersections with the *k* silhouette cones from the *k* cameras. Or, in other words, we can compute an image-based visual hull from the desired viewpoint.

An image-based visual hull computed from the desired viewpoint effectively gives the *shape* of the visual hull in the form of a depth image. However, we would like to have the proper colors along the with shape. We can compute the colors using a bit of additional computation to back project the visual hull to the *k* camera images and sample the colors. The complete algorithm is as follows:

```
compute VHULL_d from view [P_d, C_d]

for each pixel p = [x, y, 1] in VHULL_d
    extract depth_min from VHULL_d[x][y]
    for each camera image CAM_k
        backproject p to p_k = [x_k, y_k, 1] using Equation 1
        color_k = CAM_k[x_k][y_k]
    endfor
    VHULL_d[x][y] = weighted_avg(color_k)
endfor
```

The function `weighted_avg` simply computes some weighted average of the colors sampled from the *k* camera images. A color weight may be 0 if the camera makes no contribution to the color (e.g., it is occluded) or 1 if the camera contributes all the color (e.g., a winner-take-all strategy). In some cases, calculating the weights may be non-trivial. We use the winner-take-all approach in our implementation. That is, we assign a "best" camera a weight of 1 and assign all other cameras 0 weights. We define the best camera as the camera whose viewing ray is closest to that of the viewing direction. This strategy for assigning camera weights ignores the occlusion problem, and cameras may be selected which actually do not see the pixel to be colored.

Analysis

Due to its backward mapped nature, the ray-casting algorithm has a complexity fundamentally different than the previous two rendering algorithms. The running time is $O(km^2)$, as the visual hull calculation is $O(km^2)$, and the pixel coloring loop backprojects each of m^2 pixels k times. This running time is noteworthy as it is proportional to the size of the desired image and independent of the size of the camera images (for $m > n$). For $m = n$, the algorithm is n-squared, which compares favorably to the n-cubed forward mapped algorithms. However, the hidden constant is large, so this advantage is not realized at typical values of n.

This algorithm is slower than the forward mapped algorithms, but potentially produces images of higher quality (image quality and speed depend on the choice of color weighting). However, since the runtime of this algorithm includes the explicit visual hull calculation, the comparison is slightly unfair. Also, because it is backward mapped, problems with close range viewpoints are avoided. Ray-casting results are shown in Figures 11a, 11b, and 11c.

(a) (b) (c)

Figure 11. Images rendered from three novel viewpoints using ray-casting.

Conclusion

We have introduced the image-based visual hull as an approximate object representation for real-time dynamic acquired rendering systems. The needs of these systems require algorithms that allow for both the analysis of video inputs and the synthesis of rendered outputs to occur in real-time. Our algorithms for creating and rendering image-based visual hulls satisfy these requirements.

We have shown that the visual hull is a reasonable object representation to use in terms of accuracy and robustness. It provides a reasonable approximation to object shape in most cases, and requires only simple silhouette segmentation for acquisition.

We have demonstrated an efficient real-time algorithm for creating visual hulls. First, we exploit epipolar geometry to reduce three-dimensional volume intersections to simpler two-dimensional line intersections. Then, we use a line-caching approach to reuse previously computed results giving a further increase in performance.

Finally, we have presented a number of algorithms for rendering views of image-based visual hulls from novel viewpoints. The texture extrusion algorithm is fast but does not make use of all available color information. The texture projection algorithm, while slower, does utilize color information from all possible cameras. Both algorithms, however, suffer from a problem with viewpoints that are too close to the object. This problem is remedied by the ray-casting algorithm, which generates an image directly from the visual hull calculation.

Acknowledgements

Support for this research was provided by DARPA contract N30602-97-1-0283, and the Massachusetts Institute of Technology's Laboratory for Computer Science. We would also like to thank Anne McCarthy for providing artwork.

References

[Bichsel94] Bichsel, M., *"Segmenting Simply Connected Moving Objects in a Static Scene,"* **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. 16, No. 11, November 1994, pp. 1138-1142.

[Chen95] Chen, S.E., "*QuickTime VR - An Image-Based Approach to Virtual Environment Navigation,*" **Computer Graphics** (SIGGRAPH '95 Conference Proceedings), August 6-11, 1995, pp. 29-38.

[Curless96] Curless, B., and M. Levoy. "*A Volumetric Method for Building Complex Models from Range Images,"* **Computer Graphics** (SIGGRAPH '96 Conference Proceedings), August 4-9, 1996, pp. 43-54.

[Faugeras93] Faugeras, O., **Three-dimensional Computer Vision: A Geometric Viewpoint**, The MIT Press, Cambridge, Massachusetts, 1993.

[Friedman97] Friedman, N., and Russell, S., "*Image Segmentation in Video Sequences,"* **Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence**, 1997.

[Gortler96] Gortler, S.J., Grzeszczuk, R., Szeliski, R, and Cohen, M.F., "*The Lumigraph,*" **Computer Graphics** (SIGGRAPH'96 Conference Proceedings), August 4-9, 1996, pp. 43-54.

[Kanade97] Kanade, T., Rander, P. W., Marayanan, P. J., *"Virtualized Reality: Constructing Virtual Worlds from Real Scenes,"* **IEEE MultiMedia**, Vol.4, No.1, Jan. - Mar. 1997, pp.34-47.

[Koenderink90] Koenderink, J. J., **Solid Shape**, The MIT Press, Cambridge, Massachusetts, 1990.

[Lacroute94] Lacroute, P., Levoy, M., *"Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation,"* **Computer Graphics** (SIGGRAPH '94 Conference Proceedings), July 24-29, 1994, pp. 451-458.

[Laurentini94] Laurentini, A., *"The Visual Hull Concept for Silhouette-Based Image Understanding,"* **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. 16, No. 2, February 1994, pp. 150-162.

[Levoy96] Levoy, M. and P. Hanrahan, "*Light Field Rendering,*" **Computer Graphics** (SIGGRAPH'96 Conference Proceedings), August 4-9, 1996, pp. 31-42.

[McMillan95] McMillan, L., and Bishop, G., "*Plenoptic Modeling: An Image-Based Rendering System,*" **Computer Graphics** (SIGGRAPH '95 Conference Proceedings), August 6-11, 1995, pp. 39-46.

[McMillan96] McMillan, L., *"An Image-Based Approach to Three-Dimensional Computer Graphics,"* Ph.D. Thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1996.

[Pollard98] Pollard, S., and Hayes, S.,"*View Synthesis by Edge Transfer with Applications to the Generation of Immersive Video Objects,"* **Proceedings of the ACM Symposium on Virtual Reality Software and Technology**, November 2-5, 1998, pp. 91-98.

[Potmesil87] Potmesil, M., *"Generating Octree Models of 3D Objects from Their Silhouettes in a Sequence of Images,"* **Computer Vision, Graphics, and Image Processing**, Vol. 40, 1987, pp. 1-29.

[Seitz97] Seitz, S. M., Dyer, C. R., *"Photorealistic Scene Reconstruction by Voxel Coloring,"* **Computer Vision and Pattern Recognition Conference**, 1997, pp. 1067-1073.

[Shade98] Shade, J., Gortler, S., He, L., and Szeliski, R., *"Layered Depth Images,"* **Computer Graphics** (SIGGRAPH '98) Conference Proceedings), July 19-24, 1998, pp. 231-242.

[Shashua97] Shashua, A., *"Trilinear Tensor: The Fundamental Construct of Multiple-view Geometry and its Applications,"* **International Workshop on Algebraic Frames For The Perception Action Cycle (AFPAC)**, Kiel Germany Sep. 8-9, 1997.

[Smith96] Smith, A. R., and Blinn, J. F., *"Blue Screen Matting,"* **Computer Graphics** (SIGGRAPH '96 Conference Proceedings), August 4-9, 1996, pp. 21-30.

[Szeliski92] Szeliski, R., "*Rapid Octree Construction from Image Sequences,"* **CVGIP: Image Understanding**, Vol. 58, No. 1, July 1993, pp. 23-32.

[VanHook86] Van Hook, T., *"Real-time shaded NC milling display,"* **Computer Graphics** (SIGGRAPH '86 Conference Proceedings), 1986, pp. 15-20.

[Vedula98] Vedula, S., Rander, P., Saito, H., Kanade, T., "*Modeling, Combining, and Rendering Dynamic Real-World Events from Image Sequences,"* **4th International Conference on Virtual Systems and Multimedia conference proceedings**, Nov. 1998.