SIGGRAPH'03 Tutorial Course #11

Interactive Geometric & Scientific Computations Using Graphics Hardware

http://gamma.cs.unc.edu/SIG03_COURSE

Organized by

Dinesh Manocha
University of North Carolina at Chapel Hill

Speakers

Dinesh Manocha
University of North Carolina at Chapel Hill

Michael Doggett
ATI

Shankar Krishnan
AT & T Labs

Ming C. Lin
University of North Carolina at Chapel Hill

Marc Pollefeys
University of North Carolina at Chapel Hill

Timothy Purcell
Stanford

Peter Schröder
Caltech

Matthias Wloka
NVIDIA

Abstract

Fast graphics hardware including dedicated vertex processing, 3D rasterization, texturing, and pixel processing is becoming as ubiquitous as floating-point hardware. The ubiquity and performance of this hardware leads us to consider the extent to which this hardware can be harnessed to solve geometric and scientific problems beyond the conventional domain of image synthesis for the sake of pretty animation. In particular, there are a number of complicated geometric and scientific problems whose solutions provide the basis for many application areas in graphics, robotics, vision, simulation, computer gaming, visualization and high-performance computing. Many of the sophisticated "behind-the-curtain" geometric computations are often hard to perform accurately and robustly with reasonable efficiency. At the same time, the graphics processing units offer a lot of potential as generally programmable SIMD and streaming units. This course covers all aspects of using graphics rasterization hardware for interactive geometric and scientific computations.

This course will start with an overview with some of the graphics hardware features that lend themselves to solving geometric and scientific problems. Next we will talk about software APIs and issues in implementing some basic geometric queries on this hardware. After that the course will deal with three main different application areas: geometric arrangements, collision and reconstruction problems, scientific computation including linear solvers, Fast Fourier transforms dynamic and fluid simulation and finally global illumination and interactive walkthroughs. Each talk will present some novel algorithms for these geometric or scientific problems that make use of the capabilities of the rasterization hardware. The speakers will also summarize their experiences in implementing different algorithms on graphics processors, surprises and technical lessons

Course Presenters Information

- **Dinesh Manocha**
Professor, Department of Computer Science,
CB #3175 University of North Carolina,
Chapel Hill, NC 27599-3175
Phones: (919) 962-1749 (office)
Fax: (919) 962-1799
Email: dm@cs.unc.edu
URL: http://www.cs.unc.edu/~dm

Biography: Dinesh Manocha is currently a professor of computer science at the University of North Carolina at Chapel Hill. He received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi in 1987; M.S. and Ph.D. in computer science at the University of California at Berkeley in 1990 and 1992, respectively. During the summers of 1988 and 1989, he was a visiting researcher at the Olivetti Research Lab and General Motors Research Lab, respectively. He received Alfred and Chella D. Moore fellowship and IBM graduate fellowship in 1988 and 1991, respectively, and a Junior Faculty Award in 1992. He was selected an Alfred P. Sloan Research Fellow, received NSF Career Award in 1995, Office of Naval Research Young Investigator Award in 1996, and Hettleman Prize for scholarly achievement at UNC Chapel Hill in 1998. His research interests include geometric and solid modeling, interactive computer graphics, physically based modeling, virtual environments, robotics and scientific computation. He has published more than 120 papers in leading conferences and journals on computer graphics, geometric and solid modeling, robotics, symbolic and numeric computation, virtual reality, molecular modeling and computational geometry. He has served as a program committee member for many leading conferences on virtual reality, computer graphics, computational geometry, geometric and solid modeling and molecular modeling. He was the program co-chair for the first ACM Siggraph workshop on simulation and interaction in virtual environments and program chair of first ACM Workshop on Applied Computational Geometry. He was the guest editor of special issues of International Journal of Computational Geometry and Applications. He has also edited and co-authored two research monographs and consulted for a number of companies including Intel, Mechanical Dynamics, Boeing, Division, TC2 corporation etc.

He has been working on topics related to interactive computer graphics and geometric algorithms for more than ten years. These include collision detection, proximity computations, interactive walkthroughs, visibility, motion planning, multi-pass rendering, and discretized geometric computations. Some of the software systems developed by his research groups have been widely used. He has taught courses on computer graphics, computational geometry and scientific computing at the University of North Carolina for the last six years. He has given invited talks at a number of conferences and workshops and has been a speaker in SIGGRAPH courses. He has also organized other SIGGRAPH courses in the past.

- **Michael Doggett**

ATI Research
62 Forest Street
Marlborough, MA 01752
(508) 303-3900 x3863 (o)
MDoggett@ati.com

Biography: Michael Doggett works as an architect on graphics hardware at ATI Research. He completed his B.S. degree in Computer Science in 1990, B.E. degree in Electrical Engineering in 1992, and Ph.D. in 1997 all at the School of Computer Science and Engineering at The University of New South Wales, Sydney, Australia. From 1996 to 1998 he worked as Chief Engineer at Conja Pty Ltd, a Special Effects, Animation and Design company. From 1998 to 2001 he was a member of the research staff of the Computer Graphics Laboratory (GRIS) at the Computer Science Department of the University of Tuebingen as a PostDoc where he worked on custom hardware for Volume Rendering and Displacement Mapping. He has been involved in teaching courses at the University of New South Wales and the University of Tuebingen. He is the paper co-chair for Graphics Hardware 2002 and has served on the program and review committee for several conferences. He has published numerous papers and is a member of IEEE Computer Society, and ACM.

- **Shankar Krishnan**

Principal Technical Staff Member
AT&T Shannon Laboratory
180 Park Avenue, Room E-201
Florham Park, NJ 07932
(973) 360-8609 (Work)
(973) 660-0336 (Home)
(973) 360-8077 (Fax)
krishnas@research.att.com

Biography: Shankar Krishnan is a Principal Technical Staff Member at AT&T Labs Research and a member of the Information Visualization and Display Research department, where he contributes towards the development of practical new techniques for working with geometric representations of information, with a particular emphasis on problems concerning large-scale networks and services. Prior to joining AT&T Labs, Shankar graduated with a Ph.D. from the University of North Carolina at Chapel Hill. Shankar's primary research interests include 3D computer graphics, hardware-assisted geometric algorithms, and reliable geometric and numeric computation. Shankar has authored several papers in these areas and has given a number of technical presentations in leading conferences in computer graphics, computational geometry and geometric modeling.

- **Ming C. Lin**

Associate Professor, Department of Computer Science
CB #3175 University of North Carolina,
Chapel Hill, NC 27599-3175
Phones: (919) 962-1974 (office)
Fax: (919) 962-1799
Email: lin@cs.unc.edu
URL: http://www.cs.unc.edu/~lin

Biography: Ming C. Lin received her B.S., M.S., Ph.D. degrees in Electrical Engineering and Computer Science in 1988, 1991, 1993 respectively from the University of California, Berkeley. She is currently an assistant professor in the Computer Science Department at the University of North Carolina (UNC), Chapel Hill. Prior to joining UNC, she was an assistant professor in the Computer Science Department at both Naval Postgraduate School and North Carolina A&T State University, and a Program Manager at the U.S. Army Research Office. She received the NSF Young Faculty Career Award in 1995 and Honda Research Initiation award in 1997. Her research interests include real time 3D graphics for virtual environments, applied computational geometry, physically based modeling, robotics and distributed interactive simulation. She has served as a program committee member for many leading conferences on virtual reality, computer graphics, and computational geometry. She was the general chair of the First ACM Workshop on Applied Computational Geometry and the co-Chair of 1999 ACM Symposium on Solid Modeling and Applications. She is also a guest editor of the International Journal on Computational Geometry and Applications, the co-editor of "Applied Computation Geometry", and the Category Editor of ACM Computing Reviews in Computer Graphics. She has also consulted for a number of companies including Intel, Mechanical Dynamics and Division.

Ming has been working in computational geometry, computer graphics and virtual environments for more than nine years. Over the last five years, she has led the development of a number of algorithms and systems for interactive collision detection. These include I-COLLIDE, RAPID, V-COLLIDE, S-COLLIDE, H-COLLIDE, SWIFT, SWIFT++, PIVOT, PQP and DEEP. They have been widely used by a number of researchers and the technology has been licensed by more than 30 commercial organizations. Over the last five years, she has taught courses on computer graphics, physically based modeling, computational geometry and robotics at Naval Postgraduate School, NC A & T University and the University of North Carolina at Chapel Hill. She has given invited lectures at many conferences and meetings, including Computer Games Developers Conference and SIGGRAPH.

- **Marc Pollefeys**

  Assistant Professor
  Department of Computer Science
  University of North Carolina
  Chapel Hill, NC 27599
  Phone: (919) 962-1845
  Fax: (919) 962-1799
  Email: marc@cs.unc.edu

Biography: Marc Pollefeys is an Assistant Professor of Computer Vision in the Department of Computer Science at the University of North Carolina at Chapel Hill. Previously he was a postdoctoral researcher at the Katholieke Universiteit Leuven in Belgium, where he also received his M.S. and Ph.D. degrees in 1994 and 1999, respectively. One of his main research goals is to develop flexible approaches to capture visual representations of real world objects, scenes and events. Dr. Pollefeys has received several prizes for his research, including the prestigious Marr prize at ICCV '98. He is the author or co-author of more than 70 technical papers. He is a regular reviewer for most of the major vision, graphics and photogrammetry journals. He has organized workshops and has served on the program committees of many conferences.
He has organized courses on 'obtaining 3D models with a hand-held camera' at SIGGRAPH 2000, 2001 and 2002, as well as related courses at ECCV 2000, 3DIM 2001. He has co-organized a course on 'multiple view geometry' at CVPR 2001 with Anders Heyden and will be co-organizing a similar course at CVPR 2003 with Andrew Zisserman. He has also contributed to the course on /'acquisition and rendering of surface lightfields/Image-based modeling/' organized at SIGGRAPH 2001 and 2002.

- **Timothy Purcell**

  Gates Computer Science Building, Room 398
  Stanford University
  Stanford, CA 94305
  phones: (650) 723-1367 (work), (650) 497-2251 (home)
  fax: (650) 723-0033
  email: tpurcell@graphics.stanford.edu
  url: http://graphics.stanford.edu/~tpurcell

Biography: Tim Purcell is currently finishing his Ph.D. in computer science at Stanford University. He received a B.S. in computer science from the University of Utah in 1998 and an M.S. in computer science from Stanford University in 2001. He is a recipient of the National Science Foundation Graduate Research Fellowship, and is a 2002-03 NVIDIA fellowship winner. His current research interests include stream programming, ray tracing, and leveraging GPUs for non-traditional uses. He has given a number of technical presentations including a SIGGRAPH course in 2001 and paper talk in 2002. He has also given several invited talks about his research to various companies and organizations including Intel, NVIDIA, and the Silicon Valley ACM SIGGRAPH Chapter.

- **Peter Schröder**

  Professor of Computer Science and Applied and Computational Mathematics
  California Institute of Technology
  Computer Science
  1200 E. California Boulevard
  MC 256-80
  Pasadena, CA 91125
  Phones: (626) 395-4269 (office)
  Fax: (626) 792-4257
  Email: ps@cs.caltech.edu
  URL: http://www.multires.caltech.edu/~ps/

Biography: Peter Schröder is a professor of computer science and applied & computational mathematics at Caltech where he directs the Multi-Res Modeling Group. His research focuses on efficient and robust numerical methods for computer graphics and simulation applications. He is best known for his contributions to the theory and algorithms underlying wavelets, subdivision surfaces, and more broadly, Digital Geometry Processing. In recognition of this work he has received many awards including Sloan and Packard Foundation Fellowships. His work has been published widely including many contributions to the SIGGRAPH conference. As organizer and speaker he has been involved in many highly successful SIGGRAPH courses on Wavelets, Subdivision, and Digital Geometry Processing. He is now applying his experience in massively parallel computers to programmable graphics cards and recently taught a new undergraduate class at Caltech on ``Hacking the GPU."

- **Matthias Wloka**
  Technical Developer Relations
  NVIDIA Corporation
  2701 San Tomas Expressway
  Santa Clara, CA 95050, MS 08
  408 486 2698
  mwloka@nvidia.com

Biography: Matthias Wloka works in the technical developer relations group at Nvidia. There, he gets to collaborate with game-developers on, for example, performance-optimizations and advising how to efficiently implement desired effects into their game. Matthias is always tinkering with the latest graphics hardware to explore the limits of interactive real-time rendering. Before joining Nvidia, Matthias was a game developer himself, working for GameFX/THQ Inc. He received his M.Sc in computer science from Brown University in 1990, and his B.Sc from Christian Albrechts University in Kiel, Germany in 1987.

Table of Contents

Interactive Geometric and Scientific Computations Using Graphics Hardware

Ming C. Lin and Dinesh Manocha

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{lin,dm}@cs.unc.edu
http://gamma.cs.unc.edu

Abstract

Fast graphics hardware, including dedicated vertex processing, 3D rasterization, texturing, and pixel processing, is becoming as ubiquitous as floating-point hardware. The development time between new generations of graphics processor units (GPUs) is currently much less than the development cycle for CPUs. Moreover, the rasterization performance of the GPUs appears to be progressing at a rate faster than Moore's law. Along with multi-pass capabilities, programmability and fast readback bandwidth, the GPUs are becoming useful co-processors for diverse applications that are beyond the conventional domain of image synthesis.

In this paper, we give a brief overview of using GPUs for geometric and scientific applications. These include developing real-time algorithms for different geometric problems including intersection queries, Voronoi diagrams and distance fields, penetration depth computation, robot motion planning, visibility determination and model simplification as well as scientific computations including sparse matrix solvers, conjugate gradient and optimization. All these algorithms effectively utilize the SIMD capabilities and treat GPUs as an efficient processor of images. The main issues, as compared to CPU-based implementations, include lack of general-purpose programming tools for the GPUs, limited precision and storage. We also demonstrate some applications of these algorithms to fast physically-based simulation, real-time navigation of dynamic environments, and interactive display of complex 3D environments.

1 Introduction

High-performance 3D graphics systems are becoming as ubiquitous as floating-point hardware. They are now a part of almost every personal computer or game console. In fact, the two major computational components of a computer system are its main processor (CPU) and its graphics processor, also known as the GPU. While the CPUs are used for general purpose computation, the GPUs were primarily designed for drawing and filling primitives, geometric transformations and texturing. The main application has been fast rendering of lighted, smooth shaded, depth buffered, texture mapped, anti-aliased triangles for visual simulation, virtual reality, and computer gaming. Some of the recent GPUs also include advanced features like multi-texturing [MH99, SAFL99], pixel textures [HS99], programmable shading and programmable vertex engines [LKM01], and support for floating-point fragment pipelines and frame buffers [POAU00]. As graphics hardware becomes more programmable, the barrier between the CPU and the GPU is being redefined. The GPU can also be regarded as an efficient processor of images or a useful co-processor for many diverse applications.

One of the first GPUs was the Geometry Engine (GE) proposed by Clark [Cla82]. It was fabricated using a $3\mu m$ feature size and housed in a 40-pin package. The GPUs have progressed at a fast rate over the last two decades, both in terms of chip complexity as well as rendering performance. Compared to the first GE, a recent GPU like NVIDIA's GeForce3 was manufactured using a $0.18\mu m$ process with a 550-pin package. Its peak fill-rate is 3.84 billion AA samples/second, can perform 960 billion operations per second and has memory bandwidth of 8 GB/sec. Its overall performance is more than three and a half orders of magnitude higher as compared to the first GPU released about 15 years. Details of different GPUs are shown in Fig. 1. The performance growth curve of GPUs has an average slope of $2.4X$, whereas the CPUs have improved in performance by $1.7X$ (per year) over the same time period. In other words, the GPUs have been progressing at a rate faster than Moore's law and this trend is likely to continue in the near-future.

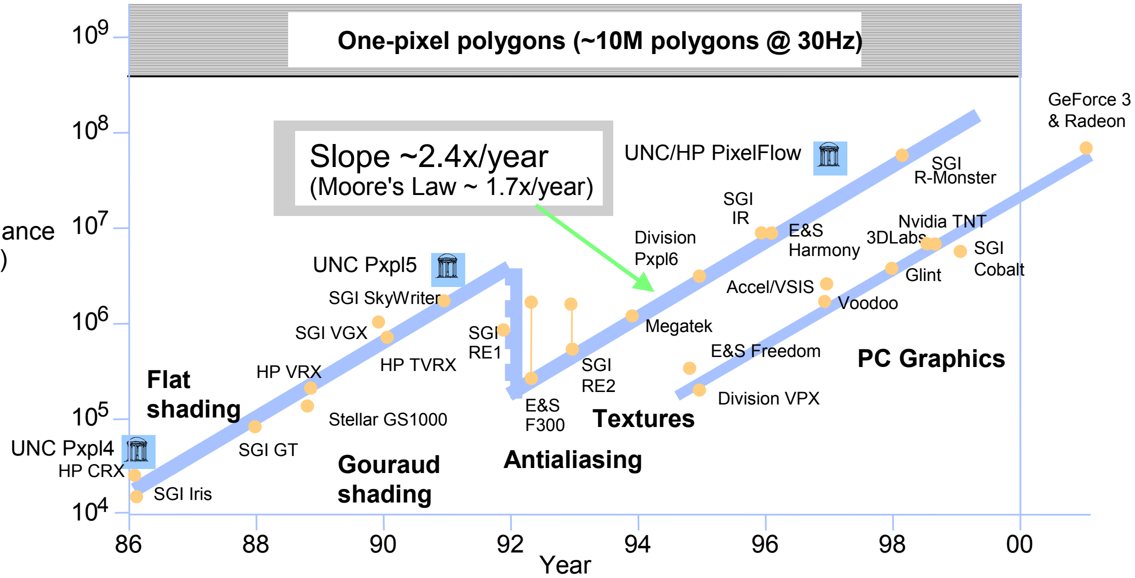


Figure 1: *Performance of Graphics GPUs. They have been progressing at a rate faster than Moore's law. Data Courtesy of John Poulton, UNC Chapel Hill.*

The current GPUs are optimized for rasterization of 3D geometric primitives. They can also be regarded as an efficient processor of images. Moreover, the vertex and pixel shaders provide the application programmer a great deal of flexibility and power. Because of these capabilities, an incredible array of new algorithms and real-time implementations of old algorithms have been made possible.

One of the recent trend in computer graphics has been to use the computation power of CPUs for real-time software rendering. Some examples include real-time ray-tracing of complex scenes using shared memory systems or clusters of PCs [PMS$^+$99, WSB01]. In contrast with these efforts, we focus on using GPUs as a co-processor for geometric and scientific applications.

Why Geometric and Scientific Applications: Many geometric algorithms deal with discretized inputs or outputs [GY86, GGHT97, GM95]. In many cases, the underlying operations can be performed in parallel. This is somewhat similar to rendering algorithms that are implemented in a GPU and are very amenable to pipelining. Furthermore, many applications demand very high computation power for interactive performance and current CPUs are relatively slower by one or two orders of magnitude. Examples of such problems include proximity computations, contact analysis between rigid and deformable models for dynamic simulation, motion planning and navigation in complex static and dynamic environments, visibility computations and model simplification for real-time walkthroughs, visual simulation of diverse dynamic phenomena, such as fluids, clouds and smoke etc. The need to compute real-time solutions of these problems arises in interactive computer graphics, virtual environments, simulation-based design, computer gaming, robotics and scientific applications.

Interpolation-based graphics rasterization hardware is increasingly being used for different geometric applications. These include visibility and shadow computations [ZMHH97], CSG rendering [EJR89, GHF86, Wie96], proximity queries [RMS92, HCK$^+$99, HZLM01], morphing [KR92], motion planning [DLRG90, PHLM00], object reconstruction [MBR$^+$00, Lok01] etc. A recent survey on different applications is given in [TPK01]. All these algorithms perform computations in the 2-D discretized image-space and their accuracy is governed by the underlying pixel resolution. While the initial results are promising, the current approaches can either handle only 2D (or 2.5D) inputs at interactive rates. Other major issues in using GPUs are the difficulty of programmings, lack of high precision and storage. Some of the recent trends, including higher level languages (e.g. NVIDIA's Cg, DirectX's HLSL and OpenGL's SLang), support for 32bit floating point from start to finish of the pipeline, etc. seem to be overcomming these barriers. They have recently been used for many interesting applications including interactive visibility computations on very large models [GSYM02, GLY$^+$03], sparse matric conjugate gradient solver and multigrid solvers [BFGS03], ray-tracing [PBMH02], visual simulation of some dynamic phenomena based on the coupled map lattice [HCSL02], non-linear diffusion [SR01], etc. More information about some of these recent applications is available at:

`http://wwwx.cs.unc.edu/˜harrism/gpgpu/index.shtml`.

Goals of the Course: This course highlights many issues in effectively using the GPUs for different geometric problems. These include:

1. Can we treat GPUs as co-processors and design faster algorithms for geometric computations? What formal models do we use to analyze their performance?

2. What are the main limitations arising from the lack of high precision and programming tools for the GPUs? How can we improve their accuracy?

3. What kind of applications can benefit from the features and capabilities of GPUs?

In particular, we will cover these topics.

- **Faster Algorithms for Geometric Problems:** We will consider three classes of geometric problems. These include proximity computations, arrangements and visibility computations.
- **Faster algorithms for Scientific Problems:** We will survey some fast algorithms for sparse matrix conjugate solvers, regular-grid multigrid solvers and fast fourier transforms as well as some applications to fluid dynamics.
- **Programmability Issues:** We will provide a brief survey of the current set of tools and languages available to program the GPUs. We also address a number of issues in implementing the algorithms on current GPUs.
- **Applications:** We will highlight a number of applications to physically-based simulation, computer vision, robot motion planning, simulation of natural phenomenan and real-time rendering.

References

[BFGS03] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. The gpu as numerical simulation engine. *Proc. of ACM SIGGRAPH*, 2003. To Appear.

[Cla82] J.H. Clark. The geometry engine: A vlsi geometry system for graphics. *Proc. of ACM SIGGRAPH*, pages 127–133, 1982.

[DLRG90] B. R. Donald, J. Lengyel, M. Reichert, and D. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. *Comput. Graph.*, 24(4):327–335, 1990. Proc. of ACM SIGGRAPH.

[EJR89] D. Epstein, F. Jansen, and J. Rossignac. Z-buffering rendering from csg: The trickle algorithm. Technical report, IBM Research Report RC15182, 1989.

[GGHT97] M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.

[GHF86] Jack Goldfeather, Jeff P. M. Hultquist, and Henry Fuchs. Fast constructive-solid geometry display in the Pixel-Powers graphics system. In *Proc. of ACM SIGGRAPH*, volume 20, pages 107–116, 1986.

[GLY$^+$03] N. Govindaraju, B. Lloyd, S. Yoon, A. Sud, and D. Manocha. Interactive shadow generation in complex environments. Technical report, University of North Carolina, 2003. To Appear in Proc. of ACM SIGGRAPH 2003.

[GM95] Leonidas Guibas and David Marimont. Rounding arrangements dynamically. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 190–199, 1995.

[GSYM02] N. Govindaraju, A. Sud, S. Yoon, and D. Manocha. Interactive visibility culling in complex environments with occlusion-switches. Technical Report CS-02-027, University of North Carolina, 2002. To appear in Proc. of ACM Symposium on Interactive 3D Graphics.

[GY86] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *Proc. 27th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 143–152, 1986.

[HCK$^+$99] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardw are. *Proceedings of ACM SIGGRAPH*, pages 277–286, 1999.

[HCSL02] M. Harris, G. Coombe, G. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2002.

[HS99] W. Heidrich and H. P. Seidel. Realistic hardware-accelerated shading and lighting. In *Proc. of ACM SIGGRAPH*, pages 171–178, 1999.

[HZLM01] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha. Fast and simple geometric proximity queries using graphics hardware. *Proc. of ACM Symposium on Interactive 3D Graphics*, 2001.

[KR92] A. Kaul and J. Rossignac. Solid-interpolating deformations: construction and animation of pips. *Computer and Graphics*, 16:107–116, 1992.

[LKM01] E. Lindholm, M. Kilgard, and H. Moreton. A user-programmable vertex engine. *Proc. of ACM SIGGRAPH*, pages 149–158, 2001.

[Lok01] B. Lok. Online model reconstruction for interactive virtual environments. *Proc. of Symposium on Interactive 3D Graphics*, pages 69–72, 2001.

[MBR$^+$00] W. Matusik, C. Buehler, R. Raskar, S. Gortler, and L. McMillan. Image-based visual hulls. *Proc. of ACM SIGGRAPH*, pages 369–374, 2000.

[MH99] M. McCool and W. Heidrich. Texture shaders. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 117–126, 1999.

[PBMH02] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. on Graphics (Proc. of SIGGRAPH'02)*, 21(3):703–712, 2002.

[PHLM00] C. Pisula, K. Hoff, M. Lin, and D. Manocha. Randomized path planning for a rigid body based on hardware accelerated voronoi sampling. In *Proc. of 4th International Workshop on Algorithmic Foundations of Robotics*, 2000.

[PMS$^+$99] S. Parker, W. Martic, P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. *Symposium on Interactive 3D Graphics*, 1999.

[POAU00] M. Peercy, M. Olano, J. Airey, and J. Ungar. Interactive multi-pass programmable shading. *Proc. of ACM SIGGRAPH*, pages 425–432, 2000.

[RMS92] Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider. Interactive inspection of solids: Cross-sections and interferences. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 353–360, July 1992.

[SAFL99] M. Segal, K. Akeley, C. Frazier, and J. Leech. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., 1999.

[SR01] R. Strzodka and M. Rumpf. Nonlinear diffusion in graphics hardware. *Visualization*, pages 75–84, 2001.

[TPK01] T. Theoharis, G. Papaiannou, and E. Karabassi. The magic of the z-buffer: A survey. *Proc. of 9th International Conference on Computer Graphics, Visualization and Computer Vision, WSCG*, 2001.

[Wie96] T F Wiegand. Interactive rendering of csg models. *Computer Graphics Forum*, 15(4):249–261, 1996.

[WSB01] I. Wald, P. Slusallek, and C. Benthin. Interactive distributed ray-tracing of highly complex models. In *Rendering Techniques*, pages 274–285, 2001.

[ZMHH97] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. *Proc. of ACM SIGGRAPH*, 1997.

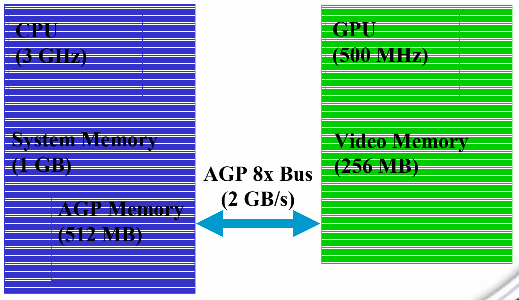
Overview of Graphics Hardware

Matthias M Wloka, NVIDIA

Overview of Graphics Hardware

Matthias Wloka
NVIDIA Corporation

PC Graphics (Current)

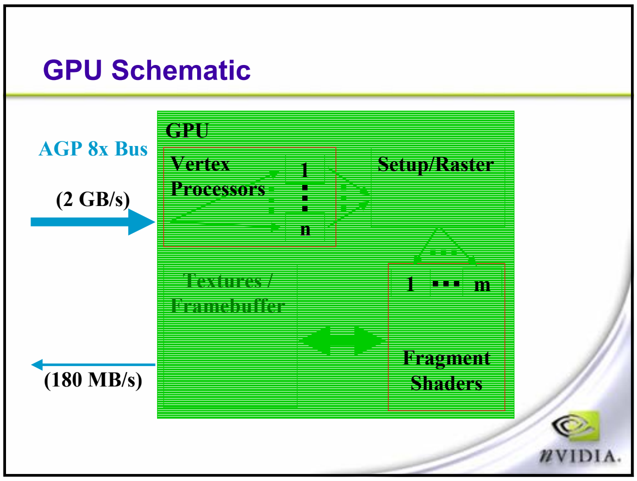


Usual Co-Processor Pitfalls

- Synchronization temporarily idles ALL processors
- Specialized co-processor architecture
  - GPU's deep pipeline means restart is expensive
  - Different mind-set needed to map problems to architecture

GPU as a Co-Processor? Careful!

- CPU programmed as von Neumann architecture
- GPU designed to render graphics
  - MAY be able to abuse it for other computations
- GPU is NOT von Neumann architecture
  - Deep pipeline architecture
  - Pipeline stages are multi-pipe SIMD designs
  - Stages are vector-processors
  - Optimized for large table look-ups (textures)
  - AGP interconnect not symmetric



GPU Schematic

AGP 8x Bus

(2 GB/s)

(180 MB/s)

GPU

Vertex Processors

1

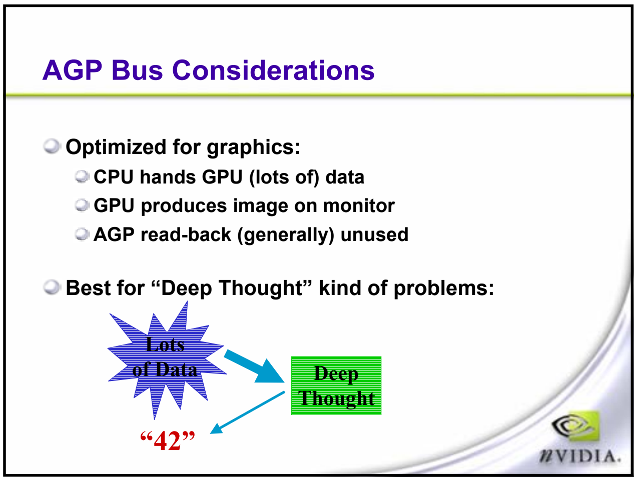
n

Setup/Raster

Textures / Framebuffer

1 ▪▪▪ m

Fragment Shaders



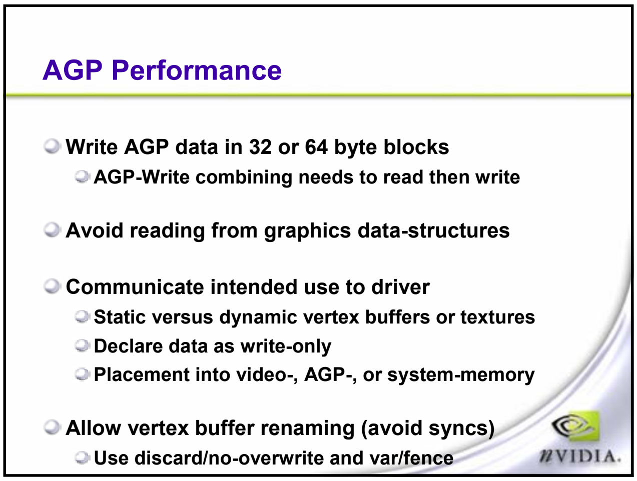
AGP Bus Considerations

- Optimized for graphics:
  - CPU hands GPU (lots of) data
  - GPU produces image on monitor
  - AGP read-back (generally) unused
- Best for "Deep Thought" kind of problems:

Lots of Data

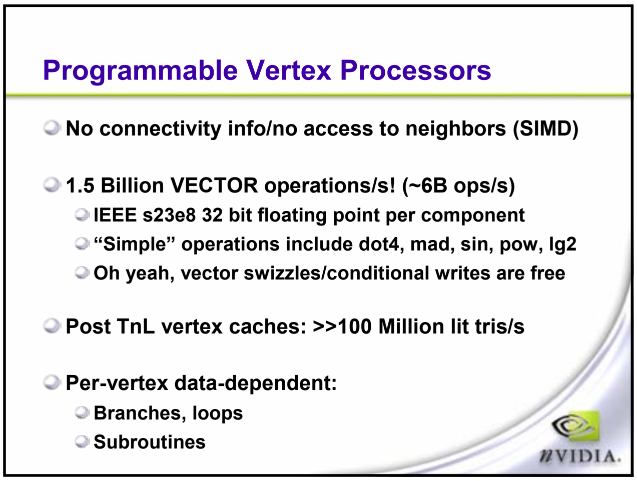
Deep Thought

"42"



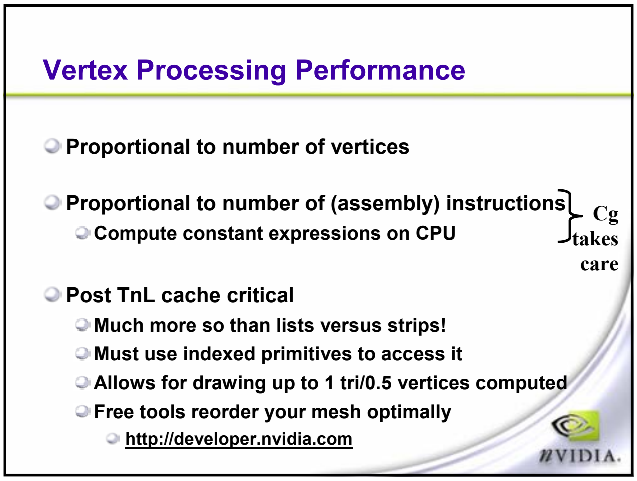
AGP Performance

- Write AGP data in 32 or 64 byte blocks
  - AGP-Write combining needs to read then write
- Avoid reading from graphics data-structures
- Communicate intended use to driver
  - Static versus dynamic vertex buffers or textures
  - Declare data as write-only
  - Placement into video-, AGP-, or system-memory
- Allow vertex buffer renaming (avoid syncs)
  - Use discard/no-overwrite and var/fence



Programmable Vertex Processors

- No connectivity info/no access to neighbors (SIMD)
- 1.5 Billion VECTOR operations/s! (~6B ops/s)
  - IEEE s23e8 32 bit floating point per component
  - "Simple" operations include dot4, mad, sin, pow, lg2
  - Oh yeah, vector swizzles/conditional writes are free
- Post TnL vertex caches: >>100 Million lit tris/s
- Per-vertex data-dependent:
  - Branches, loops
  - Subroutines

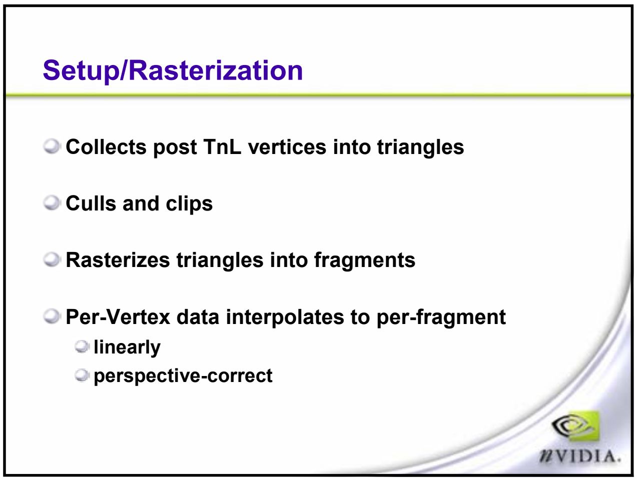


Vertex Processing Performance

- Proportional to number of vertices
- Proportional to number of (assembly) instructions
  - Compute constant expressions on CPU

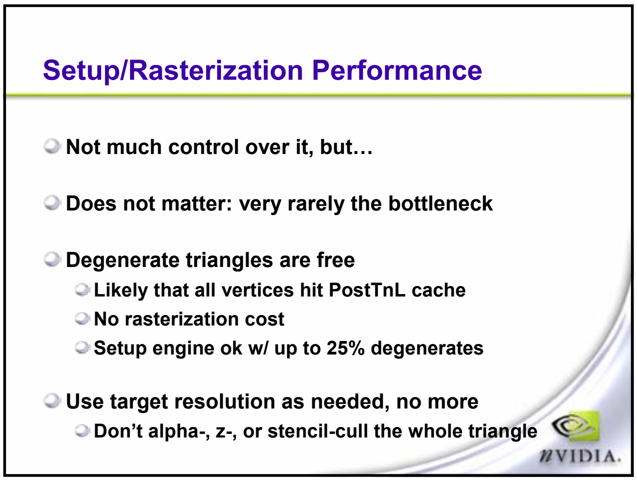
Cg takes care

- Post TnL cache critical
  - Much more so than lists versus strips!
  - Must use indexed primitives to access it
  - Allows for drawing up to 1 tri/0.5 vertices computed
  - Free tools reorder your mesh optimally
    - http://developer.nvidia.com



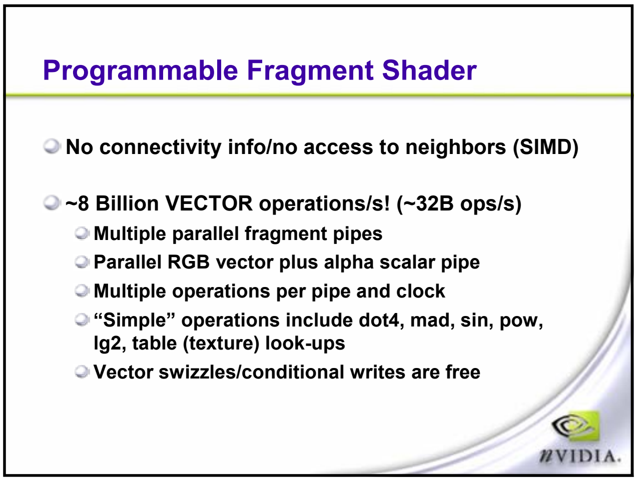
Setup/Rasterization

- Collects post TnL vertices into triangles
- Culls and clips
- Rasterizes triangles into fragments
- Per-Vertex data interpolates to per-fragment
  - linearly
  - perspective-correct



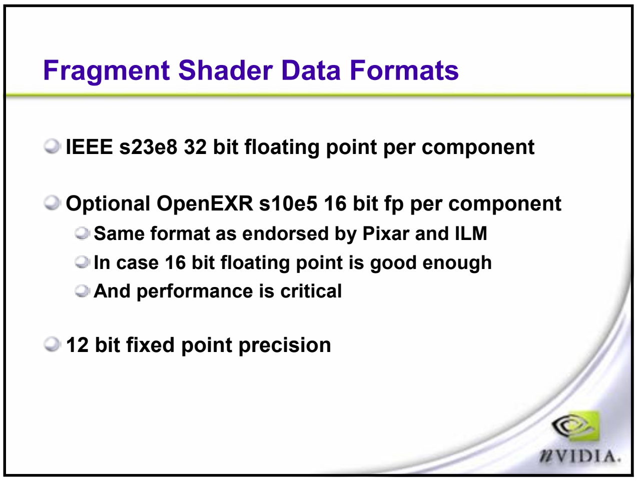
Setup/Rasterization Performance

- Not much control over it, but…
- Does not matter: very rarely the bottleneck
- Degenerate triangles are free
  - Likely that all vertices hit PostTnL cache
  - No rasterization cost
  - Setup engine ok w/ up to 25% degenerates
- Use target resolution as needed, no more
  - Don’t alpha-, z-, or stencil-cull the whole triangle



Programmable Fragment Shader

- No connectivity info/no access to neighbors (SIMD)
- ~8 Billion VECTOR operations/s! (~32B ops/s)
  - Multiple parallel fragment pipes
  - Parallel RGB vector plus alpha scalar pipe
  - Multiple operations per pipe and clock
  - “Simple” operations include dot4, mad, sin, pow, lg2, table (texture) look-ups
  - Vector swizzles/conditional writes are free



Fragment Shader Data Formats

- IEEE s23e8 32 bit floating point per component
- Optional OpenEXR s10e5 16 bit fp per component
  - Same format as endorsed by Pixar and ILM
  - In case 16 bit floating point is good enough
  - And performance is critical
- 12 bit fixed point precision

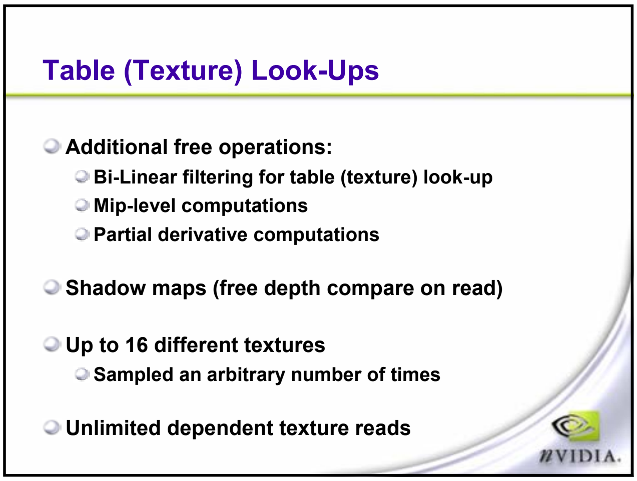
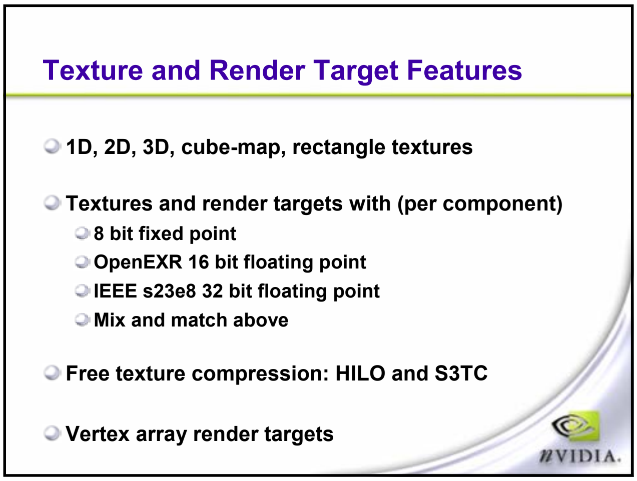


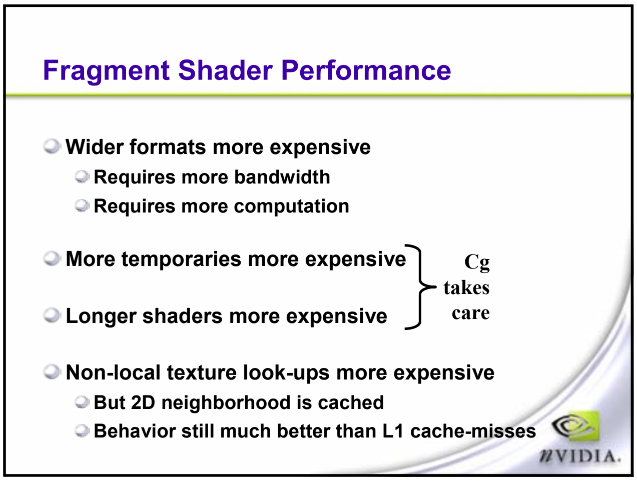
Table (Texture) Look-Ups

- Additional free operations:
  - Bi-Linear filtering for table (texture) look-up
  - Mip-level computations
  - Partial derivative computations
- Shadow maps (free depth compare on read)
- Up to 16 different textures
  - Sampled an arbitrary number of times
- Unlimited dependent texture reads



Texture and Render Target Features

- 1D, 2D, 3D, cube-map, rectangle textures
- Textures and render targets with (per component)
  - 8 bit fixed point
  - OpenEXR 16 bit floating point
  - IEEE s23e8 32 bit floating point
  - Mix and match above
- Free texture compression: HILO and S3TC
- Vertex array render targets

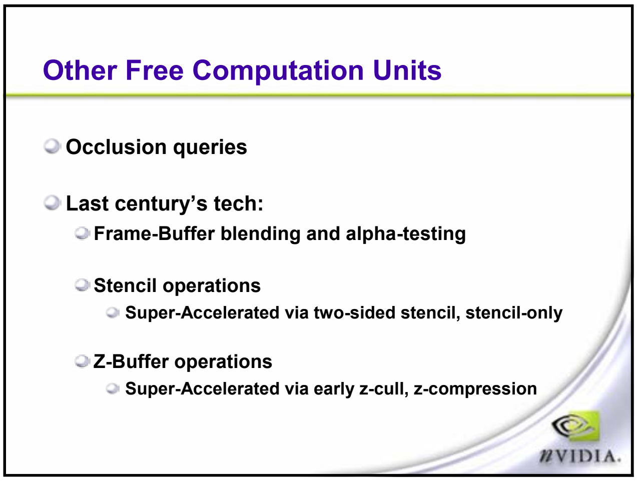


Fragment Shader Performance

- Wider formats more expensive
  - Requires more bandwidth
  - Requires more computation
- More temporaries more expensive
- Longer shaders more expensive

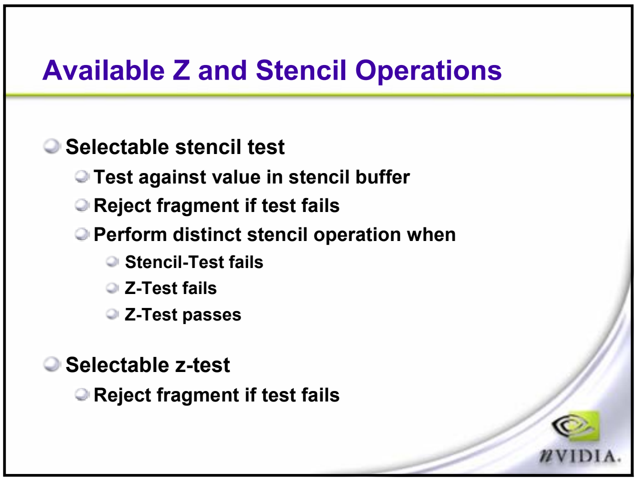
$\left.\begin{array}{l}\text{More temporaries more expensive}\\ \text{Longer shaders more expensive}\end{array}\right\}$ Cg takes care

- Non-local texture look-ups more expensive
  - But 2D neighborhood is cached
  - Behavior still much better than L1 cache-misses



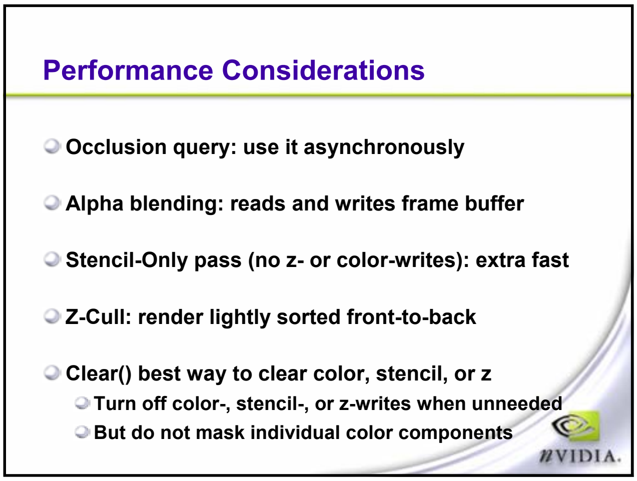
Other Free Computation Units

- Occlusion queries
- Last century's tech:
  - Frame-Buffer blending and alpha-testing
  - Stencil operations
    - Super-Accelerated via two-sided stencil, stencil-only
  - Z-Buffer operations
    - Super-Accelerated via early z-cull, z-compression



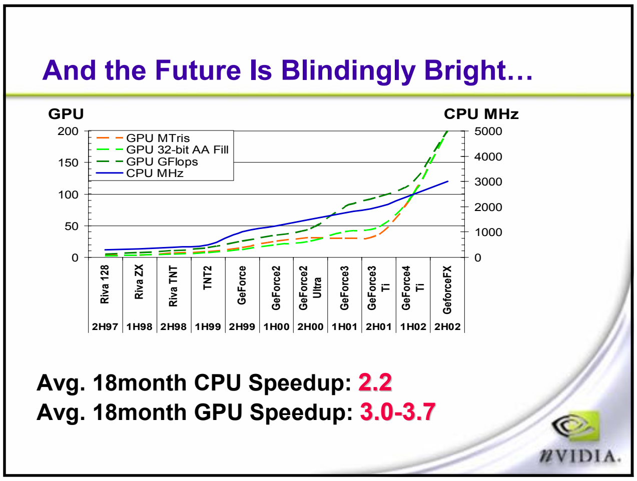
Available Z and Stencil Operations

- Selectable stencil test
  - Test against value in stencil buffer
  - Reject fragment if test fails
  - Perform distinct stencil operation when
    - Stencil-Test fails
    - Z-Test fails
    - Z-Test passes
- Selectable z-test
  - Reject fragment if test fails



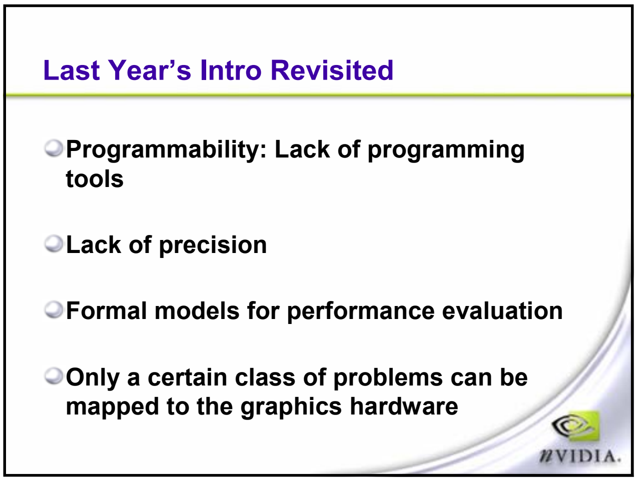
Performance Considerations

- Occlusion query: use it asynchronously
- Alpha blending: reads and writes frame buffer
- Stencil-Only pass (no z- or color-writes): extra fast
- Z-Cull: render lightly sorted front-to-back
- Clear() best way to clear color, stencil, or z
  - Turn off color-, stencil-, or z-writes when unneeded
  - But do not mask individual color components



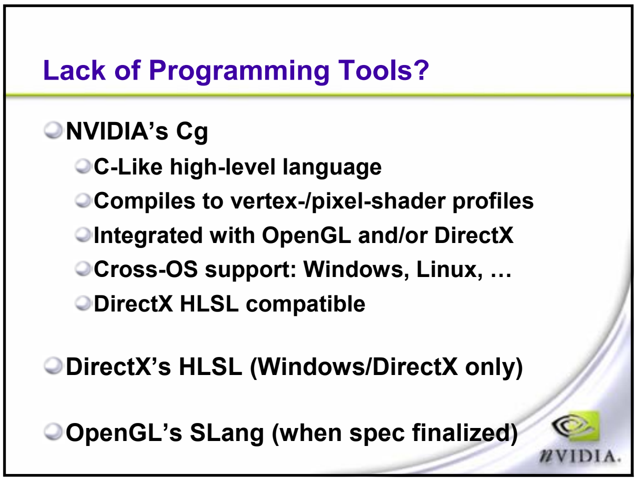
And the Future Is Blindingly Bright…

Avg. 18month CPU Speedup: 2.2
Avg. 18month GPU Speedup: 3.0-3.7



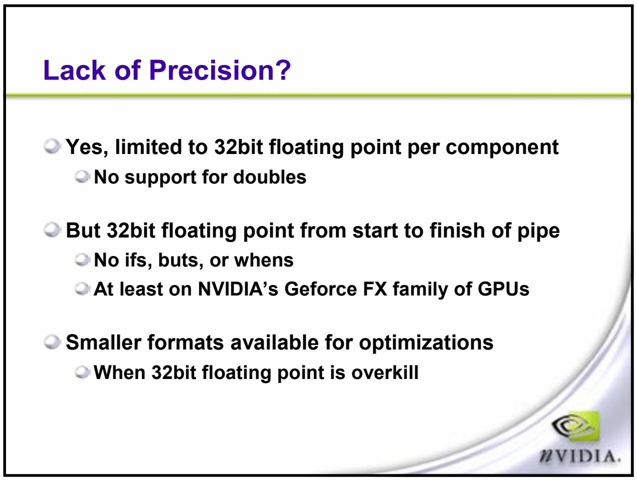
Last Year’s Intro Revisited

- Programmability: Lack of programming tools
- Lack of precision
- Formal models for performance evaluation
- Only a certain class of problems can be mapped to the graphics hardware



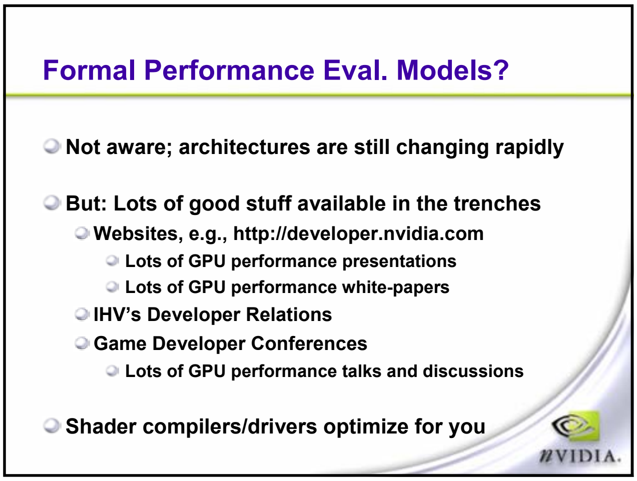
Lack of Programming Tools?

- NVIDIA’s Cg
  - C-Like high-level language
  - Compiles to vertex-/pixel-shader profiles
  - Integrated with OpenGL and/or DirectX
  - Cross-OS support: Windows, Linux, …
  - DirectX HLSL compatible
- DirectX’s HLSL (Windows/DirectX only)
- OpenGL’s SLang (when spec finalized)



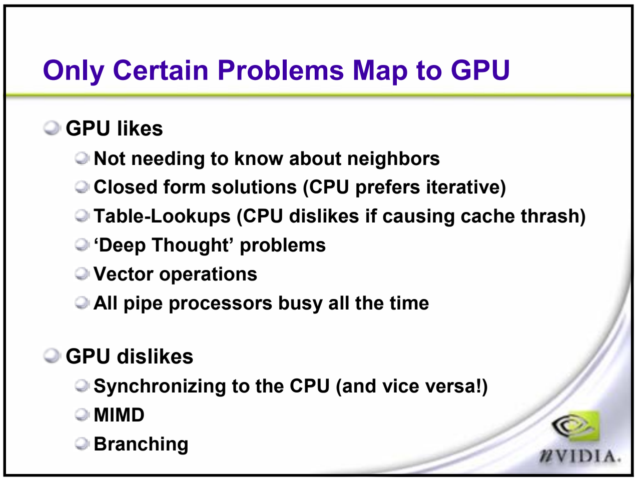
Lack of Precision?

- Yes, limited to 32bit floating point per component
  - No support for doubles
- But 32bit floating point from start to finish of pipe
  - No ifs, buts, or whens
  - At least on NVIDIA’s Geforce FX family of GPUs
- Smaller formats available for optimizations
  - When 32bit floating point is overkill



Formal Performance Eval. Models?

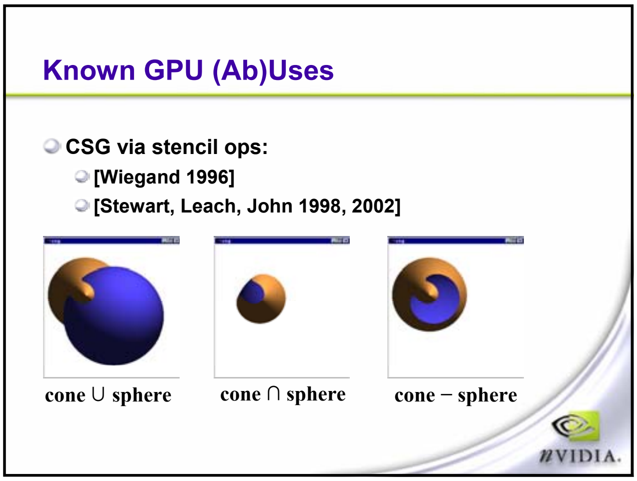
- Not aware; architectures are still changing rapidly
- But: Lots of good stuff available in the trenches
  - Websites, e.g., http://developer.nvidia.com
    - Lots of GPU performance presentations
    - Lots of GPU performance white-papers
  - IHV’s Developer Relations
  - Game Developer Conferences
    - Lots of GPU performance talks and discussions
- Shader compilers/drivers optimize for you



Only Certain Problems Map to GPU

- GPU likes
  - Not needing to know about neighbors
  - Closed form solutions (CPU prefers iterative)
  - Table-Lookups (CPU dislikes if causing cache thrash)
  - ‘Deep Thought’ problems
  - Vector operations
  - All pipe processors busy all the time
- GPU dislikes
  - Synchronizing to the CPU (and vice versa!)
  - MIMD
  - Branching

NVIDIA.



Known GPU (Ab)Uses

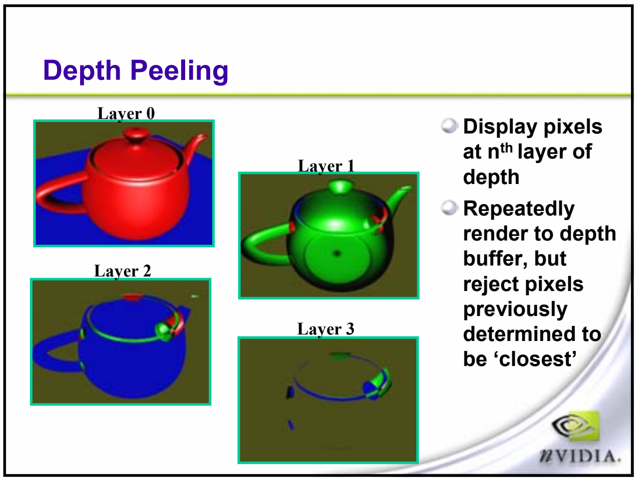
- CSG via stencil ops:
  - [Wiegand 1996]
  - [Stewart, Leach, John 1998, 2002]

cone ∪ sphere

cone ∩ sphere

cone − sphere

NVIDIA.



Depth Peeling

Layer 0

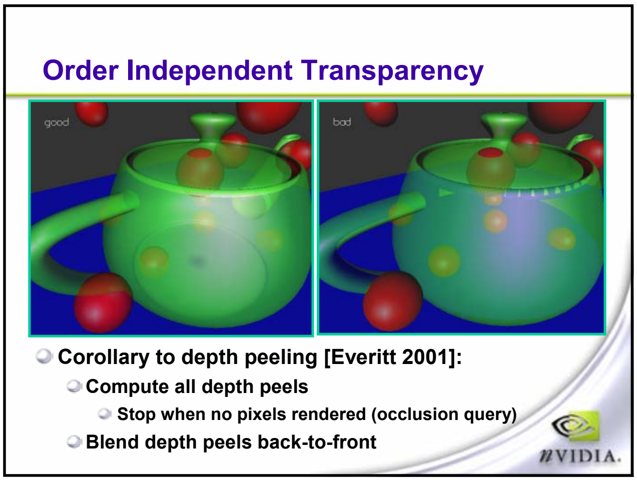
Layer 1

Layer 2

Layer 3

- Display pixels at nth layer of depth
- Repeatedly render to depth buffer, but reject pixels previously determined to be ‘closest’

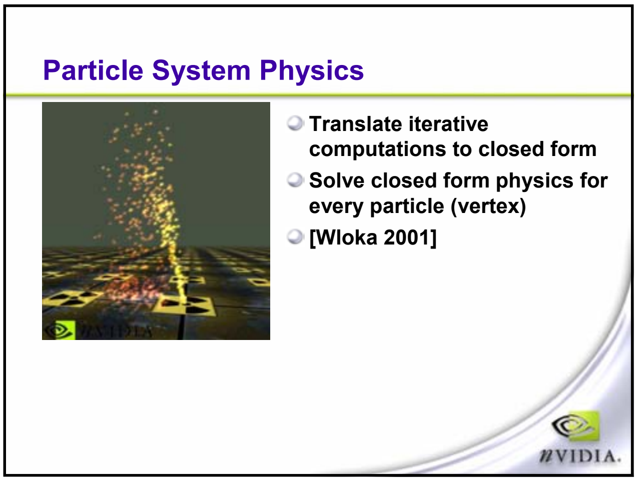
NVIDIA.



Order Independent Transparency

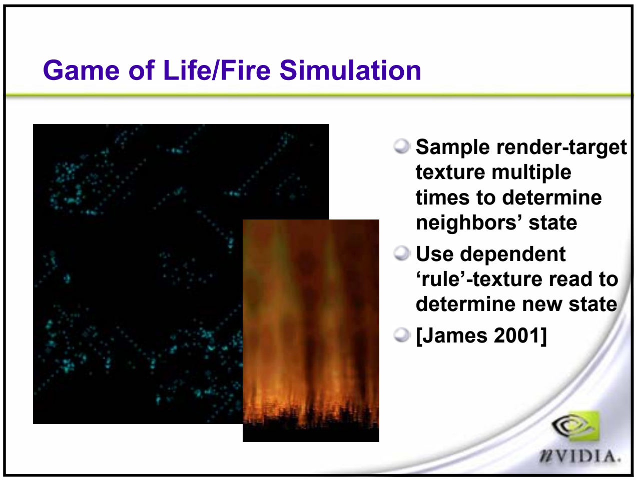
- Corollary to depth peeling [Everitt 2001]:
  - Compute all depth peels
    - Stop when no pixels rendered (occlusion query)
  - Blend depth peels back-to-front

NVIDIA.



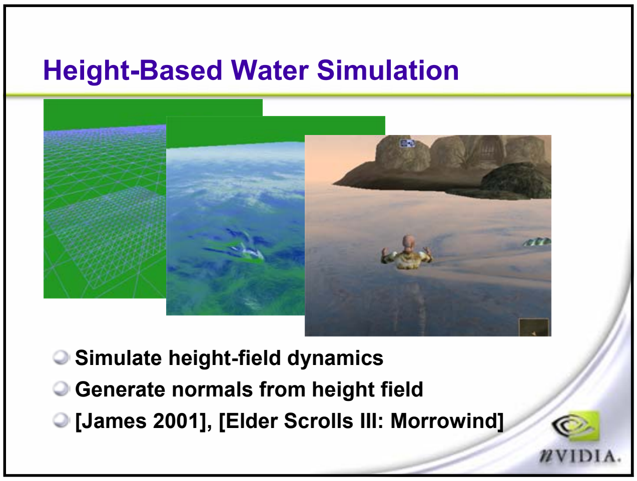
Particle System Physics

- Translate iterative computations to closed form
- Solve closed form physics for every particle (vertex)
- [Wloka 2001]



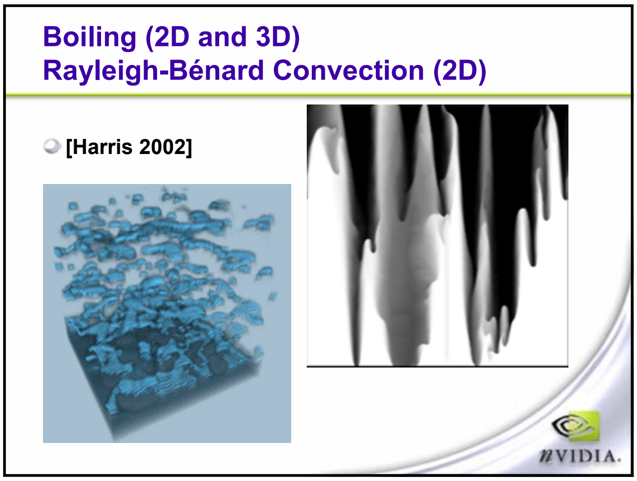
Game of Life/Fire Simulation

- Sample render-target texture multiple times to determine neighbors' state
- Use dependent 'rule'-texture read to determine new state
- [James 2001]



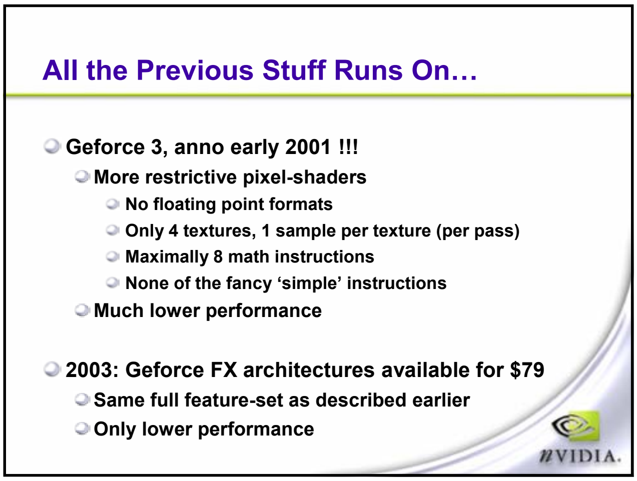
Height-Based Water Simulation

- Simulate height-field dynamics
- Generate normals from height field
- [James 2001], [Elder Scrolls III: Morrowind]



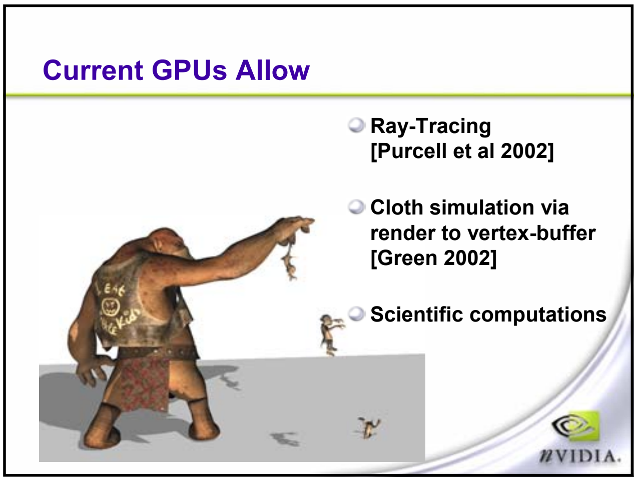
Boiling (2D and 3D)
Rayleigh-Bénard Convection (2D)

- [Harris 2002]



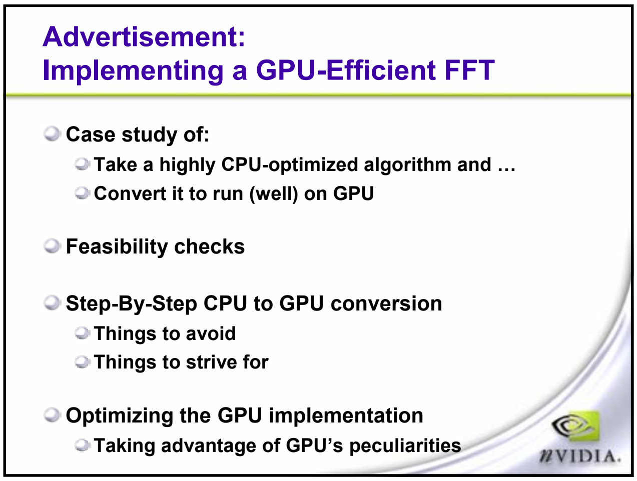
All the Previous Stuff Runs On…

- Geforce 3, anno early 2001 !!!
  - More restrictive pixel-shaders
    - No floating point formats
    - Only 4 textures, 1 sample per texture (per pass)
    - Maximally 8 math instructions
    - None of the fancy 'simple' instructions
  - Much lower performance
- 2003: Geforce FX architectures available for $79
  - Same full feature-set as described earlier
  - Only lower performance



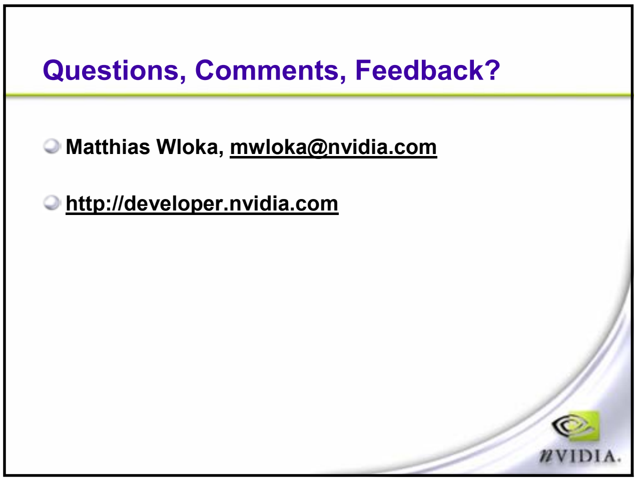
Current GPUs Allow

- Ray-Tracing [Purcell et al 2002]
- Cloth simulation via render to vertex-buffer [Green 2002]
- Scientific computations



Advertisement: Implementing a GPU-Efficient FFT

- Case study of:
  - Take a highly CPU-optimized algorithm and …
  - Convert it to run (well) on GPU
- Feasibility checks
- Step-By-Step CPU to GPU conversion
  - Things to avoid
  - Things to strive for
- Optimizing the GPU implementation
  - Taking advantage of GPU's peculiarities



Questions, Comments, Feedback?

- Matthias Wloka, mwloka@nvidia.com
- http://developer.nvidia.com

Graphics Hardware Functionality for Geometric Computations with OpenGL, Circa Spring 2002

Mark J. Kilgard
NVIDIA Corporation, Austin
mjk@nvidia.com

April 2002

Introduction

This paper offers a whirlwind tour of contemporary graphics hardware functionality, focusing on the task of accelerating geometric computations. NVIDIA's Quadro4 XGL and GeForce4 Ti *G*raphics *P*rocessing *U*nits (GPUs) manifest all the functionality to be discussed. While other GPUs from other vendors may also manifest much identical or similar functionality, I constrain this papers discussion to NVIDIA's current (Spring 2002) top-of-the-line GPU generation because these GPUs are widely regarded as the fastest and most capable GPUs available as of this writing and, pragmatically, my own expertise is limited to these GPUs. I recognize that other GPUs, such as ATI's Radeon 8500, provide similar or identical functionality in many cases.

Again due to the limits of my own expertise, I'll discuss GPU functionality in terms of the OpenGL programming interface. I recognize that Microsoft's Direct3D programming interface exposes similar or identical functionality in most cases. However OpenGL's extension mechanism provides a means to expose NVIDIA's *complete* GPU hardware functionality which in a few cases is not otherwise exposed by the most recent version of Direct3D for the PC, namely DirectX 8.1. For example, general hardware shadow mapping and hardware occlusion queries are available today just through OpenGL today.[1] Additionally, full OpenGL support is available for Linux and Apple's OS X operating system whereas DirectX is only supported on Windows systems so the functionality described is broadly available on a variety of platforms.

Focus on Functionality for Geometric Computations

GPUs sold today have a comparable design and transistor complexity to CPUs. Graphics is a highly parallel process with regular and very pipeline-able algorithms. This combined with the appetite for increasing realistic graphics in markets from scientific visualization to Computer Aided Design (CAD) to 3D video games means that the semiconductor industries ever increasing transistor densities and counts makes graphics hardware ideally positioned to improve in performance at rates consistent with the so-called Moore's Law.

[1] The version of DirectX for Xbox does support these additional features, but the PC version of DirectX 8 does not.

Traditionally, the end result of this increasing graphics hardware horsepower has been rendering images to be displayed to a computer user, whether a CAD designer or teenage video game enthusiast. Typically, each image is rendered to be displayed to the user, and then discarded so that yet another image can be displayed within a fraction of a second to maintain the illusion of animation.

This paper focuses not on the traditional use of graphics hardware for animated rendering but rather the use of graphics hardware functionality for geometric computations. The argument for using GPUs for geometric computations is that graphics hardware is expected to out-strip the performance of conventional CPU hardware for the class of computations that graphics hardware is designed to accelerate. At the same time, the trend in graphics hardware design is towards increasing programmability, rather than mere configurability, of the graphics pipeline. This means that graphics hardware that in the past was over-specialized for conventional 3D rendering can be brought to bear on tasks not conventionally though of as being amenable to graphics hardware acceleration.

The point of this paper is not to present such applications but rather to note the functionalities within contemporary graphics hardware (circa Spring 2002) as embodied by the set of OpenGL extensions supported by NVIDIA's current top-of-the-line GPUs, the NVIDIA GeForce4 Ti and Quadro4 XGL lines.

If geometric computations are to be competitive with CPU algorithms and, in fact, more efficient, it is critical that the graphics hardware is used efficiently. Not only does this paper focus on functionality helpful to implementing geometric computation algorithms using graphics hardware but also highlights the highest performance means to use contemporary graphics hardware.

The remainder of this paper discusses graphics hardware functionality in roughly the order of the graphics hardware pipeline. All the OpenGL extensions cited have specifications that can be found in the OpenGL extension registry [8] or in NVIDIA's collection of formatted OpenGL extension specifications [7].

Efficient Vertex Presentation

OpenGL provides multiple methods for sending vertex information to graphics hardware. Immediate mode uses the classic `glVertex3f`, etc. commands. While convenient, immediate mode vertex transfers typically throttle the vertex transfer speeds due to API overhead and the general inefficiency of transferring per-vertex parameters one parameter at a time. Display lists provide a more efficient means to "batch up" immediate mode commands statically for fast play back. OpenGL 1.1 added support for vertex arrays where the OpenGL implementation is first configured with arrays of per-vertex parameters in main memory, potentially interleaved arrays or separate arrays. Then lists of vertex array indices indicate how to present vertices efficiently to the graphics hardware. The `glDrawArrays` command can be used for sequential indices while the `glDrawElements` command can be passed an array of random indices.

The `EXT_compiled_vertex_array` extension adds the facility to lock a range of indices. When a range of indices is locked with `glLockArraysEXT`, the OpenGL implementation can assume that the vertex array data references by the indicated range of vertices will not change until a matching glUnlockArraysEXT unlocks the range. This mechanism is used by id Software's Quake3 game engine so most OpenGL implementations today implement this extension today very efficiently. Be warned that some OpenGL implementations only optimize vertex array configurations that are very similar or identical to the configurations used by Quake3; NVIDIA's recent drivers provide very general acceleration to this functionality however.

NVIDIA provides an even more optimized variation on vertex arrays through its NV_vertex_array_range extension. Applications seeking to achieve the very best vertex processing rates are recommended to use this extension however the effort may not be worth it unless your application is truly sending millions of vertices per second to be transformed. The vertex array range mechanism provides a way to allocate memory for high-bandwidth transfers to the GPU. This memory is usually Advanced Graphics Port (AGP) memory but can also be local video on the graphics card. Once this memory is allocated, the glVertexArrayRangeNV command configures a range of this memory for highly efficient vertex transfers. When the GL_VERTEX_ARRAY_RANGE_NV client-side enable is enabled, vertex array commands send the requested vertex indices directly to the GPU and the GPU issues high-bandwidth read requests for the required memory. This is in contrast to conventional vertex array calls where when vertex array calls (glDrawArrays or glDrawElements) return, the vertex array memory can be immediately modified meaning that the CPU must immediately copy the vertex data to the GPU.

Note that the vertex array range functionality provides extremely fast vertex transfer rates but only so long as you abide by its rules. The memory allocated for the vertex array range is *uncached* memory meaning that CPU reads from this memory are exceedingly slow relative to conventional cached CPU memory. Also note all vertex array configurations can be efficiently pulled by the CPU. These various restrictions are noted the NV_vertex_array_range OpenGL extension.

Another difficulty exposed to applications that choose to use the vertex array functionality is how to update vertex data within the vertex array range dynamically and efficiently. There is no locking mechanism for the vertex array range memory; it is simply up to the application to be sure not to access memory corresponding to "in flight" vertex array indices. This is facilitated by the NV_fence OpenGL extension that provides a way to know when the hardware has completed all the commands prior to a glSetFenceNV command. Vertex array range memory accessed by commands prior to a finished fence (determined with glFinishFenceNV) can be modified, assuming of course that these indices were not also issued subsequent to the fence.

The original NV_vertex_array_range extension specified that enabling and disabling the vertex array range with the GL_VERTEX_ARRAY_RANGE_NV client-side enable caused the hardware to stall. The later NV_vertex_array_range2 extension introduced a second enable enumerant GL_VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV that allows an application to enable and disable the vertex array range without an implicit vertex array range flush that causes the hardware to stall. Use the later enable if your application wishes to mix vertex array range and non-vertex array range vertex array transfers.

Direct3D permits a similar level of performance through its use of vertex buffers and streams though it lacks a synchronization mechanism as efficient as the fence mechanism.

Vertex Programs

Vertex programs (known as *vertex shaders* in Direct3D) provide a means for a graphics application to specify a linear sequence of floating-point instructions on a per-vertex basis. This is in contrast to the conventional Transformation & Lighting (T&L) operations provided by OpenGL. This is particularly useful in the context of geometric computations where the required per-vertex computations for an algorithm to be implemented may not readily map to the conventional per-vertex position transformation, lighting, and texture coordinate generation operations supported by OpenGL.

Vertex programs can play an important role in off-loading vertex-level computations from the CPU onto the GPU. As a rough rule of thumb, a GeForce4 Ti or Quadro4 XGL can execute two vertex program instructions per clock (twice the rate of the GeForce3 that introduced the functionality) and these GPUs are typically clocked in the neighborhood of 300 million clocks/second. These instructions are operate on four-component floating-point vectors so a `DP4` instruction is the equivalent of 4 floating-point multiplies and then three floating-point additions to sum up the 4 vector multiply results. Vertex programs require no overhead for looping over each vertex, data loading or conversion, or data storage. This makes the effective floating-point hardware utilization (the percentage of the time that the floating-point hardware is actually busy) substantially higher than a CPU.

There are limitations to the vertex program approach. A vertex program executes on each vertex *in isolation* so there is no knowledge of adjacent vertices. There is no support for conditional branching or looping. The resulting transformed vertices are not available to the CPU; the transformed vertices are immediately consumed by the hardware rasterizer.

Comprehensive coverage of vertex programs is beyond the scope of this white paper. See the SIGGRAPH 2001 paper "A User Programmable Vertex Engine" [1] and the `NV_vertex_program` OpenGL extension for more details.

Rasterization

Current GPUs setup geometric primitives (triangles, quads, lines, and points) at amazing rates. The GeForce4 Ti and Quadro4 XGL GPUs can setup a primitive every 5 clocks. Prior high-end multi-chip graphics hardware would provide a relatively large FIFO between vertex processing and rasterization so that primitives could "pile up" if the rasterizer ever fell behind. This allows the vertex processing and rasterization workloads to balance each other out over a period of time. With the advent of single-chip GPU implementations, there is no longer a FIFO required to transfer data from a vertex processing chip to a rasterization chip; it all happens within the same chip. Rather than waste transistors on a FIFO, these same transistors can be invested in simply making the setup process of rasterization so fast that it can "keep up" with the rate that primitives can be generated by the vertex processor and primitive assembly. While few applications can drive the 60 million theoretical triangle setup rate of a GeForce4 Ti, this high rate is available if needed.

Geometric algorithms often require back-face culling to draw all front-facing, then all back-facing primitives, or simply as an optimization to avoid rendering non-visible polygons of closed models. Back face culling is "free" in NVIDIA's GPU line; this was not always true of older graphics hardware.

Efficient early depth buffer culling (so-called *Z Cull*) functionality makes it advantageous to render your scene in coarse front-to-back order when depth buffering. The hardware can reject fragments at a much higher rate when depth occlusion can be determined earlier in the graphics pipeline, in particular, prior to texturing. Discarding such fragments early conserves precious memory bandwidth for visible pixels. Geometric algorithms should definitely exploit this efficient early depth buffer culling.

Alpha testing and *depth replace* (see the subsequent Texture Shader section) can interfere with efficient early depth buffer culling because texturing and alpha testing occurs prior to depth buffering in the OpenGL fragment pipeline. Texturing can change the alpha value and thereby change whether the alpha test fails or succeeds. Similarly if the depth value is generated by a depth replace operation that depends on texture results, the early depth buffer culling prior to texturing must be disabled. Keep this in mind when using either alpha testing or depth replace. Stencil testing can create similar ambiguities that undermine early depth buffer culling because the stencil test occurs before the depth test in the OpenGL fragment pipeline.

Depth Clamping

Near and far clip planes are the bane of most 3D graphics programmers. The near clip plane clips geometry that is close to the eye while the far clip plane clips away geometry in the distance. These clip planes are conventionally required to ensure a reasonable range of depth buffer precision and to make sure that all fragment depth values are representable within the depth buffer.

The GeForce 3, GeForce4 Ti, and Quadro4 XGL GPUs support a capability known as *depth clamping* exposed by the `NV_depth_clamp` extension. When `GL_DEPTH_CLAMP_NV` is enabled, the near and far clip planes are effectively disabled (fragments with a window-space *w* values less than or equal to zero, i.e., fragments behind the viewer, are still discarded). Interpolated fragment depth values either larger or smaller than range of depth values provided by the current depth range are *clamped* to within the depth range.

Geometric computations that only require depth values within a given range but break if primitives are clipped by the near or far clip plane can benefit from depth clamping. An example of one such algorithm is the robust stenciled shadow volume algorithm described by Cass Everitt and myself [4].

Conventional Texture Targets

Textures can be thought of as arbitrary multi-dimensional memory accesses that benefit from texture filtering. Many clever and efficient graphics hardware algorithms can be constructed by pre-computing complex functions in textures for subsequent processing during a rendering pass. OpenGL 1.3 supports 1D, 2D, 3D, and cube map texture accesses.

Cube maps are new with OpenGL 1.3. Cube maps in particular can be useful as a means of constructing a function of an un-normalized vector. For example, an un-normalized vector direction can be computed per-vertex (perhaps by a vertex program or `GL_OBJECT_LINEAR` or `GL_EYE_LINEAR` texture coordinate generation). Subsequently, these vectors can be interpolated and then used to access a so-called "normalization cube map" [6] that supplies a normalized version of the vector that can be used for per-fragment lighting or other operations during fragment coloring.

Texture Rectangles

Another useful texture type (*target* in OpenGL terminology) is the texture rectangle introduced by NVIDIA's `NV_texture_rectangle` extension. Conventional texture mapping uses normalized texture coordinates normalized to the [0,1] range prior to accessing the texture image and each texture image must have power-of-two dimensions. The texture rectangle target (`GL_TEXTURE_RECTANGLE_NV`) is like a 2D texture target except that the texture image is not restricted to power-of-two dimensions. For example, a 23x59 texture image would work just fine. Additionally, the texture coordinate range is the [0,*width*]x[0,*height*] range rather than a normalized range. Moreover, borders, the `GL_REPEAT` wrap mode, and mipmapping are not supported for texture rectangles.

In practice, texture rectangles are very useful for image processing tasks or re-using the results from a previous frame buffer rendering as a texture. The texture coordinates for a texture rectangle are still projective. One application for this is configuring window-space texture coordinate generation. This allows a texture rectangle with a one-to-one correspondence of its texels to frame buffer pixels. By copying some intermediate result of a rendering result into a texture rectangle and then using window-space texture coordinate generation, these intermediates can be re-used in subsequent passes.

The `NV_texture_rectangle` extension is supported by all GeForce GPUs.

Shadow Mapping

The OpenGL Architectural Review Board (ARB) has just recently (February 2002) standardized official ARB extensions for shadow mapping. These are the `ARB_depth_texture` and `ARB_shadow` extensions. The former provides new texture formats for textures containing depth components. The later provides a new texture filtering mode for "percentage closer" filtering. This filtering scheme compares the interpolated *R* texture coordinate to a depth texture's sampled depth values at each texture sample and then weights these comparisons to generate a final filtered texel.

This functionality is based on the prior proprietary `SGIX_depth_texture` and `SGIX_shadow` extensions, first implemented by SGI on the RealityEngine and InfiniteReality graphics hardware. NVIDIA also supports these extensions on GeForce3, GeForce4 Ti, and Quadro4 XGL GPUs. NVIDIA has recently also implemented the official ARB extensions (the two extensions are very similar; the ARB extension adds only minor new functionality) [3].

Do not let the *shadow* name fool you. These extensions are useful in other contexts besides shadows. Shadow mapping can be thought of as a read-only depth test with better filtering. Geometric computations that require multiple depth buffers can use shadow mapping to simulate multiple depth buffers. Cass Everitt's *Order Independent Transparency* algorithm [2] is a good example of this kind of innovative use of shadow mapping.

Texture Shader

The GeForce3, GeForce4 Ti, and Quadro4 XGL GPUs from NVIDIA also support functionality known as the texture shader for more general texture lookups. The texture shader consists of a set of 4 texture shader stages. Among the functionality possible are dependent texture accesses where the result of one texture access is used as the texture coordinate set of a second texture access. The texture shader functionality is fully described in the `NV_texture_shader`, `NV_texture_shader2`, and `NV_texture_shader3` OpenGL extension specifications. Conventional OpenGL texture lookups are performed (using the hierarchy of texture enables) unless the `GL_TEXTURE_SHADER_NV` enable is set, in which case, the texture shader functionality is used. The details of the texture shader functionality are beyond the scope of this article so see the texture shader OpenGL extensions for details [1].

One capability specific texture shader capability is of interest for those interested in implementing geometric computations. The `GL_DOT_PRODUCT_DEPTH_REPLACE_NV` and `GL_DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV` texture shader operations (used in conjunction with prior texture shader operations to access a texture and, in the non-affine case, to compute a second dot product) can replace a fragment's interpolated depth value with a new depth value computed through a combination of the texture access and one or two dot products. In the case of the non-affine depth replace operation, the new depth value is the result of a division of the two dot products so that depth values can be generated with correct projective properties required for perspective views.

One application of this feature is so-called "Z correct" displacement mapping. Unfortunately, displacement mapping in only the window-space Z direction is not particularly useful except for better rendering interfaces between, say, terrain and water (see NVIDIA's "tide pool" demo).

Other applications in the fields of image-based rendering or geometric computations promise to be much more interesting. Cass Everitt's *Order Independent Transparency* algorithm [2] is an example of a novel algorithm that uses the depth replace functionality.

Register Combiners

Conventional OpenGL fragment coloring uses zero or more texture environment applications, one for each enabled texture, and then a color sum and fog application. OpenGL 1.3 provides considerably more flexible texture environment functions than previous versions of OpenGL. However, OpenGL 1.3 still requires a sequential model of texture environment application.

In contrast, the `NV_register_combiners` extension (available on all GeForce GPUs; and augmented by the `NV_register_combiners2` extension on GeForce3, GeForce4 Ti, and Quadro4 XGL GPUs) provides a considerably more configurable system for fragment coloring compared to OpenGL 1.3. Basic register combiners provide a register model that supports a variety of signed 3-component (RGB) and 1-component (alpha) math operations including dot products. Input values are the interpolated primary (diffuse) and secondary (specular) colors, each filtered texture unit color result, the constant fog color and per-fragment fog factor, and a set of RGBA constants. Additionally, there is a "free" final combiner to generate the final RGBA fragment color.

The original GeForce functionality for register combiners had two global RGBA constants and up to two general combiners stages. The GeForce3 (via the `NV_register_combiners2` extension) introduced two RGBA constants per stage and up to eight general combiner stages.

Both the texture shader and register combiners functionality have cumbersome APIs due to the plethora of available options and configurations. NVIDIA has made available a library known as *NVParse* [12] that makes it substantially easier to program texture shader and register combiners functionality because you can describe your desired configuration with a succinct, human-readable textural description rather than dozens of OpenGL API calls. I highly recommend *NVParse*.

Stencil Testing

Stencil testing has been widely used since its inception for various geometric computations included Constructive Solid Geometry (CSG), capping, shadow volumes, etc. DirectX 6 introduced two new stencil operations, *modulo increment* and *modulo decrement*, that wrap rather than clamp when at the maximum and zero stencil values respectively. These are exposed in OpenGL through the `EXT_stencil_wrap` extension. Various situations are where stencil increment & decrement clamping caused overflow situations can be alleviated by using the new wrapping increment & decrement operations.

Future graphics hardware is likely to support two-sided stencil testing hardware where front- and back-facing primitives can have distinct, independent stencil state.

Both two-sided stencil testing and the wrapping increment & decrement operations are useful for stenciled shadow volume rendering [4].

Blending and Logic Ops

Conventional OpenGL 1.0 supports basic frame buffer blending and lacks logic op support for color buffers (logic ops are supported for color index buffers in OpenGL 1.0). OpenGL 1.1 added logic op support for color buffers. OpenGL 1.2 specified the `ARB_imaging` subset that includes support for subtractive, minimum, and maximum blending as well as a constant blend color. These additional blend modes are sometimes exposed by various OpenGL extensions through the EXT_blend_substract, EXT_blend_minmax, and EXT_blend_color extensions.

These additional blend modes can be useful in the context of geometric computations (and image processing). For example, subtractive and additive blending can use the color buffer as four distinct counters. Update of each "counter" can be independently controlled by glColorMask. Additionally, the color logic op can be used to treat the color buffer as a bit vector. A typical 32-bit RGBA frame buffer would allow parallel OR, XOR, AND, etc. operations on frame buffer values being treated as bit vectors.

Occlusion Queries

Geometric computations often involve looping over a particular rendering operation until no more pixels are drawn. Unfortunately, most existing hardware is not good at reporting when pixels are rendered in a given rendering pass. This forces algorithms that require such loops to make a worst-case assumption and likely loop rendering many more times than is actually required.

The HP_occlusion_test and NV_occlusion_query extensions [9] provide a means to determine if any pixels were rendered within a given interval of OpenGL rendering commands. The HP extension gets the job done but only a single occlusion test can be active at one time and the results are returned synchronously. The NV extension allows a large number of occlusion queries to be active and the queries can be retired asynchronously. This means that if you issue several dozen occlusion queries, by the time you go to query the first of your occlusion queries, the result can be returned while the other queries are still active. You only need to asynchronously block to wait on the result of a query if the interval of commands for the query have not yet complete. If the commands have completed, there is no wait.

Additionally, the NV extension returns a count of the number of rendered pixels within the occlusion query interval rather than merely a Boolean value (the HP extension returns only a Boolean).

Pixel Buffers, a.k.a. Pbuffers

Geometric computations with graphics hardware are often hindered by the fact that the displayed frame buffer is a volatile surface. Rendering the results of geometric computations into the frame buffer may be very fast, but other windows, menus, etc. overlap your window, the frame buffer results may be corrupted.

Pixel buffers or *pbuffers* [14] provide a means in OpenGL to allocate off-screen, potentially non-volatile frame buffer memory for rendering. Both WGL (the Microsoft Windows interface for OpenGL) and GLX (the X Window System interface for OpenGL) support a pbuffer extension. See the GLX_ARB_pbuffer and WGL_ARB_pbuffer OpenGL extensions. Developers of algorithms for geometric computations involving graphics hardware are encouraged to use pbuffers.

Render to Texture Support

While OpenGL 1.1 provides fast commands to copy frame buffer data to textures, this still requires an extra copy of pixels. If a pbuffer could be used directly as a texture, that would save this extra copy and improve rendering performance. Geometric computations regularly involve the requirement to reuse frame buffer rendering results as textures so this *render to texture* support is very helpful. The `GLX_ARB_render_texture` (for X) and `WGL_ARB_render_texture` (for Windows) provide the programming interfaces required for rendering into pbuffers and then using the results as a texture. These extensions provide the capability to support both rendering into 2D and cube map textures.

Note that it is not possible to use a pbuffer as a texture when you are actually rendering into that pbuffer.

NVIDIA has provided further extensions [15] that permit pbuffers to be used as texture rectangles and support depth-component textures (for shadow mapping). See NVIDIA's `WGL_NV_render_depth_texture` and `WGL_NV_render_texture_rectangle` extension specifications.

Because this render to texture support is relatively new, whitepapers detailing how to use the functionality are not yet available at the time of this writing. Please check the NVIDIA Developer web site though.

Automatic Mipmap Generation

NVIDIA's OpenGL driver supports the `SGIS_generate_mipmap` extension that provides automatic generation of mipmap levels given a base texture level. The graphic hardware's fast bilinear down-sampling hardware is used to construct the additional mipmap levels. No data must be read back to the CPU when building these GPU-constructed mipmaps so the process is very efficient.

Mipmaps can be generated automatically for texture images specified explicitly (i.e. via `glTexImage2D`, etc.; indeed, NVIDIA's driver support automatically generating mipmap levels is often faster than the conventional `gluBuild2DMipmaps` routine), texture images copied from the frame buffer (i.e. via `glCopyTexSubImage2D`, etc.), or pbuffers used with render-to-texture support. This extension is very easy to use; you simply set the `GL_GENERATE_MIPMAP_SGIS` texture parameter for a texture object to `GL_TRUE`. When true, whenever the base level of the mipmap is specified, the other mipmap levels are automatically generated.

Conclusions

This whirlwind tour of contemporary graphics functionality for geometric computations just scratches the surface of these various topics. Please consult the references and the mentioned OpenGL extension specifications for more information.

Keep in mind that the functionality described reflects merely today's functionality for high-performance, reasonably-priced GPUs. Future GPUs promise more functionality, particularly programmable functionality, at better still price/performance points.

I look forward to seeing the GPU-accelerated algorithms for geometric computations that are developed using current and future GPU functionality.

References

[1] Sebastien Domine and John Spitzer. *Texture Shaders*. http://developer.nvidia.com/view.asp?IO=texture_shaders

[2] Cass Everitt. *Interactive Order Independent Transparency*. Printed in these course notes. http://developer.nvidia.com/view.asp?IO=Interactive_Order_Transparency

[3] Cass Everitt, Ashu Rege, and Cem Cebenoyan. Hardware Shadow Mapping. http://developer.nvidia.com/view.asp?IO=hwshadowmap_paper

[4] Cass Everitt and Mark Kilgard. Practical and Robust Stenciled Shadow Volumes. Printed in these course notes. http://developer.nvidia.com/view.asp?IO=robust_shadow_volumes

[5] Erik Lindholm, Mark Kilgard, and Henry Moreton. *A User-Programmable Vertex Engine*, SIGGRAPH 2001 Proceedings. http://developer.nvidia.com/view.asp?IO=SIGGRAPH_2001

[6] Mark Kilgard. *Practical and Robust Bump-mapping Technique for Today's GPUs*. http://developer.nvidia.com/view.asp?IO=Practical_Bumpmapping_Tech

[7] *NVIDIA OpenGL Extensions Specifications*. Mark Kilgard, editor. http://developer.nvidia.com/view.asp?IO=nvidia_opengl_specs. March 2001.

[8] OpenGL Extension Registry. http://oss.sgi.com/projects/ogl-sample/registry

[9] Ashu Rege, *Occlusion (HP and NV Extensions).* http://developer.nvidia.com/view.asp?IO=gdc_occlusion

[10] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification* (Version 1.3). www.opengl.org

[11] John Spitzer. *Texture Compositing with Register Combiners*. http://developer.nvidia.com/view.asp?IO=registercombiners

[12] John Spitzer. *NVParse*. http://developer.nvidia.com/view.asp?IO=nvparse

[13] John Spitzer and Cass Everitt, *Using* `GL_NV_vertex_array_range` *and* `GL_NV_fence` *on GeForce Products and Beyond*, http://developer.nvidia.com/view.asp?IO=Using_GL_NV_fence

[14] Chris Wynn, *Using P-Buffers for Off-Screen Rendering in OpenGL.* http://developer.nvidia.com/view.asp?IO=PBuffers_for_OffScreen

[15] Chris Wynn, OpenGL Render-to-Texture. http://developer.nvidia.com/view.asp?IO=gdc_oglrtt

Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering

Cass Everitt and Mark J. Kilgard
March 12, 2002
NVIDIA Corporation, Copyright 2002
Austin, Texas

ABSTRACT

Twenty-five years ago, Crow published the shadow volume approach for determining shadowed regions in a scene. A decade ago, Heidmann described a hardware-accelerated stencil buffer-based shadow volume algorithm.

However, hardware-accelerated stenciled shadow volume techniques have not been widely adopted by 3D games and applications due in large part to the lack of robustness of described techniques. This situation persists despite widely available hardware support. Specifically what has been lacking is a technique that robustly handles various "hard" situations created by near or far plane clipping of shadow volumes.

We describe a robust, artifact-free technique for hardware-accelerated rendering of stenciled shadow volumes. Assuming existing hardware, we resolve the issues otherwise caused by shadow volume near and far plane clipping through a combination of (1) placing the conventional far clip plane “at infinity”, (2) rasterization with infinite shadow volume polygons via homogeneous coordinates, and (3) adopting a *zfail* stencil-testing scheme. *Depth clamping*, a new rasterization feature provided by NVIDIA's GeForce3 & GeForce4 Ti GPUs, preserves existing depth precision by not requiring the far plane to be placed at infinity. We also propose *two-sided stencil testing* to improve the efficiency of rendering stenciled shadow volumes.

Keywords

Shadow volumes, stencil testing, hardware rendering.

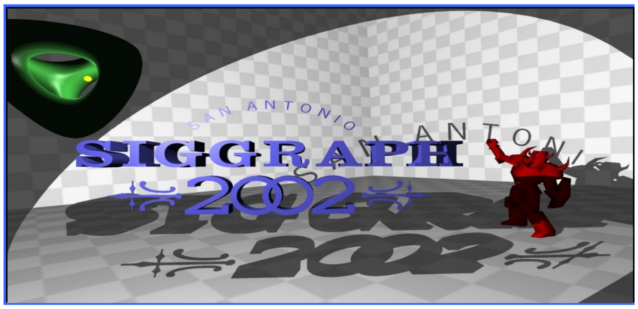
1. INTRODUCTION

Crow’s shadow volume approach [10] to shadow determination is twenty-five years old. A shadow volume defines a region of space that is in the shadow of a particular occluder given a particular ideal light source. The shadow test determines if a given point being tested is inside the shadow volume of any occluder. Hardware stencil testing provides fast hardware acceleration for shadow determination using shadow volumes. Despite the relative age of the shadow volume approach and the widespread availability of stencil-capable graphics hardware, use of shadow volumes in 3D games and applications is rare.

We believe this situation is due to the lack of a practical and robust algorithm for rendering stenciled shadow volumes. We propose here an algorithm to address this gap. Our algorithm is practical because it requires only features available in OpenGL 1.0. The algorithm is robust because shadow volume scenarios that vexed previous algorithms, such as a light within an open container, are handled automatically and correctly.

We focus on robustly solving the problem of hardware-accelerated stenciled shadow volume rendering for a number of reasons, many noted by other authors [9][10][11][19]:

- Shadow volumes provide omni-directional shadows.



- Shadow volumes automatically handle self-shadowing of objects if implemented correctly.
- Shadow volumes perform shadow determination in window space, resolving shadow boundaries with pixel accuracy (or sub-pixel accuracy when multisampling is available).
- Lastly, the fundamental stencil testing functionality required for hardware-accelerated stenciled shadow volumes is now ubiquitous due to the functionality’s standardization by OpenGL 1.0 (1991) and DirectX 6 (1998) respectively. It is near impossible to purchase a new PC in 2002 without stencil testing hardware.

Stenciled shadow volumes have their limitations too. Shadow volumes model ideal light sources so the resulting shadow boundaries lack soft edges. Shadow volume techniques require polygonal models. Unless specially handled, such polygonal models must be closed (2-manifold) and be free of non-planar polygons. Silhouette computations for dynamic scenes can prove expensive. Stenciled shadow volume algorithms are inherently multi-pass. Rendering shadow volumes can consume tremendous amounts of pixel fill rate.

2. PREVIOUS WORK

2.1 Pre-Stencil Testing Work

Crow [10] first published the shadow volume approach in 1977. Crow recognizes that the front- or back-facing orientations of consistently rendered shadow volume polygons with respect to the viewer indicate enters into and exits out of shadowed regions. Crow also recognizes that some care must be taken to determine if the viewer’s eye point is within a shadow volume.

Crow’s formulation fundamentally involves walking a pixel’s view ray originating at the eye point and counting the number of shadow volume enters and exits encountered prior to the first visible rasterized fragment.

Brotman and Badler [8] in 1984 adapted Crow’s shadow volume approach to a software-based, depth-buffered, tiled renderer with deferred shading and support for soft shadows through numerous light sources all casting shadow volumes.

Pixel-Planes [13] in 1985 provides hardware support for shadow volume evaluation. In contrast to Crow’s original ray walking approach, the Pixel-Planes algorithm relies on determining if a pixel is within an infinite polyhedron defined by a single occluder triangle plane and its three shadow volume planes. This determination is made for every pixel and for every occluder polygon in the scene. Each "point within a volume" test is computed by evaluating the corresponding set of plane equations.

Pixel-Planes is a unique architecture because the area of a rasterized triangle in pixels does not affect the triangle's rasterization time. Otherwise, the algorithm's evaluation of *every* per-triangle shadow volume plane equation at *every* pixel would be terribly inefficient.

Bergeron [3] in 1986 generalizes Crow's original shadow volume approach. Bergeron explains how to handle open models and models containing non-planar polygons properly. Bergeron explicitly notes the need to close shadow volumes so that a correct initial count of how many shadow volumes the eye is within can be computed.

Fournier and Fussell [12] in 1988 discuss shadow volumes in the context of frame buffer computations. In their computational model, each pixel in a frame buffer maintains a depth value and shadow depth count. Fournier and Fussell's frame buffer computation model lays the theoretical foundations for subsequent hardware stencil buffer-based algorithms.

2.2 Stencil Testing-Based Work

2.2.1 The Original Approach

Heidmann [14] in 1991 describes an algorithm for using the then-new stencil buffer support of SGI's VGX graphics hardware [1]. Heidmann recognizes the problem of stencil buffer overflows and demonstrates combining contributions from multiple light sources with the accumulation buffer to simulate soft shadows.

Heidmann's approach is a multi-pass rendering algorithm. First, the color, depth, and stencil buffers are cleared. Second, the scene is drawn with only ambient and emissive lighting contributions and using depth testing for visibility determination. Now the color and depth buffers contain the color and depth values for the closest fragment rendered at each pixel. Then shadow volume polygons are rendered into the scene but just updating stencil.

Front-facing polygons update the frame buffer with the following OpenGL per-fragment operations (for brevity, we drop the `gl` and GL prefixes for OpenGL commands and tokens):

```
Enable(CULL_FACE);          // Face culling enabled
CullFace(BACK);             //  to eliminate back faces

ColorMask(0,0,0,0);         // Disable color buffer writes
DepthMask(0);               // Disable depth buffer writes
StencilMask(~0);            // Enable stencil writes

Enable(DEPTH_TEST);         // Depth test enabled
DepthFunc(LEQUAL);          //  less than or equal

Enable(STENCIL_TEST);       // Stencil test enabled
StencilFunc(ALWAYS,0,~0)    //  always pass
StencilOp(KEEP,KEEP,INCR);  //  increment on zpass
```

Similarly, back-facing polygons update the frame buffer with the following OpenGL state modifications:

```
CullFace(FRONT);             // Now eliminate front faces

StencilOp(KEEP,KEEP,DECR);   // Now decrement on zpass
```

Heidmann's described algorithm computes the front- or back-facing orientation of shadow volume polygons on the CPU. We note (as have other authors [5][15]) that the shadow volume polygons can be rendered in two passes: first, culling back-facing polygons to increment pixels rasterized by front-facing polygons; second, culling front-facing polygons to decrement pixels rasterized by back-facing polygons. This leverages the graphics hardware's ability to make the face culling determination automatically and minimizes hardware state changes at the cost of rendering the shadow volume polygons twice. Utilizing the hardware's face culling also avoids inconsistencies if the CPU and graphics hardware determine a polygon's orientation differently in razor's edge cases.

After the shadow volume polygons are rendered into the scene, a pixel's stencil value is equal to zero if the light illuminates the pixel and greater than zero if the pixel is shadowed. The scene can then be re-rendered with the appropriate light configured and enabled, with stencil testing enabled to update only pixels with a zero stencil value (meaning the pixel is not shadowed), and "depth equal" depth testing (to update only visible fragments). The light's contribution can be accumulated with either the accumulation buffer or additive blending.

This can be repeated for multiple light sources, clearing the stencil buffer between rendering the shadow volumes and summing the contribution of each light.

2.2.2 Near and Far Plane Clipping and Capping

Heidmann fails to mention in his article a problem that seriously undermines the robustness of his approach. With arbitrary scenes, the near and/or far clip planes may (and, in fact, often will) clip the infinite shadow volumes. Each shadow volume is, by construction, a half-space (dividing the entirety of space into the region shadowed by a given occluder and everything else). However, near and far plane clipping can "slice open" an otherwise well-defined half space. Disturbing the shadow volume in this way leads to incorrect shadow depth counting that, in turn, results in glaringly incorrect shadowing.

Diefenbach [11] in 1996 recognized the problem created by near plane clipping for shadow volume rendering. Diefenbach presents a method that he claims works "for any shadow volume geometry from any viewpoint," but the method, in fact, does not work in several cases. Figure 1 illustrates three cases where Diefenbach's method fails.

Another solution to the shadow volume near plane clipping problem mentioned by Diefenbach is capping off the shadow volume's intersection with the near clip plane. Other authors [2][4][9][16][17] have also suggested this approach. The problem with near plane capping of shadow volumes is that it is, as described by Carmack [9], a "fragile" procedure.

Capping involves projecting each occluder's back-facing polygons to the near clip plane. This can be complicated when only one or two of a projected polygon's vertices intersect the near plane and careful plane-plane intersection computations are required in such cases. The capping process is further complicated when a back-facing occluder polygon straddles the near clip plane.

Rendering capping polygons at the near clip plane is difficult because of the razor's edge nature of the near clip plane. If you are not careful, the very near plane you are attempting to cap can clip your capping polygons! Additionally if the capping polygons are not "watertight" (2-manifold) with the shadow volume being

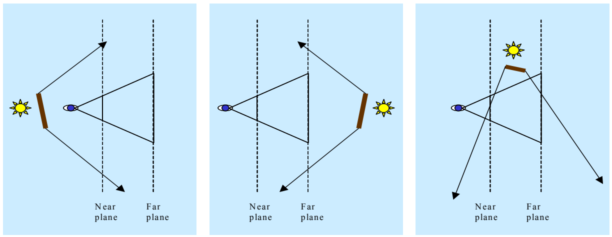


Figure 1: Three cases where Deifenbach's capping algorithm fails because some or all pixels requiring capping are covered by neither a front-nor back-facing polygon so Diefenbach's approach cannot correct these pixels.

capped then rasterization cracks or double hitting of pixels can create shadowing artifacts. These artifacts appear as exceedingly narrow regions of the final scene where areas that clearly should be illuminated are shadowed and vice versa. These artifacts are painfully obvious in animated scenes.

Watertight capping is non-trivial, particularly if shadow volumes are drawn using object-space geometry so that fast dedicated vertex transformation hardware can be exploited. Kilgard [16] proposes creating a "near plane ledge" whereby closed capping polygons can be rendered in a way that avoids clipping by the near clip plane even when rendering object-space shadow volume geometry. This approach cedes a small amount of depth buffer precision for the ledge. Additionally shadow volume capping polygons must be rendered twice, incrementing front-faces and decrementing back-facing geometry because the orientation (front- or back-facing) of a polygon can occasionally flip when a polygon of nearly zero area in window space is transformed from object space to window space due to floating-point numerics. Otherwise, shadow artifacts result.

Even when done carefully, shadow volume near plane capping is treacherous because of the fragile nature of required ray-plane intersections and the inability to guarantee identical and bit-exact CPU and GPU floating-point computations. In any case, capping computations burden the CPU with an expensive task that our algorithm obviates.

2.2.3 Zpass vs. Zfail Stenciled Shadow Volumes

The conventional stenciled shadow volume formulation is to increment and decrement the shadow depth count for front- and back-facing polygons respectively when the depth test passes. Bilodeau [5] in 1999 noted that reversing the depth comparison works too. Another version of this alternative formulation is to decrement and increment the shadow depth count for front- and back-facing polygons respectively depth test fails (without reversing the comparison).

Carmack [9] in 2000 realized the equivalence of the two formulations because they both achieve the *same* result, if in the "depth test fail" formulation, the shadow volume is "closed off" at both ends (rather than being open at the ends). Compare the following OpenGL rendering state modifications with the settings for conventional shadow volume rendering in section 2.2.1.

Front-facing shadow volume rendering configuration:

```
    StencilOp(KEEP,DECR,KEEP); // decrement on zfail
```

Back-facing configuration:

```
    StencilOp(KEEP,INCR,KEEP); // increment on zfail
```

What Carmack describes is projecting back faces, with respect to the light source, some large but finite distance (importantly, still within the far clip plane) and also treating the front faces of the occluder, again with respect to the light source, as a part of the shadow volume boundary too. This still has a problem because when a light source is arbitrarily close to a single occluder polygon, any finite distance used to project out the back faces of the occluder to close off the shadow volume may not extend far enough to ensure that objects beyond the occluder are properly shadowed.

Still Carmack's insight is fundamental to our new algorithm. We call Bilodeau and Carmack's approach *zfail* stenciled shadow volume rendering because the stencil increment and decrement operations occur when the depth test fails rather than when it passes. We call the conventional approach *zpass* rendering.

One way to think of the *zfail* formulation, in contrast to the *zpass* formulation, is that the *zfail* version counts shadow volume intersections from the opposite direction. The *zpass* formulation counts shadow volume enters and exits along each pixel's view ray between the eye point and the first visible rasterized fragment. Technically due to near plane clipping, the counting occurs only between the ray's intersection point with the near clip plane and the first visible rasterized fragment. The objective of shadow volume capping is to introduce sufficient shadow volume enters so that the eye can always be considered "out of shadow" so the stencil count can reflect the true absolute shadow depth of the first visible rasterized fragment.

The *zfail* formulation instead counts shadow volume enters and exits along each pixel's view ray between *infinity* and the first visible rasterized fragment. Technically due to far plane clipping, the counting occurs only between the ray's intersection with the far plane and the first visible fragment. By capping the open end of the shadow volume at or before the far clip plane, we can force the idea that *infinity* is always outside of the shadow volume.

3. OUR ALGORITHM

3.1 Requirements

For our algorithm to operate robustly, we require the following:

- Models for occluding objects must be composed of triangles only (avoiding non-planar polygons), be closed (2-manifold), and have a consistent winding order for triangles within the model. Homogeneous object coordinates are permitted, assuming w≥0.
- Light sources must be ideal points. Homogeneous light positions (w≥0) allow both positional and directional lights.
- Connectivity information for occluding models must be available so that silhouette edges with respect to a light position can be determined at shadow volume construction time.
- The projection matrix must be perspective, not orthographic.
- Functionality available in OpenGL 1.0 [18] and DirectX 6: transformation and clipping of homogeneous positions; front and back face culling; masking color and depth buffer writes; depth buffering; and stencil-testing support.
- The renderer must support N bits of stencil buffer precision, where 2^N is greater than the maximum shadow depth count ever encountered during the processing of a given scene.

This requirement is scene dependent, but 8 bits of stencil buffer precision (typical for most hardware today) is reasonable for typical scenes.

- The renderer must guarantee "watertight" rasterization (no double hitting of pixels or missed pixels along shared edges of rasterized triangles).

Support for non-planar polygons and open models can be achieved using special case handling along the lines described by Bergeron [3].

3.2 Approach

We developed our algorithm by methodically addressing the fundamental limitations of the conventional stenciled shadow volume approach. We combine (1) placing the conventional far clip plane "at infinity"; (2) rasterizing infinite (but fully closed) shadow volume polygons via homogeneous coordinates; and (3) adopting the *zfail* stencil-testing scheme.

This is sufficient to render shadow volumes robustly because it avoids the problems created by the far clip plane "slicing open" the shadow volume. The shadow volumes we construct project "all the way to infinity" through the use of homogeneous coordinates to represent the shadow volume's infinite back projection. Importantly, though our shadow volume geometry is infinite, it is also fully closed. The far clip plane, in eye-space, is infinitely far away so it is impossible for any of the shadow volume geometry to be clipped by it.

By using the *zfail* stencil-testing scheme, we can always assume that infinity is "beyond" all closed shadow volumes if we, in fact, close off our shadow volumes at infinity. This means the shadow depth count can always start from zero for every pixel. We need not worry about the shadow volume being clipped by the near clip plane since we are counting shadow volume enters and exits from infinity, rather than from the eye, due to *zfail* stencil-testing. No fragile capping is required so our algorithm is both robust and automatic.

3.2.1 Far Plane at Infinity

The standard perspective formulation of the projection matrix used to transform eye-space coordinates to clip space in OpenGL (see `glFrustum` [18]) is

$$\mathbf{P} = \begin{bmatrix} \frac{2\times Near}{Right-Left} & 0 & \frac{Right+Left}{Right-Left} & 0 \\ 0 & \frac{2\times Near}{Top-Bottom} & \frac{Top+Bottom}{Top-Bottom} & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2\times Far\times Near}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where *Near* and *Far* are the respective distances from the viewer to the near and far clip planes in eye-space.

P is used to transform eye-space positions to clip-space positions:

$$\begin{bmatrix} x_c & y_c & z_c & w_c \end{bmatrix}^T = \mathbf{P} \begin{bmatrix} x_e & y_e & z_e & w_e \end{bmatrix}^T$$

We are interested in avoiding far plane clipping so we only concern ourselves with the third and fourth row of **P** used to compute clip-space z_c and w_c. Regions of an assembled polygon with interpolated clip coordinates outside $-w_c \le z_c \le w_c$ are clipped by the near and far clip planes.

We consider the limit of **P** as the far clip plane distance is driven to infinity (this is not novel; Blinn [7] mentions the idea):

$$\lim_{Far\to\infty} \mathbf{P} = \mathbf{P_{inf}} = \begin{bmatrix} \frac{2\times Near}{Right-Left} & 0 & \frac{Right+Left}{Right-Left} & 0 \\ 0 & \frac{2\times Near}{Top-Bottom} & \frac{Top+Bottom}{Top-Bottom} & 0 \\ 0 & 0 & -1 & -2\times Near \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The first, second, and fourth rows of $\mathbf{P_{inf}}$ are the same as **P**; only the third row changes. There is no longer a *Far* distance.

A vertex that is an infinite distance from the viewer is represented in homogeneous coordinates with a zero w_e coordinate. If the vertex is transformed into clip space using $\mathbf{P_{inf}}$, assuming the vertex is in front of the eye, meaning that z_e is negative (the OpenGL convention), then w_c=z_c so this transformed vertex is *not* clipped by the far plane. Moreover, its non-homogeneous depth z_c/w_c must be 1.0, generating the maximum possible depth value.

It may be surprising, but positioning the far clip plane at infinity typically reduces the depth buffer precision only marginally. Consider how much we would need to shrink our window coordinates so we can represent within the depth buffer an infinite eye-space distance in front of the viewer. The projection **P** transforms (0,0,-1,0) in eye-space (effectively, an infinite distance in front of the viewer) to the window-space depth *Far*/(*Far*-*Near*). The largest window coordinate representable in the depth buffer is 1 so we must scale *Far*/(*Far*-*Near*) by its reciprocal to "fit" infinity in the depth buffer. This scale factor is (*Far*-*Near*)/*Far* and is very close to 1 if *Far* is many times larger than *Near* which is typical.

Said another way, using $\mathbf{P_{inf}}$ instead of **P** only compresses the depth buffer precision slightly in typical scenarios. For example, if *Near* and *Far* are 1 and 100, then the depth buffer's precision must be squeezed by just 1% to represent an infinite distance in front of the viewer.

3.2.2 Infinite Shadow Volume Polygons

We assume that given a light source position and a closed model with its edge-connectivity, we can determine the subset of *possible silhouette* edges for the model. A possible silhouette edge is an edge shared by two triangles in a model where one of the two triangles faces a given light while the other triangle faces away from the light.

We call these edges "possible silhouette" edges rather than just silhouette edges because these edges are not necessarily boundaries between shadowed and illuminated regions as implied by the conventional meaning of silhouette. It is possible that an edge is an actual silhouette edge, but it is also possible that the edge is itself in shadow.

Assume we have computed the plane equations in the form Ax+By+Cz+Dw=0 for every triangle in a given model. The plane equation coefficients must be computed using a vertex ordering consistent with the winding order shared by all the triangles in the model such that Ax+By+Cz+Dw is non-negative when a point (x,y,z,w) is on the front-facing side of the triangle's plane. Assume we also know the light's homogeneous position L in the coordinate space matching the plane equations. For each triangle, evaluate d=AL_x+BL_y+CL_z+DL_w for the triangle's plane equation coefficients and the light's position. If d is negative, then the triangle is back-facing with respect to *L*; otherwise the triangle is front-facing with respect to *L*. Any edge shared by two triangles with one triangle front-facing and the other back-facing is a possible silhouette edge.

To close a shadow volume completely, we must combine three sets of polygons: (1) all of the possible silhouette polygon edges extruded to infinity away from the light; (2) all of the occluder's back-facing triangles, with respect to *L*, projected away from the light to infinity; and (3) all of the occluder's front-facing triangles with respect to *L*.

Each possible silhouette edge has two vertices *A* and *B,* represented as homogeneous coordinates and ordered based on the front-facing triangle's vertex order. The shadow volume extrusion polygon for this possible silhouette is formed by the edge and its projection to infinity away from the light. The resulting quad consists of the following four vertices:

$$\langle B_x, B_y, B_z, B_w \rangle$$
$$\langle A_x, A_y, A_z, A_w \rangle$$
$$\langle A_x L_w - L_x A_w, A_y L_w - L_y A_w, A_z L_w - L_z A_w, 0 \rangle$$
$$\langle B_x L_w - L_x B_w, B_y L_w - L_y B_w, B_z L_w - L_z B_w, 0 \rangle$$

The last two vertices are the homogeneous vector differences of *A*-*L* and *B*-*L*. These vertices represent directions heading away from the light, explaining why they have *w* coordinate values of zero. We do assume $A_w \geq 0$, $B_w \geq 0$, $L_w \geq 0$, etc.

When we use a perspective transform of the form $\mathbf{P_{inf}}$, we can render shadow volume polygons without the possibility that the far plane will clip these polygons.

For each back-facing occluder triangle, its respective triangle projected to infinity is the triangle formed by the following three vertices:

$$\langle A_x L_w - L_x A_w, A_y L_w - L_y A_w, A_z L_w - L_z A_w, 0 \rangle$$
$$\langle B_x L_w - L_x B_w, B_y L_w - L_y B_w, B_z L_w - L_z B_w, 0 \rangle$$
$$\langle C_x L_w - L_x C_w, C_y L_w - L_y C_w, C_z L_w - L_z C_w, 0 \rangle$$

where *A*, *B*, and *C* are each back-facing occluder triangle's three vertices (in the triangle's vertex order).

The front-facing polygons with respect to *L* are straightforward. Given each triangle's three vertices *A*, *B*, and *C* (in the triangle's vertex order), the triangle is formed by the vertices:

$$\langle A_x, A_y, A_z, A_w \rangle$$
$$\langle B_x, B_y, B_z, B_w \rangle$$
$$\langle C_x, C_y, C_z, C_w \rangle$$

Together, these three sets of triangles form the closed geometry of an occluder's shadow volume with respect to the given light.

3.3 Rendering Procedure

Now we sketch the complete rendering procedure to render shadows with our technique. Pseudo-code with OpenGL commands is provided to make the procedure more concrete.

1. Clear the depth buffer to 1.0; clear the color buffer.

```
Clear(DEPTH_BUFFER_BIT | COLOR_BUFFER_BIT);
```

2. Load the projection with $\mathbf{P_{inf}}$ given the aspect ratio, field of view, and near clip plane distance in eye-space.

```
float Pinf[4][4];
Pinf[1][0] = Pinf[2][0] = Pinf[3][0] = Pinf[0][1] =
Pinf[2][1] = Pinf[3][1] = Pinf[0][2] = Pinf[1][2] =
Pinf[0][3] = Pinf[1][3] = Pinf[3][3] = 0;
Pinf[0][0] = cotangent(fieldOfView)/aspectRatio;
Pinf[1][1] = cotangent(fieldOfView);
Pinf[3][2] = -2*near; Pinf[2][2] = Pinf[2][3] = -1;
MatrixMode(PROJECTION); LoadMatrixf(&Pinf[0][0]);
```

3. Load the modelview matrix with the scene's viewing transform.

```
MatrixMode(MODELVIEW); loadCurrentViewTransform();
```

4. Render the scene with depth testing, back-face culling, and all light sources disabled (ambient & emissive illumination only).

```
Enable(DEPTH_TEST); DepthFunc(LESS);
Enable(CULL_FACE);  CullFace(BACK);
Enable(LIGHTING);   Disable(LIGHT0);
LightModelfv(LIGHT_MODEL_AMBIENT, &globalAmbient);
drawScene();
```

5. Disable depth writes, enable additive blending, and set the global ambient light contribution to zero (and zero any emissive contribution if present).

```
DepthMask(0);
Enable(BLEND); BlendFunc(ONE,ONE);
LightModelfv(LIGHT_MODEL_AMBIENT, &zero);
```

6. For each light source:

   A. Clear the stencil buffer to zero.

```
Clear(STENCIL_BUFFER_BIT);
```

   B. Disable color buffer writes and enable stencil testing with the *always* stencil function and writing stencil..

```
ColorMask(0,0,0,0);
Enable(STENCIL_TEST);
StencilFunc(ALWAYS,0,~0); StencilMask(~0);
```

   C. For each occluder:

      a. Determine whether each triangle in the occluder's model is front- or back-facing with respect to the light's position. Update `triList[].backfacing`.

      b. Configure *zfail* stencil testing to increment stencil for back-facing polygons that fail the depth test.

```
CullFace(FRONT); StencilOp(KEEP,INCR,KEEP);
```

      c. Render all possible silhouette edges as quads that project from the edge away from the light to infinity.

```
Vert L = currentLightPosition;
Begin(QUADS);
  for (int i=0; i<numTris; i++) // for each triangle
    // if triangle is front-facing with respect to the light
    if (triList[i].backFacing==0)
      for (int j=0; j<3; j++) // for each triangle edge
        // if adjacent triangle is back-facing
        //  with respect to the light
        if (triList[triList[i].adjacent[j]].backFacing) {
          // found possible silhouette edge
          Vert A = triList[i].v[j];
          Vert B = triList[i].v[(j+1) % 3]; // next vertex
          Vertex4f(B.x,B.y,B.z,B.w);
          Vertex4f(A.x,A.y,A.z,A.w);
          Vertex4f(A.x*L.w-L.x*A.w,
                   A.y*L.w-L.y*A.w,
                   A.z*L.w-L.z*A.w, 0); // infinite
          Vertex4f(B.x*L.w-L.x*B.w,
                   B.y*L.w-L.y*B.w,
                   B.z*L.w-L.z*B.w, 0); // infinite
        }
End(); // quads
```

d. Specially render all occluder triangles. Project and render back facing triangles away from the light to infinity. Render front-facing triangles directly.

```
#define V triList[i].v[j] // macro used in Vertex4f calls
Begin(TRIANGLES);
  for (int i=0; i<numTris; i++) // for each triangle
    // if triangle is back-facing with respect to the light
    if (triList[i].backFacing)
      for (int j=0; j<3; j++) // for each triangle vertex
        Vertex4f(V.x*L.w-L.x*V.w, V.y*L.w-L.y*V.w,
                 V.z*L.w-L.z*V.w, 0); // infinite
    else
      for (int j=0; j<3; j++) // for each triangle vertex
        Vertex4f(V.x,V.y,V.z,V.w);
End(); // triangles
```

e. Configure *zfail* stencil testing to decrement stencil for front-facing polygons that fail the depth test.

```
CullFace(BACK);  StencilOp(KEEP,DECR,KEEP);
```

f. Repeat steps (c) and (d) above, this time rendering front facing polygons rather than back facing ones.

D. Position and enable the current light (and otherwise configure the light's attenuation, color, etc.).

```
Enable(LIGHT0);
Lightfv(LIGHT0, POSITION, &currentLightPosition.x);
```

E. Set stencil testing to render only pixels with a zero stencil value, i.e., visible fragments illuminated by the current light. Use *equal* depth testing to update only the visible fragments, and then, increment stencil to avoid double blending. Re-enable color buffer writes again.

```
StencilFunc(EQUAL, 0, ~0); StencilOp(KEEP,KEEP,INCR);
DepthFunc(EQUAL);  ColorMask(1,1,1,1);
```

F. Re-draw the scene to add the contribution of the current light to illuminated (non-shadowed) regions of the scene.

```
drawScene();
```

G. Restore the depth test to *less*.

```
DepthFunc(LESS);
```

7. Disable blending and stencil testing; re-enable depth writes.

```
Disable(BLEND);  Disable(STENCIL_TEST);  DepthMask(1);
```

3.4 Optimizations

Possible silhouette edges form closed loops. If a loop of possible silhouette edges is identified, then sending `QUAD_STRIP` primitives (2 vertices/projected quad), rather than independent quads (4 vertices/projected quad) will reduce the per-vertex transformation overhead per shadow volume quad. Similarly, the independent triangle rendering used for capping the shadow volumes can be optimized for rendering as triangle strips or indexed triangles.

The `INCR` *zpass* stencil operation in step 6.E avoids the double blending of lighting contributions in the usually quite rare circumstance when two fragments alias to the exact same pixel location and depth value. Using the `KEEP` *zpass* stencil operation instead can avoid usually unnecessary stencil buffer writes, improving rendering performance in situations where double blending is deemed unlikely.

In the case of a directional light, all the vertices of a possible silhouette edge loop project to the same point at infinity. In this case, a `TRIANGLE_FAN` primitive can render these polygons extremely efficiently (1 vertex/projected triangle).

If the application determines that the shadow volume geometry for a silhouette edge loop will never pierce or otherwise require capping of the near clip plane's visible region, *zpass* shadow volume rendering can be used instead of *zfail* rendering. The *zpass* formulation is advantageous in this context because it does not require the rendering of any capping triangles. Mixing the *zpass* and *zfail* shadow volume stencil testing formulations for different silhouette edge loops does not affect the net shadow depth count as long as each particular loop uses a single formulation.

Shadow volume geometry can be re-used from frame to frame for any light and occluder that have not changed their geometric relationship to each other.

4. IMPROVED HARDWARE SUPPORT

4.1 Wrapping Stencil Arithmetic

DirectX 6 and the OpenGL *EXT_stencil_wrap* extension provide two additional *increment wrap* and *decrement wrap* stencil operations that use modulo, rather than saturation, arithmetic. These operations reduce the likelihood of incorrect shadow results due to an increment operation saturating a stencil value's shadow depth count. Using the wrapping operations with an *N*-bit stencil buffer, there remains a remote possibility that a net 2^N increments (or a multiple of) may alias with the unshadowed zero stencil value and lead to incorrect shadows, but in practice, particularly with an 8-bit stencil buffer, this is quite unlikely.

4.2 Depth Clamping

NVIDIA's GeForce3 and GeForce4 Ti GPUs support *depth clamping* via the *NV_depth_clamp* OpenGL extension. When enabled, depth clamping disables the near and far clip planes for rasterizing geometric primitives. Instead, a fragment's window-space depth value is clamped to the range [min(*zn*,*zf*),max(*zn*,*zf*)] where *zn* and *zf* are the near and far depth range values. Additionally when depth clamping is enabled, no fragments with non-positive w_c are generated.

With depth clamping support, a conventional projection matrix with a finite far clip plane distance can be used rather than the $\mathbf{P}_{\mathbf{inf}}$ form. The only required modification to our algorithm is enabling `DEPTH_CLAMP_NV` during the rendering of the shadow volume geometry.

Depth clamping recovers the depth precision (admittedly quite marginal) lost due to the use of a $\mathbf{P}_{\mathbf{inf}}$ projection matrix. More significantly, depth clamping generalizes our algorithm so it works with orthographic, not just perspective, projections.

4.3 Two-Sided Stencil Testing

We propose *two-sided stencil testing*, a new stencil functionality that uses distinct front- and back-facing stencil state when enabled. Front-facing primitives use the front-facing stencil state for their stencil operation while back-facing primitives use the back-facing state. With two-sided stencil testing, shadow volume geometry need only be rendered once, rather than twice.

Two-sided stencil testing generates the same number of stencil buffer updates as the two-pass approach so in fill-limited shadow volume rendering situations, the advantage of a single pass is marginal. However, pipeline bubbles due to repeated all front-facing or all back-facing shadow volumes lead to inefficiencies using two passes. Perhaps more importantly, two-sided stencil testing reduces the CPU overhead in the driver by sending shadow volume polygon geometry only once.

Because stencil increments and decrements are intermixed with two-sided stencil testing, the wrapping versions of these operations are mandatory.

5. EXAMPLES

Figures 2 through 5 show several examples of our algorithm.

6. FUTURE WORK

Because of the extremely scene-dependent nature of shadow volume rendering performance and space constraints here, we defer thorough performance evaluation of our technique. Still we are happy to report that our rendering examples, including examples that seek to mimic the animated behavior of a sophisticated 3D game (see Figure 4), achieve real-time rates on current PC graphics hardware.

Yet naïve rendering with stenciled shadow volumes consumes tremendous amounts of stencil fill rate. We expect effective shadow volume culling schemes will be required to achieve consistent interactive rendering rates for complex shadowed scenes. Portal, BSP, occlusion, and view frustum culling techniques can all improve performance by avoiding the rendering of unnecessary shadow volumes. Additional performance scaling will be through faster and cleverer hardware designs that are better tuned for rendering workloads including stenciled shadow volumes.

Future graphics hardware will support more higher-order graphics primitives beyond triangles. Combining higher-order hardware primitives with shadow volumes requires automatic generation of shadow volumes in hardware. Two-sided stencil testing will be vital since it only requires one rendering of automatically generated shadow volume geometry. Automatic generation of shadow volumes will also relieve the CPU of this chore.

7. CONCLUSIONS

Our stenciled shadow volume algorithm is robust, straightforward, and requires hardware functionality that is ubiquitous today. We believe this will provide the opportunity for 3D games and applications to integrate shadow volumes into their basic rendering repertoire. The algorithm we developed is the result of careful integration of known, but not previously integrated, techniques to address methodically the shortcomings of existing shadow volume techniques caused by near and/or far plane clipping.

8. ACKNOWLEDGEMENTS

We are grateful to John Carmack, Matt Craighead, Eric Haines, and Matt Papakipos for fruitful discussions, Steve Burke for modeling the SIGGRAPH logo and Loop subdivision surface for us; and James Green, Brian Collins, Rich B, and Stecki for designing and animating the Quake2 models that we picture.

REFERENCES

[1] Kurt Akeley and James Foran, "Apparatus and method for controlling storage of display information in a computer system," *US Patent 5,394,170*, filed Dec. 15, 1992, assigned Feb. 28, 1995.

[2] Harlen Costa Batagelo and Ilaim Costa Junior, "Real-Time Shadow Generation Using BSP Trees and Stencil Buffers," *XII Brazilian Symposium on Computer Graphics and Image Processing*, Campinas, Brazil, Oct. 1999, pp. 93-102.

[3] Philippe Bergeron, "A General Version of Crow's Shadow Volumes," *IEEE Computer Graphics and Applications*, Sept. 1986, pp. 17-28.

[4] Jason Bestimt and Bryant Freitag, "Real-Time Shadow Casting Using Shadow Volumes," Gamasutra.com web site, Nov. 15, 1999.

[5] Bill Bilodeau and Mike Songy, Creative Labs sponsored game developer conference, unpublished slides, Los Angeles, May 1999.

[6] David Blythe, Tom McReynolds, et.al., "Shadow Volumes," *Program with OpenGL: Advanced Rendering*, SIGGRAPH course notes, 1996.

[7] Jim Blinn, "A Trip Down the Graphics Pipeline: The Homogeneous Perspective Transform," *IEEE Computer Graphics and Applications*," May 1993, pp. 75-88.

[8] Lynne Brotman and Norman Badler, "Generating Soft Shadows with a Depth Buffer Algorithm," *IEEE Computer Graphics and Applications*, Oct. 1984, pp. 5-12.

[9] John Carmack, unpublished correspondence, early 2000.

[10] Frank Crow, "Shadow Algorithms for Computer Graphics," *Proceedings of SIGGRAPH*, 1977, pp. 242-248.

[11] Paul Diefenbach, *Multi-pass Pipeline Rendering: Interaction and Realism through Hardware Provisions,* Ph.D. thesis, University of Pennsylvania, tech report MS-CIS-96-26, 1996.

[12] Alain Fournier and Donald Fussell, "On the Power of the Frame Buffer," *ACM Transactions on Graphics*, April 1988, 103-128.

[13] Henry Fuchs, Jack Goldfeather, Jeff Hultquist, Susan, Spach, John Austin, Frederick Brooks, John Eyles, and John Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *Proceedings of SIGGRAPH*, 1985, pp. 111-120.

[14] Tim Heidmann, "Real Shadows Real Time", *IRIS Universe,* Number 18, 1991, pp. 28-31.

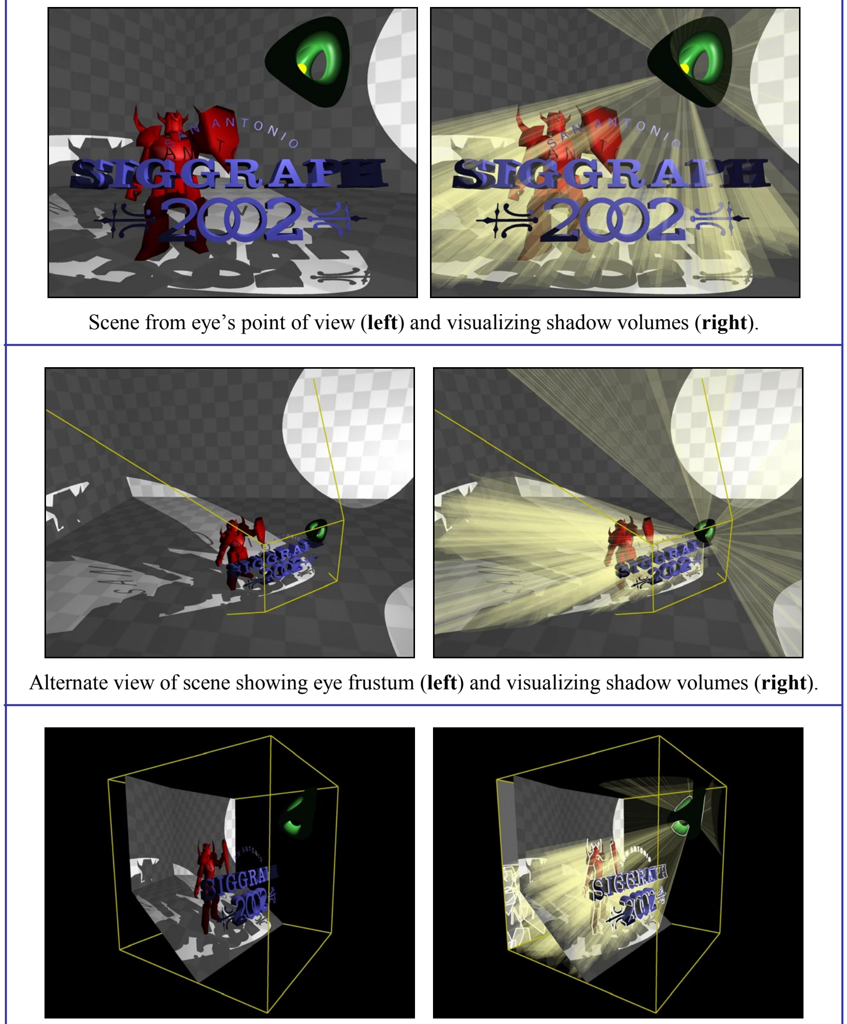
[15] Mark Kilgard, "Improving Shadows and Reflections via the Stencil Buffer," *Advanced OpenGL Game Development* course notes, Game Developer Conference, March 16, 1999, pp. 204-253.

[16] Mark Kilgard, "Robust Stencil Volumes," CEDEC 2001 presentation, Tokyo, Sept. 4, 2001.

[17] Michael McCool, "Shadow Volume Reconstruction from Depth Maps," *ACM Transactions on Graphics*, Jan. 2001, pp. 1-25.

[18] Mark Segal and Kurt Akeley, *The OpenGL Graphics System: A Specification*, version 1.3, 2001.

[19] Andrew Woo, Pierre Poulin, and Alain Fournier, "A Survey of Shadow Algorithms," *IEEE Computer Graphics and Applications*, Nov. 1990, pp. 13-32.

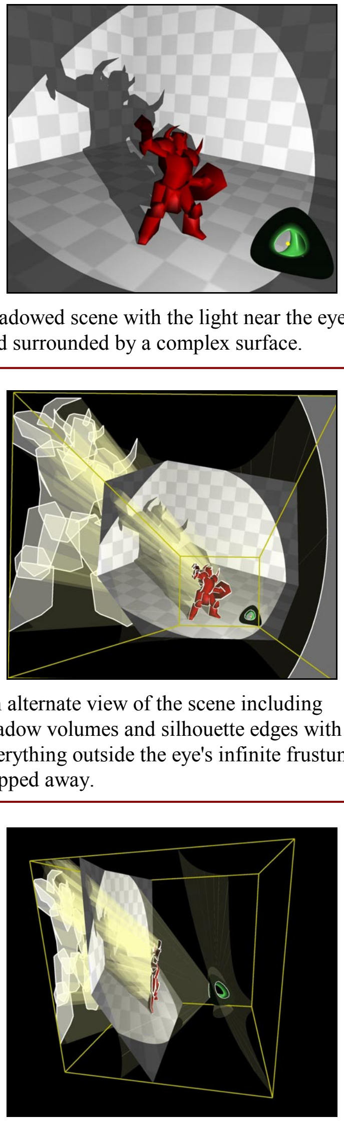


Scene from eye's point of view (**left**) and visualizing shadow volumes (**right**).

Alternate view of scene showing eye frustum (**left**) and visualizing shadow volumes (**right**).

View of scene in eye's clip space (**left**) and visualizing shadow volumes (**right**).

Figure 2: These images show a scene with a yellow light source surrounded by a green complex object. This arrangement is a "hard" case for shadow volume rendering. The infinite capping polygons can be seen behind the wall and floor in the bottom right image. All the scenes use a P_{inf} projection matrix.



Shadowed scene with the light near the eye and surrounded by a complex surface.

An alternate view of the scene including shadow volumes and silhouette edges with everything outside the eye's infinite frustum clipped away.

Same as above, except the scene is shown in eye's clip space.

Figure 3: These images illustrate the capping at infinity that is required for correct closed shadow computation.

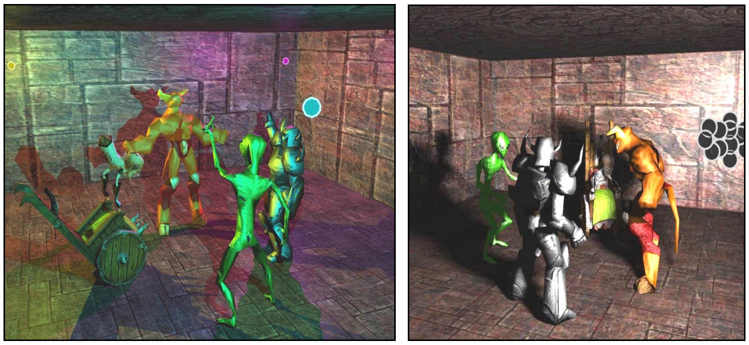


Figure 4: Game-like scenes with 3 independent colored light sources (**left**, 34 frames/second on a GeForce4 Ti 4600 at 640x480, 80+ fps for 1 light) and 12 clustered lights to simulate soft shadows (**right**, 8 fps). Characters have diffuse/specular per-pixel bump map shading, correctly shadowed.

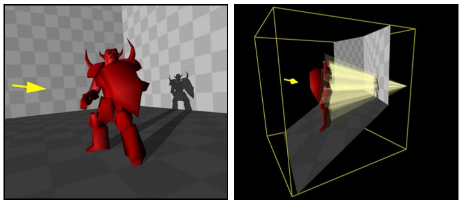


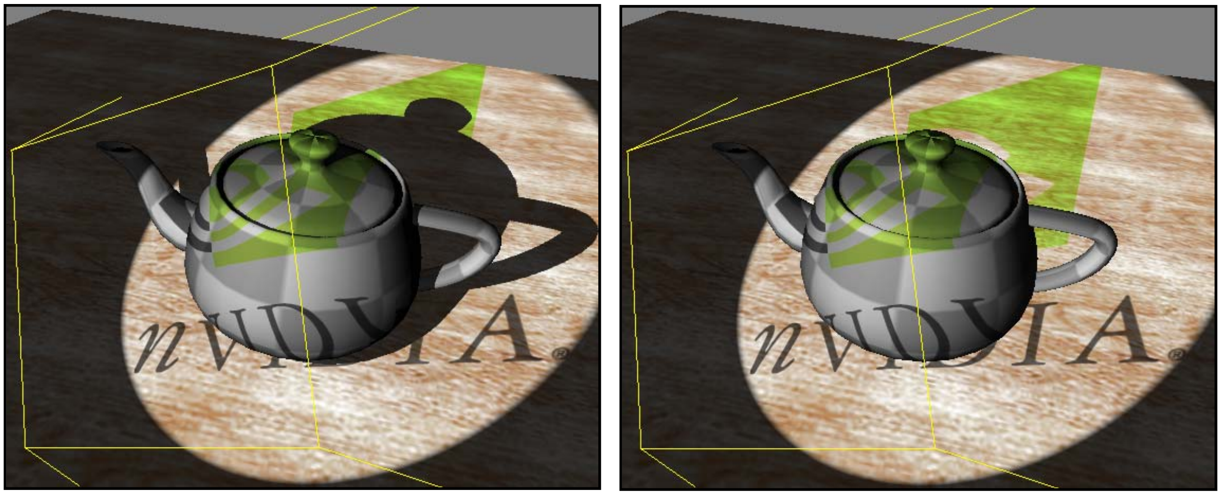
Figure 5: Shadowed scene lit by a directional light (**left**) and the corresponding clip-space view with the shadow volume's back projection meeting at infinity on the far clip plane (**right**).

Hardware Shadow Mapping

Cass Everitt
cass@nvidia.com

Ashu Rege
arege@nvidia.com

Cem Cebenoyan
cem@nvidia.com



Introduction

Shadows make 3D computer graphics look better. Without them, scenes often feel unnatural and flat, and the relative depths of objects in the scene can be very unclear. The trouble with rendering high quality shadows is that they require a visibility test for each light source at each rasterized fragment. For ray tracers, adding an extra visibility test is trivial, but for rasterizers, it is not. Fortunately, there are a number of common cases where the light visibility test can be efficiently performed by a rasterizer. The two most common techniques for hardware accelerated complex shadowing are stenciled shadow volumes and shadow mapping. This document will focus on using shadow mapping to implement shadowing for spotlights.

Shadow mapping is an image-based shadowing technique developed by Lance Williams [8] in 1978. It is particularly amenable to hardware implementation because it makes use of existing hardware functionality – texturing and depth buffering. The only extra burden it places on hardware is the need to perform a high-precision scalar comparison for each texel fetched from the shadow map texture. Shadow maps are also attractive to application programmers because they are very easy to use, and unlike stenciled shadow volumes, they require no additional geometry processing.

Hardware accelerated shadow mapping [5] is available today on GeForce3 GPUs. It is exposed in OpenGL [4] through the SGIX_shadow and SGIX_depth_texture extensions [6], and in Direct3D 8 through a special texture format.

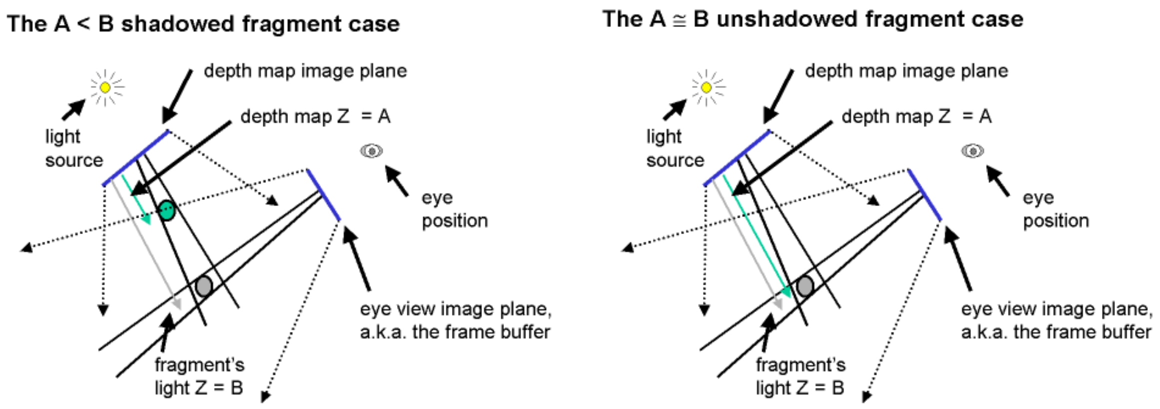


Figure 1. These diagrams were taken from Mark Kilgard's shadow mapping presentation at GDC 2001. They illustrate the shadowing comparison that occurs in shadow mapping.

How It Works

The clever insight of shadow mapping is that the depth buffer generated by rendering the scene from the light's point of view is a pre-computed light visibility test over the light's view volume. The light's depth buffer (a.k.a. the shadow map) partitions the view volume of the light into two regions: the shadowed region and the unshadowed region. The visibility test is of the form

$$p_z \le shadow\_map(p_x, p_y)$$

where p is a point in the light's image space. Shadow mapping really happens in the texture unit, so the comparison actually looks like:

$$\frac{p_r}{p_q} \le texture\_2D\left(\frac{p_s}{p_q}, \frac{p_t}{p_q}\right).$$

Note that this form of comparison is identical to the depth test used for visible surface determination during standard rasterization. The primary difference is that the rasterizer always generates fragments (primitive samples) on the regular grid of the eye's discretized image plane for depth test, while textures are sampled over a continuous space at irregular intervals. If we made an effort to sample the shadow map texture in the same way that we sample the depth buffer, there would be no difference at all. In fact, we can use shadow maps in this way to perform more than one depth test per fragment [2].

Figure 1 illustrates the depth comparison that takes place in shadow mapping. The eye position and image plane are shown, but they are not relevant to the visibility test because shadowing is view-independent.

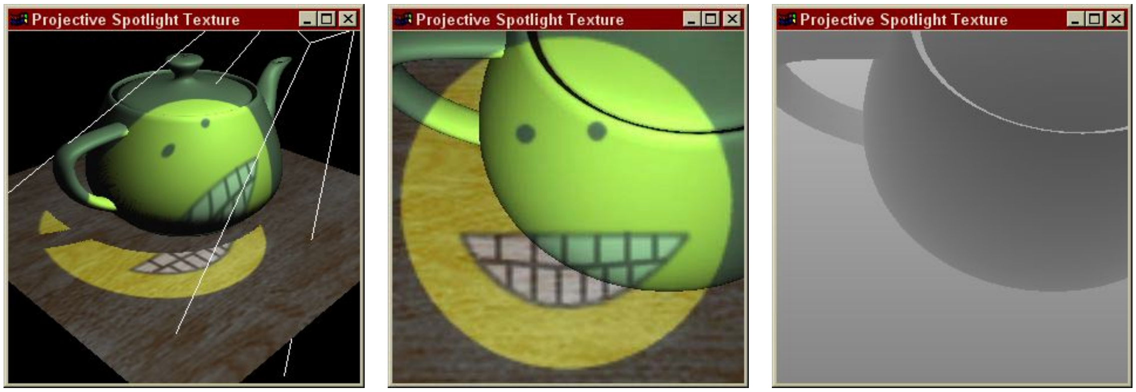

Projective Spotlight Texture
Projective Spotlight Texture
Projective Spotlight Texture

Figure 2. A shadow mapped scene rendered from the eye's point of view (left), the scene as rendered from the light's point of view (center), and the corresponding depth/shadow map (right).

How To Do It

The basic steps for rendering with shadow maps are quite simple:

- render the scene from the light's point of view,
- use the light's depth buffer as a texture (shadow map),
- projectively texture the shadow map onto the scene, and
- use "texture color" (comparison result) in fragment shading.

Figure 2 shows an example scene with shadows, the same scene shown from the light's point of view, and the corresponding shadow map (or depth buffer). Note that samples that are closer to the light are darker than samples that are further away.

Since applications already have to be able to render the scene, rendering from the light's point of view is trivial. If it is available, polygon offset should be used to push fragments back slightly during this rendering pass.

Why Is Polygon Offset Needed?

If implemented literally, the light visibility test described in the previous section is prone to self-shadowing error due to it's razor's edge nature in the case of unshadowed objects. In the hypothetical "infinite resolution, infinite precision" case, surfaces that pass the visibility test would have depth *equal* to the depth value stored in the shadow map. In the real world of finite precision and finite resolution, precision and sampling issues cause problems. These problems can be solved by adding a small bias to the shadow map depths used in the comparison.

If the problem were only one of precision, a constant bias of all the shadow map depths would be sufficient, but there is also a less obvious sampling issue that affects the magnitude of bias necessary. Consider the case illustrated in Figure 3. When the geometry is rasterized from the eye's point of view, it will be sampled in different

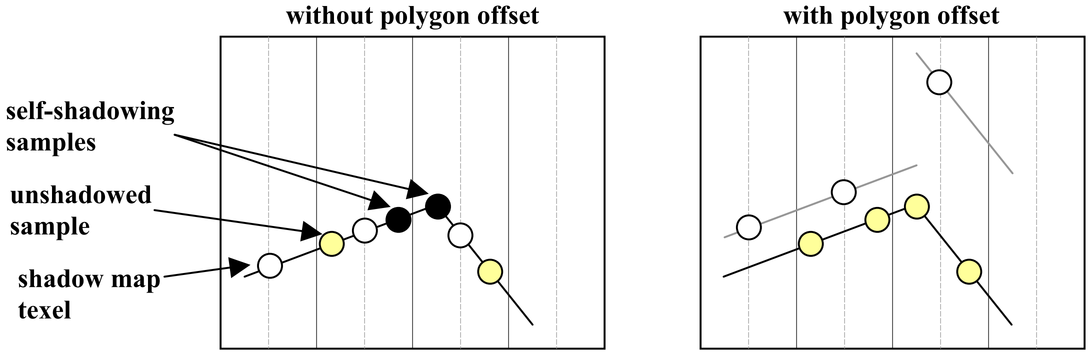


Figure 3. These figures illustrate the need for polygon offsetting to eliminate self-shadowing artifacts. The variable sampling location necessitates the use of z slope-based offset.

locations than when it was rasterized from the light's point of view. The difference in the depths of the samples is based on the slope of the polygon in light space, so in order to account for this we must supply a positive "slope factor" (typically about 1.0) to the polygon offset.

Direct3D does not expose polygon offset, so applications must provide this bias through matrix tweaks. This approach is workable, but because it fails to account for z slope, the uniform bias is generally much larger than it would otherwise need to be, which may introduce incorrectly unshadowed samples, or "light leaking".

The depth map as rendered from the light's point of view *is* the shadow map. With OpenGL, turning it into a real texture requires copying it into texture memory via glCopyTex{Sub}Image2D(). Even though the copy is card-local, it is still somewhat expensive. Direct3D's render-to-texture capability makes this copy unnecessary. You can render directly to the shadow map texture. This render-to-texture capability will also be available soon in OpenGL through extensions.

Once the shadow map texture is generated, it is projectively textured onto the scene. For shadow mapping, we compute 3D projective texture coordinates, where *r* is the sample depth in light space, and *s* and *t* index the 2D texture. Figure 4 shows these quantities, which are compared during rendering to determine light visibility.

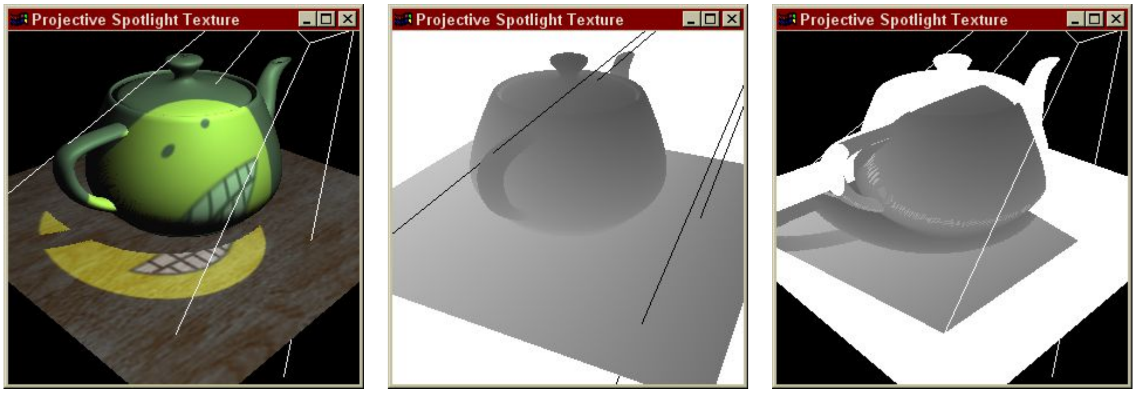

Projective Spotlight Texture
Projective Spotlight Texture
Projective Spotlight Texture

Figure 4. A shadow mapped scene (left), the scene showing each sample's distance from the light source (center), and the scene with the shadow map shadow map projected onto it (right).

The final step in rendering shadows is to actually factor the shadow computation result into the shading equation. The result of the comparison is either 1 or 0, and it is returned as the texture color. If linear filtering is enabled, the comparison is performed at the four neighboring shadow map samples, and the results are bilinearly filtered just as if they had been colors.

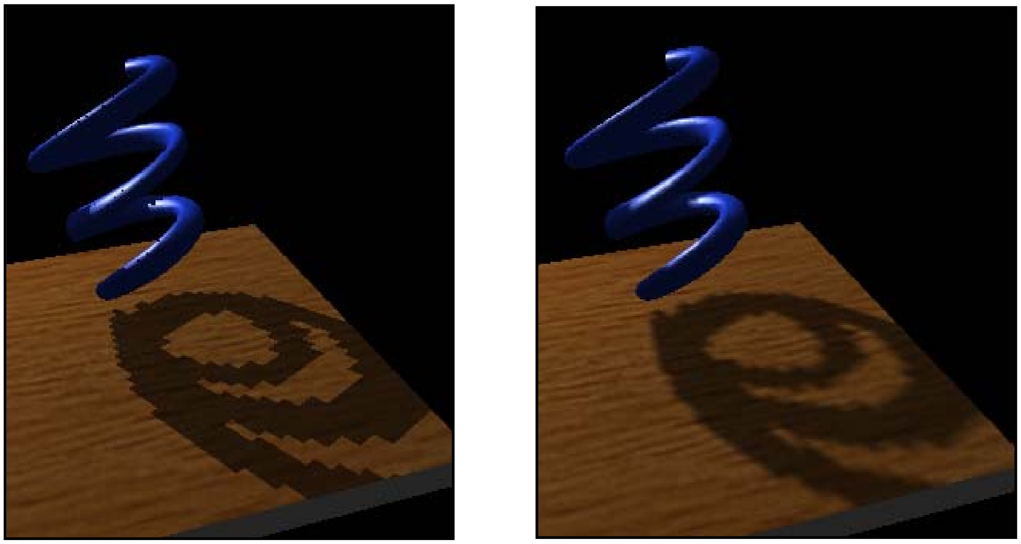


Figure 5. A very low resolution shadow map is used to demonstrate the difference between nearest (left) and linear (right) filtering for shadow maps. Credit: Mark Kilgard.

With GeForce3 hardware, it is easiest to use NV_register_combiners to implement the desired per-fragment shading based on the shadow comparison. One simple approach is to use the shadowing result directly to modulate the diffuse and specular intensity. Kilgard points out [3] that leaving some fraction of diffuse intensity in helps keep shadows areas from looking too "flat".

OpenGL API Details

Support for shadow mapping in OpenGL is provided by the SGIX_shadow and SGIX_depth_texture extensions. The SGIX_shadow extension exposes the per-texel comparison as a texture parameter, and SGIX_depth_texture defines a texture internal format of DEPTH_COMPONENT, complete with various bits-per-texel choices. It also provides semantics for glCopyTex{Sub}Image*() calls to read from the depth buffer when performing a copy.

Direct3D API Details

Support for shadow mapping in Direct3D is provided by special depth texture formats exposed in drivers version 21.81 and later. Support for both 24-bit (D3DFMT_D24S8) and 16-bit (D3DFMT_D16) shadow maps is included.

Setup

The following code snippet checks for hardware shadow map support on the default adapter in 32-bit color:

```
HRESULT hr = pD3D->CheckDeviceFormat(
        D3DADAPTER_DEFAULT,         //default adapter
        D3DDEVTYPE_HAL,             //HAL device
        D3DFMT_X8R8G8B8,            //display mode
        D3DUSAGE_DEPTHSTENCIL,      //shadow map is a depth/s surface
        D3DRTYPE_TEXTURE,           //shadow map is a texture
        D3DFMT_D24S8                //format of shadow map
);
```

Note that since shadow mapping in Direct3D relies on “overloading” the meaning of an existing texture format, the above check does not guarantee hardware shadow map support, since it’s feasible that a particular hardware / driver combo could one day exist that supports depth texture formats for another purpose. For this reason, it’s a good idea to supplement the above check with a check that the hardware is GeForce3 or greater.

Once shadow map support has been determined, you can create the shadow map using the following call:

```
pD3DDev->CreateTexture(texWidth, texHeight, 1,
        D3DUSAGE_DEPTHSTENCIL, D3DFMT_D24S8, D3DPOOL_DEFAULT,
        &pTex);
```

Note that you **must** create a corresponding color surface to go along with your depth surface since Direct3D requires you to set a color surface / z surface pair when doing a SetRenderTarget(). If you’re not using the color buffer for anything, it’s best to turn off color writes when rendering to it using the D3DRS_COLORWRITEENABLE renderstate to save bandwidth.

Rendering

Rendering uses the same ideas as in OpenGL: you render from the point of view of the light to the shadow map you created, then set the shadow map texture in a texture stage and set the texture coordinates in that stage to index into the shadow map at (s / q, t / q) and use the depth value (r / q) for the comparison. There are a few Direct3D-specific idiosyncrasies to be aware of, however:

- The (z / w) value used to compare with the value in the shadow map is in the range [0..$2^{bitdepth}$-1], not [0..1], where ‘bitdepth’ is the bitdepth of the shadowmap (24 or 16 bits). This means you have to put an additional scale factor into your texture matrix.
- Direct3D addresses pixels and texels in different ways [1], where integral screen coordinates address pixel centers and integral texture coordinates address texel boundaries. You need to take this into account when addressing the shadow map. There are two ways to do this: either offset the viewport by half a texel when rendering the shadow map, or offset by half a texel when addressing the shadow map.
- As stated earlier, there is no native polygon offset support in Direct3D. The closest thing is D3DRS_ZBIAS, but this doesn’t help us when shadow mapping since it can only be used to bias depth a constant amount *towards* the camera, not away. Instead we can get similar functionality, albeit without taking into account polygon slope, by adding a small bias amount to our texture matrix.

Here is a sample texture matrix that takes into account these limitations:

```
float fOffsetX = 0.5f + (0.5f / fTexWidth);
float fOffsetY = 0.5f + (0.5f / fTexHeight);
D3DXMATRIX texScaleBiasMat( 0.5f,     0.0f,     0.0f,      0.0f,
                            0.0f,    -0.5f,     0.0f,      0.0f,
                            0.0f,     0.0f,     fZScale,   0.0f,
                            fOffsetX, fOffsetY, fBias,     1.0f );
```

Where fZScale is the ($2^{bitdepth}$-1) and fBias is a small negative value. Note that this matrix is applied post-projection, **not** in eye space.

Once the texture coordinates have been setup properly, the hardware will automatically compare (r / q) > shadowMap[s / q, t / q] and return zero to indicate in shadow or one to indicate in light (or potentially something in between if you’re on the shadow edge and using D3DTEXF_LINEAR). The following pixel shader shows a simple use of shadow mapping (but note that you don’t have to use pixel shaders to use shadow maps, DirectX7-style texture stage states work as well):

```
tex t0   // normal map
tex t1   // decal texture
tex t2   // shadow map
dp3_sat r0, t0_bx2, v0_bx2  //light vector is in v0
mul r0, r0, t2   //modulate lighting contribution by shadow result
mul r0, r0, t1   //modulate lighting contribution by decal
```

Advantages and Limitations

As with any technique, shadow mapping has certain advantages and limitations to be aware of. The fact that it is image-based turns out to be both an advantage and a limitation. It's advantageous, because it doesn't require additional application geometry processing, it works well with GPU-created and GPU-altered geometry and correctly handles fragment culling like alpha test. The associated limitation is that because it's image based, it works well for spotlights, but not point light sources. One could imagine a cube map –based shadow mapping system, but they would require six 90-degree frusta, which would each need to be fairly high resolution, and five more passes over the geometry to generate the shadow map.

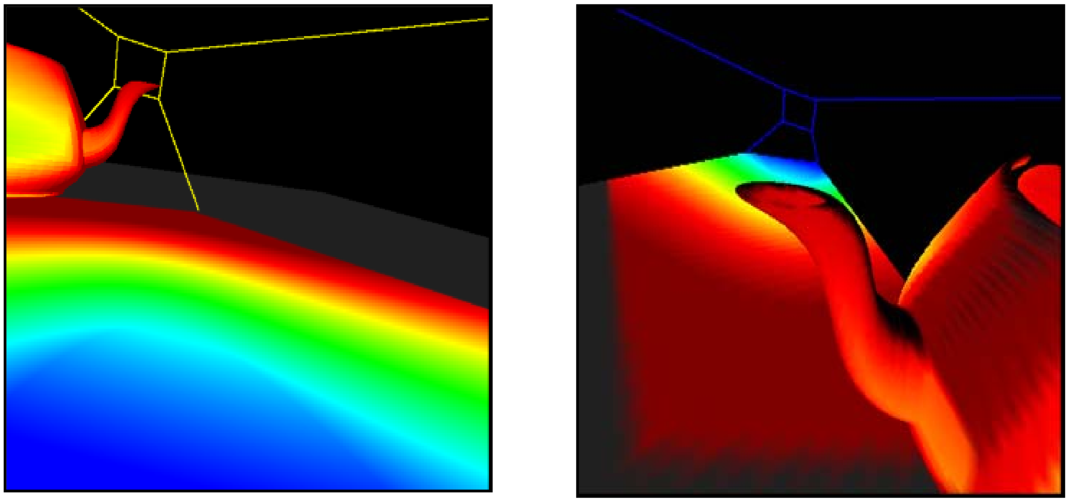


Figure 6. The "dueling frusta" problem occurs when the spotlight points toward the eye. The eye's view (left) shows the variation in sampling frequency of the shadow map, blue being the highest . The light's view (right) shows the very small portion of the light's image plane needs high frequency sampling.

Along the same lines, the quality of shadow mapping depends on how well the shadow map sampling frequency matches the eye's sampling frequency. When the eye and light have similar location and orientation, the sampling frequencies match pretty well, but when the light and eye are looking toward each other, the sampling frequencies rarely match well. Figure 6 illustrates this "dueling frusta" situation.

Another problem that comes up with any projective texture mapping is the phantom "negative projection". This is actually pretty simple to remove at the cost of an additional texture unit, or per-vertex color. The goal is just to make sure that the shadow test always returns "shadowed" for surfaces behind the light.

Finally, the polygon offset fudge factor, while quite adequate for virtually all uses of shadow mapping, can be a bit dissatisfying. Andrew Woo [9] suggested an alternative shadow map generation that is produced from averaging the nearest and second-nearest depth layers from the light's point of view. This technique can actually be implemented as a two-pass technique on GeForce3 hardware using the depth peeling technique described in [2] and with a slight twist. In the second pass, the shadow map is used to peel away the nearest surfaces, but all depths are computed as the average of the fragment's original depth and the nearest depth at that fragment's (x,y). The nearest surface (that is not peeled away), is then the average of the first and second nearest fragments!

Wang and Molnar introduce another technique to reduce the need for polygon offset [7]. Their technique works by rendering only back-faces into the shadow map, relying on the observation that back-face z-values and front-face z-values are likely far enough apart in z to not falsely self-shadow. This only helps front-faces, of course, but back-faces (with respect to the light) are, by definition, not in light, which helps hide artifacts. Note that this algorithm only works for closed polygonal objects.

Computing Transformations for Shadow Mapping

Computing the transformations required for shadow mapping can be somewhat tricky. This section provides details on the various transformations that need to be applied during the two render passes. While this section provides details for the OpenGL case, the transformations required for Direct3D are very similar with the main exception being that the texture coordinate generation is done directly via a matrix instead of the *texgen* planes. Also, keep in mind that the scale-bias matrix in Direct3D requires an additional offset to account for the discrepancy between pixel and texel coordinates as mentioned earlier, and that *eye linear* texgen is called D3DTSS_TCI_CAMERASPACEPOSITION.

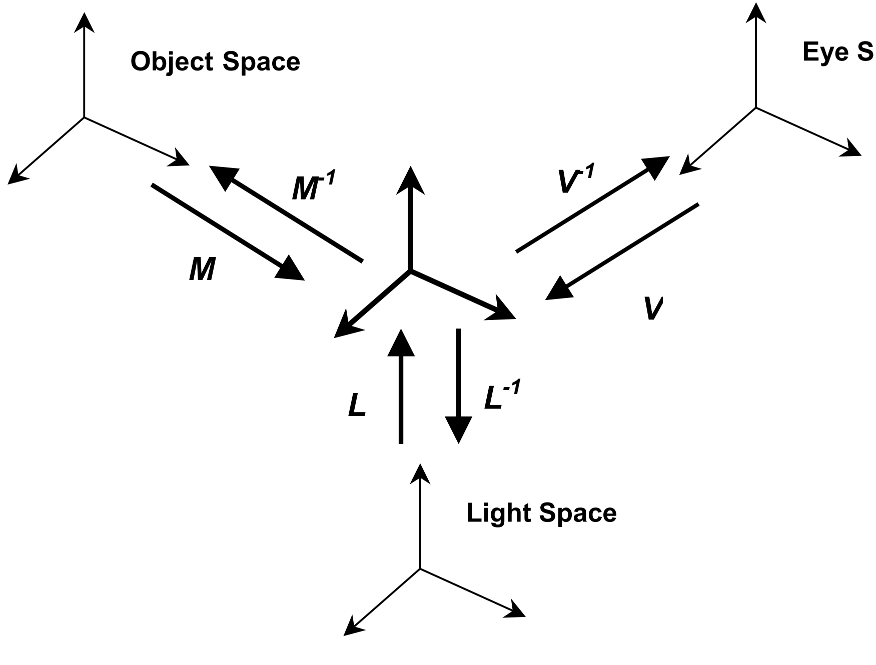


Figure 7: Schematic view of the basic transformations involved in shadow mapping.

Figure 7 shows the three primary transformations (and inverses) used in shadow mapping. Note that we use the convention of using the *forward* transforms as going *to* world coordinates. The standard 'modelview' matrix using the above notation will therefore be: $V^{-1}M$. In addition to the above transformations, we also have to account for the projections involved in the two passes – these could be different depending on the frusta for the light and eye. The projection transformation will also be applied during the texture coordinate generation phase which is depicted in Figure 8 for OpenGL. As shown in the figure, two transformations are applied to the eye coordinates – the *texgen* planes, and the *texture matrix*. For *eye linear* texgen planes, OpenGL will automatically multiply the eye coordinates with the inverse of the modelview matrix *in effect when the planes are specified*. (See Appendix A for a more detailed explanation of the texgen planes in the eye linear case.)

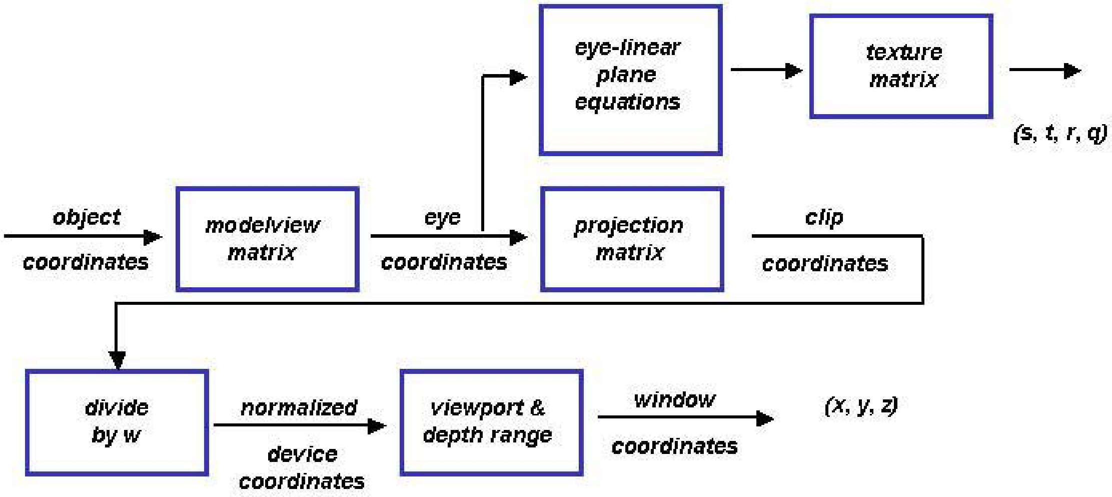


Figure 8: OpenGL Transformation Pipeline

The resulting texture coordinates are therefore computed as:

$$\mathbf{[x_e,y_e,z_e,w_e]^T = (modelview)\ [x_o,y_o,z_o,w_o]^T}$$

$$\mathbf{E_e = E_{po}(modelview_{po})^{-1}}$$

$$\mathbf{[s,t,r,q]^T = T\ E_e\ [x_e,y_e,z_e,w_e]^T}$$

Equation 1

Here the subscript '**o**' denotes object space coordinates, and the subscript '**e**' refers to eye space coordinates, $\mathbf{modelview_{po}}$ is the modelview matrix in effect when the eye linear texgen plane equations are specified, $\mathbf{E_{po}}$ is the matrix composed of the eye linear plane equations *as specified to OpenGL* (i.e. in their own object space), $\mathbf{E_e}$ is the matrix composed of the transformed plane equations (these are the plane equations that are actually stored by OpenGL), **T** is the texture matrix, and **modelview** is the modelview matrix when rendering the scene geometry.

Setting Up the Transformations

We want to set the transformations in Equation 1 to compute texture coordinates **(s,t,r,q)** such that **(s/q,t/q)** will be the fragment's location within the depth texture, and **r/q** will be the window-space z of the fragment relative to the light's frustum. In other words, we want to compute:

$$\mathbf{[s,t,r,q]^T = S\, P_{light}\, L^{-1}\, M\, [x_o,y_o,z_o,w_o]^T}$$

Equation 2

Here, **S** is the scale-bias matrix, given by:

$$\begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\mathbf{P_{light}}$ is the projection matrix for the light frustum. The "texgen matrix" ($\mathbf{E_e}$), however, is applied to *eye* coordinates $\mathbf{[x_e,y_e,z_e,w_e]^T}$ but we want to generate the coordinates in *light* space, since that is where the depth map computation takes place. So we need to take $\mathbf{[x_e,y_e,z_e,w_e]^T}$ back into world space by applying the transform **V**. That is, we want to compute $\mathbf{[s,t,r,q]^T}$ as:

$$\mathbf{[s,t,r,q]^T = S\, P_{light}\, L^{-1}\, V\, [x_e,y_e,z_e,w_e]^T}$$

Equation 3

Note that the right hand side of Equation 3 reduces to $\mathbf{S\, P_{light}\, L^{-1}\, M\, [x_o,y_o,z_o,w_o]}$, precisely what we want. A straightforward way to compute Equation 3 is to set $\mathbf{modelview_{po}}$ to *identity* and set:

$$\mathbf{T\, E_{po} = S\, P_{light}\, L^{-1}\, V}$$

Equation 4

The first observation is that we have two matrices **T** (the texture matrix) and $\mathbf{E_{po}}$ (the eye linear texgen planes specified to OpenGL) so we can compute Equation 4 in several ways. Since we are going to have to set the eye linear planes in any case, the less expensive thing to do is to not set the texture matrix at all, and use the texgen matrix **G** for the entire computation[†], i.e., set

$$\mathbf{E_{po} = S\ P_{light}\ L^{-1}\ V}$$

Equation 5

This assumes that the modelview matrix, $\mathbf{modelview_{po}}$, was identity at the time the texgen planes are set. Another improvement is to make use of the fact that OpenGL automatically multiplies $\mathbf{[x_e,y_e,z_e,w_e]^T}$ with $(\mathbf{modelview_{po}})^{-1}$ for eye linear texgen. The sole purpose of using **V** in Equation 5 is to eliminate $\mathbf{V^{-1}}$. If we set $\mathbf{modelview_{po}} = \mathbf{V^{-1}}$, then OpenGL will do the elimination for us and we can avoid having to compute **V**, the inverse of the view matrix. The steps can be summarized as follows:

First Pass (Depth Map Generation)

- Render from light's point of view. Set projection matrix to $\mathbf{P_{light}}$. Set the view portion of the modelview matrix to $\mathbf{L^{-1}}$.
- Render scene (with appropriate modeling transform(s) **M**).

Second Pass (Depth Map Comparison)

- Render from eye's point of view. Set projection matrix to be $\mathbf{P_{eye}}$. Set the view portion of the modelview matrix to be $\mathbf{V^{-1}}$.
- Set texgen to be EYE_LINEAR. Specify texgen planes as $\mathbf{E_{po}} = \mathbf{S\ P_{light}\ L^{-1}}$
- Render scene (with appropriate modeling transform(s) **M**)

Conclusions

Shadow mapping is an easy-to-use shadowing technique that makes 3D rendering just look better. It enjoys hardware acceleration on GeForce3 GPUs. There is example source code in the NVSDK (hw_shadowmaps_simple, hw_woo_shadowmaps) that demonstrate the technique, and the corresponding OpenGL extensions. Please direct questions or comments to cass@nvidia.com.

References

[1] Craig Duttweiler. Mapping Texels to Pixels in Direct3D. http://developer.nvidia.com/view.asp?IO=Mapping_texels_Pixels.

[†] Note that this technique of collapsing the texture and texgen matrices works in our case because we are setting all four planes, and using the same mode for all four planes. In general, each coordinate can have a different mode (eye linear, object linear, sphere map…) and the technique may not be applicable.

[2] Cass Everitt. Interactive Order-Independent Transparency. Whitepaper: http://developer.nvidia.com/view.asp?IO=Interactive_Order_Transparency.

[3] Mark Kilgard. Shadow Mapping with Today's Hardware. Technical presentation: http://developer.nvidia.com/view.asp?IO=cedec_shadowmap.

[4] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 1.2.1). www.opengl.org

[5] Mark Segal, et al. Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH '92*, pages 249-252, 1992.

[6] OpenGL Extension Registry. http://oss.sgi.com/projects/ogl-sample/registry/.

[7] Yulan Wang and Steven Molnar. Second-Depth Shadow Mapping. UNC-CS Technical Report TR94-019, 1994.

[8] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH '78*, pages 270-274, 1978.

[9] Andrew Woo, P. Poulin, and A. Fournier. "A Survey of Shadow Algorithms," IEEE Computer Graphics and Applications: vol 10(6), pages 13-32, 1990.

Appendix A: Another Way to Think about EYE_LINEAR planes in OpenGL

An unfortunate thing about EYE_LINEAR texgen in OpenGL is that the name implies that the plane equations are specified in eye space, when they are, in fact, specified in their own object space. There are two ways one can think about the planes specified in EYE_LINEAR texgen. As mentioned earlier, OpenGL will automatically multiply the planes specified with $(\mathbf{modelview_{po}})^{-1}$, i.e. the inverse of the modelview matrix in effect when the planes are specified. From Equation 1 we see that the net effect is to map the vertex position in eye coordinates $\mathbf{[x_e,y_e,z_e,w_e]^T}$ back to the 'object space' defined by $(\mathbf{modelview_{po}})^{-1}$. The transformed coordinates are then evaluated at each plane in this object space to get the texture coordinates. An alternate way to think about the texgen planes is to consider the matrix $\mathbf{E_e} = \mathbf{E_{po}}(\mathbf{modelview})^{-1}$, which defines a map whose domain is *eye* space, with the planes $\mathbf{E_{po}}$ being specified in object space. $\mathbf{E_e}$ therefore defines the *transformed* planes in eye space. In either case, the planes are being *specified* in the 'object space' defined by $(\mathbf{modelview_{po}})^{-1}$ and *not* in eye space.

In the shadow mapping case described earlier, the modelview matrix is set to $\mathbf{V^{-1}}$ when the texgen plane equations are specified. This is the same thing as saying that we are specifying the plane equations in *world* space. If the modelview matrix were set to identity, then we would be specifying the equations in *eye* space. The same is true if we were specifying vertex positions.

We could set the modelview matrix to $\mathbf{V}^{-1}\mathbf{L}$, and specify the plane equations in *light* space. This might be handy, because we would only need to update our plane equations if the light's projection ($\mathbf{P}_{\mathbf{light}}$) changed. We could even put the *whole* transformation into the modelview matrix as $\mathbf{V}^{-1}\mathbf{L}\mathbf{P}_{\mathbf{light}}{}^{-1}\mathbf{S}^{-1}$. In this case, the texgen planes are *always* just specified as identity ($\mathbf{E}_{\mathbf{po}} = \mathbf{I}$)!

Interactive Order-Independent Transparency

Cass Everitt
NVIDIA OpenGL Applications Engineering
cass@nvidia.com

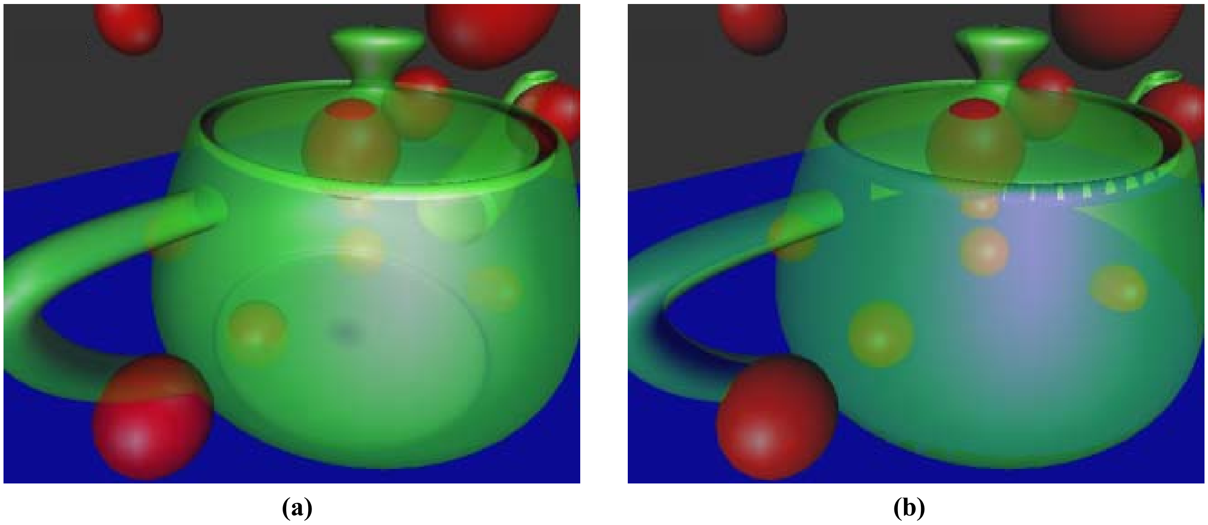


Figure 1. These images illustrate correct (a) and incorrect (b) rendering of transparent surfaces.

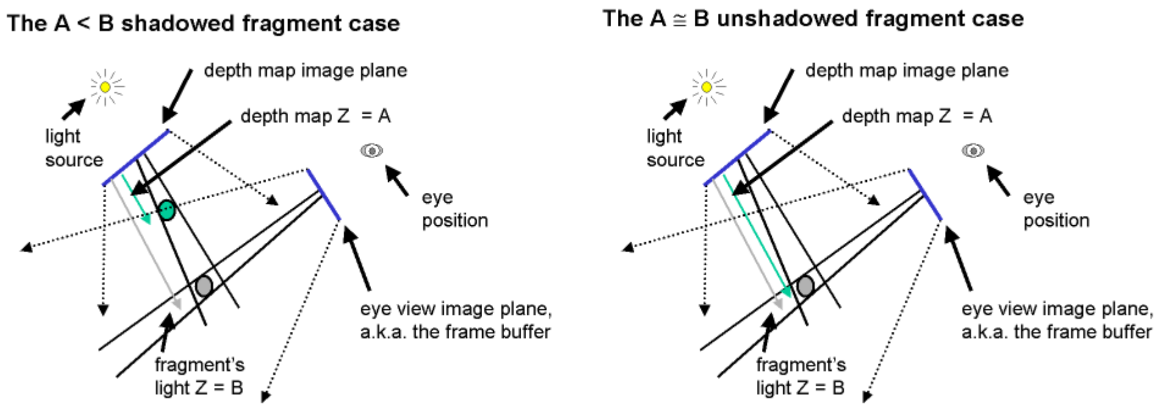
Introduction

Correctly rendering non-refractive transparent surfaces with core OpenGL functionality [9] has the vexing requirements of depth-sorted traversal and non-intersecting polygons. This is frustrating for most application developers using OpenGL because the natural order of scene traversal (usually one object at a time) rarely satisfies these requirements. Objects can be complex, with their own transformation hierarchies. Even more troublesome, with advanced graphics hardware, the vertices and fragments of objects may be altered by user-defined per-vertex or per-fragment operations within the GPU. When these features are employed, it becomes intractable to guarantee that fragments will arrive in sorted order for each pixel. The technique presented here solves the problem of order dependence by using a technique we call *depth peeling*. *Depth peeling* is a fragment-level depth sorting technique described by Mammen using *Virtual Pixel Maps* [7] and by Diefenbach using a *dual depth buffer* [3]. Though no dual depth buffer hardware fitting Diefenbach's description exists, Bastos observed that shadow mapping hardware in conjunction with alpha test can be used to achieve the same effect [2]. Using this variation of depth peeling, each unique depth in the scene is extracted into layers, and the layers are composited in depth-sorted order to produce the correctly blended final image. The peeling of a layer requires a single order-independent pass over the scene. Figure 1 contrasts correct and incorrect rendering of transparent surfaces.

The goal of this document is to enable OpenGL developers to implement this technique with NVIDIA OpenGL extensions and GeForce3 hardware. Since shadow mapping is integral to the technique a very basic introduction is provided, but the interested reader is encouraged to explore the referenced material for more detail.

Shadow Mapping

Shadow mapping is a multi-pass shadowing technique developed by Lance Williams [11] in 1978. In the first pass, the scene is rendered from the light's point of view. The depth buffer generated in that pass is copied to a special "depth texture" or shadow map. In the second pass, the shadow map is projected onto the scene using projective texture mapping [10, 4]. Unlike regular 2D projective texture mapping where the *r* coordinate is unused, we use the *r* coordinate to compute the distance of the rasterized fragment to the light source. Then, the lookup of (*s*,*t*) is the distance to the *nearest* surface to the light source (along that direction). If $r \leq$ lookup(*s*,*t*), then the current fragment is *visible to the light source*, and therefore *not in shadow*. Essentially, we use depth-buffering in the first pass to determine which surfaces are visible from the light's point of view, and in the second pass we show those surfaces as illuminated. Figure 2 helps illustrate this concept.


The A < B shadowed fragment case

The A ≅ B unshadowed fragment case

Figure 2. These diagrams were taken from Mark Kilgard's shadow mapping presentation at GDC 2001. They illustrate the shadowing comparison that occurs in shadow mapping.

We use the `SGIX_shadow` and `SGIX_depth_texture` extensions [8] to take advantage of GeForce3 shadow mapping hardware in OpenGL.[1] The `SGIX_shadow` extension provides the ability to compute a comparison of the *r* texture coordinate with the results of the 2D lookup. The `SGIX_depth_texture` extension exposes `GL_DEPTH_COMPONENT` internal texture formats and defines semantics for `glCopyTex{Sub}Image2D` for fast copies from the depth buffer to a depth texture. These features are fully accelerated on GeForce3.

[1] The OpenGL Architectural Review Board (the "ARB") has since standardized the `ARB_shadow` and `ARB_depth_texture` extensions in February 2002. These extensions are very similar to the `SGIX` extensions. The technique described here could use either the `ARB` or `SGIX` extensions. NVIDIA drivers after March 2002 support the `ARB` and `SGIX` extensions for shadow mapping.

It has been shown by Heidrich [5] that multitexturing can be used to implement a limited form of shadow mapping. It is limited in that it requires multiple texture units and it only supports *nearest* filtering and 8-bit depth texels (16-bit depth on GeForce [6]). For depth peeling, we need full depth buffer precision (24 bits) that necessitates the use of the `SGIX` shadowing extensions.

Depth Peeling

Depth peeling is the underlying technique that makes this approach for order-independent transparency possible. The standard depth test gives us the nearest fragment at each pixel, but there is also a fragment that is second nearest, third nearest, and so on. Standard depth testing gives us the *nearest* fragment without imposing any ordering restrictions, however, it does *not* give us any straightforward way to render the second nearest or nth nearest surface.

Depth peeling solves this problem. The essence of what happens with this technique is that with *n* passes over a scene, we can get *n* layers deeper into the scene. For example, with 2 passes over the scene, we can extract the nearest and second nearest surfaces in a scene. We get both the depth and color (RGBA) information for each layer.

The images we get from peeling away depth are shown in Figure 3. It can be quite confusing to make sense of the images of layer 1 and beyond, because the notion of a

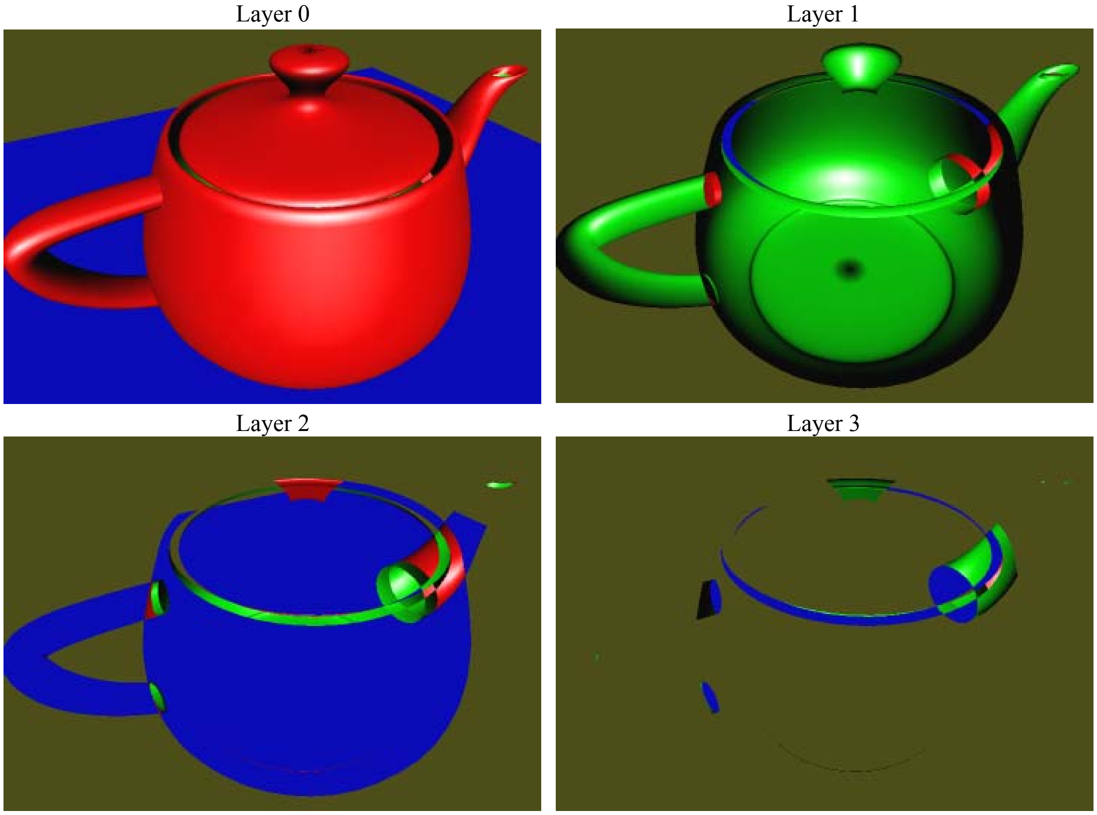


Figure 3. These images illustrate simple depth peeling. Layer 0 shows the nearest depths, layer 1 shows the second depths, and so on. Two-sided lighting with vivid coloring is used to help distinguish the surfaces.

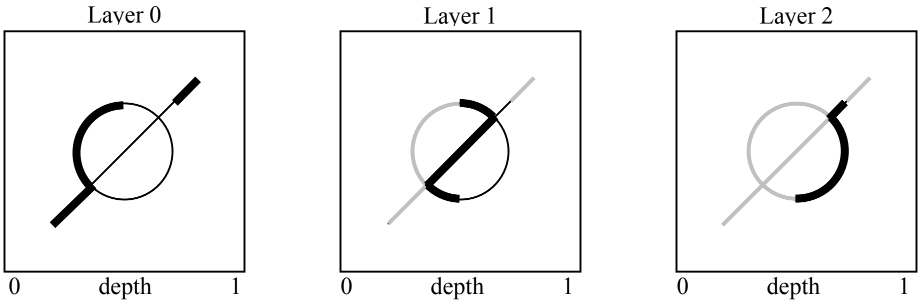


Figure 4. Depth peeling strips away depth layers with each successive pass. The frames above show the frontmost (leftmost) surfaces as bold black lines, hidden surfaces as thin black lines, and "peeled away" surfaces as light grey lines.

"second nearest surface" is unintuitive. To help distinguish the various surfaces, the teapot is rendered with two-sided lighting (outside is red and inside is green), and the ground plane is drawn in blue. Note that the image labeled 'Layer 2' is in the shape of a teapot, but most of the fragments in that layer are from the ground plane (they are blue). Without the coloring, this would be difficult to interpret.

Figure 4 provides a more diagrammatic view of depth peeling. The diagrams there are analogous to the images in Figure 3, except we are now looking at a cross section of the view volume and highlighting each layer. It is evident from the view in Figure 4 that the depths vary within each layer, and the number of samples is decreasing. The peeling process clearly happens at the fragment level, so the pieces are generally not whole polygons.

The process of depth peeling is actually a straightforward multi-pass algorithm. In the first pass we render as normal, and the depth test gives us the nearest surface. In the second pass, we use the depth buffer computed in the first pass to "peel away" depths that are less than or equal to nearest depths from the first pass. The second pass generates a depth buffer for the *second* nearest surface, which can be used to peel away the first and second nearest surfaces in the third pass. The pattern is simple, but there is a catch. We need to perform *two depth tests per fragment* for it to work!

Multiple Depth Tests

The most natural way to describe this technique is to imagine that OpenGL supported multiple simultaneous depth units, each with its own depth buffer and associated state. We diverge from Diefenbach's dual depth buffer API in that we assume there are *n* depth units, all writeable, that are executed in sequential order. The first depth test to fail discards the fragment and terminates further processing. The pseudocode in Listing 1 implements depth peeling using two depth units.

```
for (i=0; i<num_passes; i++)
{
    clear color buffer
    A = i % 2
    B = (i+1) % 2
    depth unit 0:
        if(i == 0)
            disable depth test
        else
            enable depth test
        bind buffer A
        disable depth writes
        set depth func to GREATER
    depth unit 1:
        bind buffer B
        clear depth buffer
        enable depth writes
        enable depth test
        set depth func to LESS
    render scene
    save color buffer RGBA as layer i
}
```

In each pass except the first, depth unit 0 is used to peel away the previously nearest fragments while the depth unit 1 performs "regular" depth-buffering. We decouple the depth buffer from the depth unit because it simplifies the presentation of the algorithm and more closely matches the semantics of `ARB_multitexture`. This decoupling is convenient because we need to use the depth buffer produced by depth unit 1 in pass *i* as the "peeling" depth buffer for depth unit 0 in pass *i*+1.

It is also worth mentioning that we only enable depth writes on depth unit 1. This will be important later.

Shadow Mapping as Depth Test

Shadow mapping *is* a depth test. For the purposes of our discussion, there are only a few major differences between shadow mapping and the depth-buffer algorithm:

- the shadow mapping comparison sets a fragment color attribute,
- the shadow mapping depth test is not tied to the camera position, and
- the shadow map (depth buffer) is not writeable during the shadow comparison (depth test).

It is not difficult to compensate for these differences. We write the results of the shadow mapping comparison to fragment alpha and use alpha test to discard fragments that fail the "depth test" we have chosen. We make the orientation and resolution of the shadow map identical to that of the camera. We can then use shadow mapping as a read-only depth test. This is good news, because this is all we needed to implement depth peeling as described in the previous section using our imaginary multiple depth test OpenGL. Except now, we can actually implement it using real OpenGL and *with hardware acceleration*!

An Invariance Issue

As simple as depth peeling sounds, it is actually pretty intolerant to variance. Due to the nature of the technique, many of the fragments generated in each pass will be on the razor's edge of the comparison. In our imaginary OpenGL that supports multiple depth tests, we would not expect variance to be a problem because we are re-using the same interpolator to compute depth the same way in each pass. Things are a little more complicated when we use shadow mapping as a depth test, though. This is primarily because

- z_w (window space z) is interpolated linearly in *window space* at the precision of the current depth buffer, and
- *r* and *q* are interpolated linearly in *clip space* (hyperbolically in window space) at high precision

The possible differences in precision and/or interpolation implementation are the hazards that cause variance. Consider the depth interpolation in Equation 1, which is linear in window space.

Listing 1. Pseudocode for depth peeling using multiple simultaneous depth buffers.

Where z_w is window space z, z_c is clip space z, w_c is clip space w, and the numeric specifiers 1 and 2 indicate two points that are being interpolated. When we perform shadow mapping, we must interpolate quantities as texture coordinates which vary

$$r = z_c = \frac{\alpha \frac{z_{c1}}{w_{c1}} + (1-\alpha)\frac{z_{c2}}{w_{c2}}}{\alpha \frac{1}{w_{c1}} + (1-\alpha)\frac{1}{w_{c2}}} \tag{2}$$

linearly in clip space, so we interpolate z_c and w_c as the r and q texture coordinates

$$q = w_c = \frac{\alpha \frac{w_{c1}}{w_{c1}} + (1-\alpha)\frac{w_{c2}}{w_{c2}}}{\alpha \frac{1}{w_{c1}} + (1-\alpha)\frac{1}{w_{c2}}} \tag{3}$$

respectively, and use the r/q quotient to produce a value that varies linearly in window space. For the particular case we have been considering, shadow mapping from the camera's point of view we get Equations 2 and 3.

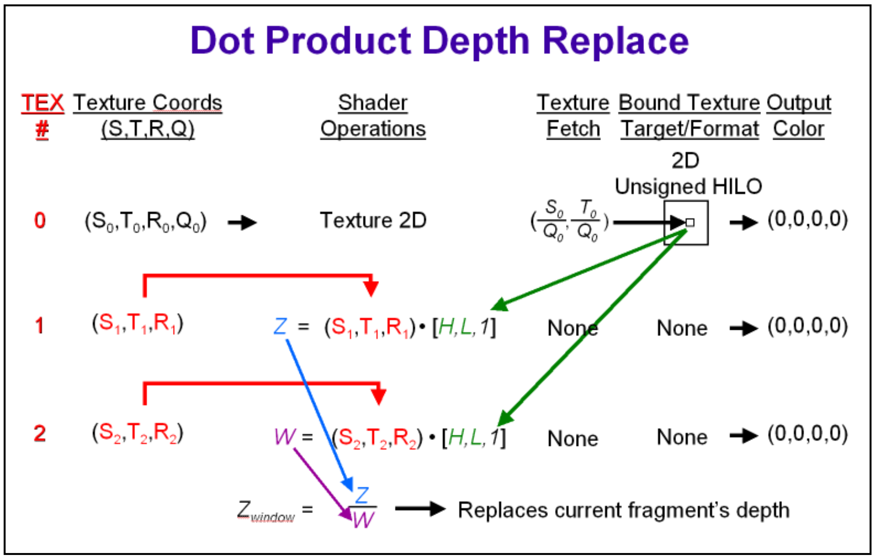
When we compute the r/q quotient, we recognize that the denominators in Equations 2 and 3 cancel, and that for our special case of shadow mapping from the camera's point of view, the numerator of Equation 3 is 1. This leaves only the numerator of Equation 2, which is *identical* to the expression in Equation 1. While this is algebraically true, the hardware may not be able to make some of these cancellations. For fragments with the

$$\frac{\left(\frac{\left(\frac{z_c}{w_c}\right)}{\left(\frac{1}{w_c}\right)}\right)}{\left(\frac{\left(\frac{w_c}{w_c}\right)}{\left(\frac{1}{w_c}\right)}\right)} \le \frac{z_c}{w_c} \tag{4}$$

same depth, hardware could evaluate the comparison shown in (4). The left side of the expression interpolates three quantities and performs four divides while the right simply interpolates one quantity.

$$z_w = \frac{z_c}{w_c} = \alpha z_{w1} + (1-\alpha) z_{w2} = \alpha \frac{z_{c1}}{w_{c1}} + (1-\alpha)\frac{z_{c2}}{w_{c2}} \tag{1}$$

Luckily GeForce3's `NV_texture_shader` extension [8] supports a mode called `GL_DOT_PRODUCT_DEPTH_REPLACE_NV` that allows us to compute fragment depth using texture coordinates. The depth computed in this texture shader replaces the fragment depth that was computed in the rasterizer. This means that for GeForce3, we can compute the depth that we store in the depth buffer in exactly the same way that we compute it when making the comparison. When we use this texture shader in generating our shadow map, there are no variances in the least significant bits. This is nice because it means we do not have to employ fudge factors to deal with LSB variance. The depth replace texture shader is very general, and this is a very simple use of it. Figure 5 illustrates the general operation of the depth replace texture shader.



Dot Product Depth Replace

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	(S_0,T_0,R_0,Q_0)	Texture 2D	$(\frac{S_0}{Q_0},\frac{T_0}{Q_0})$	2D Unsigned HILO	(0,0,0,0)
1	(S_1,T_1,R_1)	$Z = (S_1,T_1,R_1) \cdot [H, L, 1]$	None	None	(0,0,0,0)
2	(S_2,T_2,R_2)	$W = (S_2,T_2,R_2) \cdot [H, L, 1]$	None	None	(0,0,0,0)

$Z_{window} = \frac{Z}{W}$ → Replaces current fragment's depth

Figure 5. This diagram is a slightly modified slide taken from Dominé and Spitzer's GDC 2001 presentation on GeForce3 texture shaders. It describes the depth replace texture shader.

For our purposes, we really only want to interpolate z_c and w_c using a single texture coordinate for each, so we use a 1x1 `GL_UNSIGNED_HILO_NV` texture where H and L are zero. By definition, the 3rd component of an unsigned HILO is 1, so we perform a dot product of (S, T, R) with (0, 0, 1). In this way, we can interpolate the R coordinates of stages 1 and 2, and we use texture coordinate generation to make sure that R_1 is z_c and R_2 is w_c. When we perform the division z_c/w_c at each fragment, we are effectively interpolating window space depth in the same way that s/q does it in the subsequent shadow mapping pass.

There is one clarification we should make. When we consider the standard transformation pipeline, we often place the perspective divide before the viewport and depth range scale and bias. The depth replace texture shader and shadow mapping depth computation perform the divide (z_c/w_c and s/q respectively) as the *final operation*. This means that we must apply the depth range scale and bias *before* the perspective divide. Or, said another way, for depth replace and shadow mapping, we must transform coordinates into homogeneous window coordinates rather than homogeneous clip space.

```
glActiveTextureARB(GL_TEXTURE0_ARB);
simple_1x1_uhilo.bind();
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV, GL_TEXTURE_2D);

matrix4f m;
glActiveTextureARB(GL_TEXTURE1_ARB);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV, GL_DOT_PRODUCT_NV);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_NONE);
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
eye_linear_texgen();  // set EYE_LINEAR texgen with identity planes
texgen(true);         // enable texgen on s,t,r, and q
glPopMatrix();
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glTranslatef( 0, 0,.5);
glScalef( 0, 0, .5);
reshaper.apply_perspective();  // apply the camera’s perspective projection matrix
glMatrixMode(GL_MODELVIEW);

glActiveTextureARB(GL_TEXTURE2_ARB);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV, GL_DOT_PRODUCT_DEPTH_REPLACE_NV);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_NONE);
glPushMatrix();
glLoadIdentity();
eye_linear_texgen();  // set EYE_LINEAR texgen with identity planes
texgen(true);         // enable texgen on s,t,r, and q
glPopMatrix();
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
m(0,0) = 0;   m(0,1) = 0;   m(0,2) = 0;   m(0,3) = 0;
m(1,0) = 0;   m(1,1) = 0;   m(1,2) = 0;   m(1,3) = 0;
m(2,0) = 0;   m(2,1) = 0;   m(2,2) = 0;   m(2,3) = 1;  // move q to r
m(3,0) = 0;   m(3,1) = 0;   m(3,2) = 0;   m(3,3) = 0;
glMultMatrix(m);
reshaper.apply_perspective();  // apply the camera’s perspective projection matrix
glMatrixMode(GL_MODELVIEW);

glActiveTextureARB(GL_TEXTURE3_ARB);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV, GL_TEXTURE_RECTANGLE_NV);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_NONE);

glActiveTextureARB(GL_TEXTURE0_ARB);
```

Listing 2. Example code for setting up depth replace texture shader for use in depth peeling.

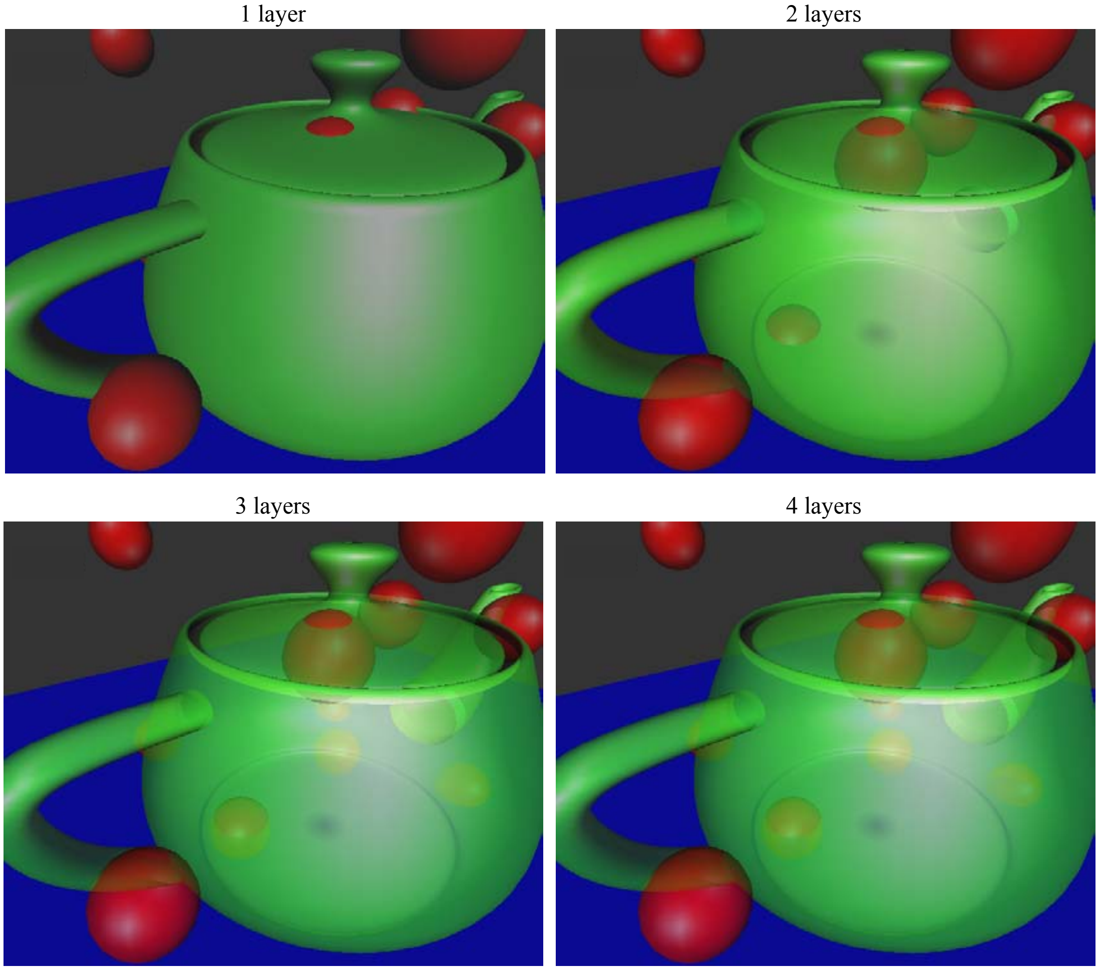
The code in Listing 2 illustrates how to set up the `GL_DOT_PRODUCT_DEPTH_REPLACE_NV` texture shader to compute window *z* in a way that closely matches the standard projective texture mapping computation of window *z*. For illustrative purposes, we use *eye linear* texgen with an identity mapping for the *r* coordinate [0 0 1 0], and we use the texture matrix to perform the transforms. The most efficient approach would be to encode the transform in the texgen plane.

Another slightly odd aspect the depth replace texture shader is illustrated in the code in Listing 2. It is that the homogeneous window coordinate must be moved from the fourth row of the texture matrix into the *r* coordinate. This is because the dot product texture shaders only perform a 3-component dot product, so all quantities must be in the *s*, *t*, or *r* coordinates.

Putting It All Together

Now we have a way to compute the RGBA color for each unique depth at every pixel. These are stored as separate layers (or viewport-sized textures). All that remains is to compute the correct order-dependent color at each pixel by compositing the layers in order. Rendering each layer as viewport-sized textured quad does this. For back-to-front compositing a (`GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`) blending function is used.

Figure 5 illustrates the results of compositing of the layers into a final image. Note also that the bottom two images in Figure 5 look virtually (but not completely) identical. For completely correct results we should extract every semitransparent sample up to the first opaque sample, but in practice that is not necessary. The nature of the transparency computation is that samples further back have diminished effect, so truncation is a reasonable (and efficient) form of approximation. For example, the scene in Figure 5 is "good enough" after three layers.



1 layer

2 layers

3 layers

4 layers

Figure 5. The depth peeled layers of the scene are correctly sorted per-fragment. If we simply save the color (RGBA) for each layer, we can composite them in depth-sorted order as a final pass. These images illustrate blending more layers for more correct transparency.

Conclusion

The technique presented is a straightforward and convenient way to render scenes with transparency because it does not require that the scene be rendered in sorted order, and it makes good use of graphics hardware. In addition, there may be no practical alternative to this approach of layer extraction and compositing for scenes that cannot be rendered in sorted order in a single pass.

Some of the figures in this paper come from the `layerz` and `order_independent_-transparency` demos that can be found in the NVIDIA OpenGL SDK, which can be found at http://www.nvidia.com/developer. The demos only illustrate the technique described here, but many variations like those described in Diefenbach [3] are possible. The GDC 2001 presentations that were used in some figures are also available at the above web site.

Acknowledgements

Rui Bastos came up with the very clever idea of *depth peeling* using shadow mapping hardware support when he was considering hardware accelerated Woo shadow maps for GeForce3 (whitepaper on that topic to follow soon). I had help from Mark Kilgard on the appropriate texture coordinate generation setup for the *depth replace* texture shader that solves the invariance problem. He also provided invaluable feedback on early drafts of this paper.

References

[1] James F. Blinn. Hyperbolic interpolation. *IEEE Computer Graphics (SIGGRAPH) and Applications*, 12(4):89 94, July 1992.

[2] Rui Bastos. Personal communication. Feb 2001.

[3] Paul Diefenbach. Pipeline Rendering: Interaction and Realism Through Hardware-Based Multi-PassRendering. University of Pennsylvania, Department of Computer Science, Ph.D. dissertation, 1996.

[4] Cass Everitt. Projective texture mapping. http://developer.nvidia.com. 2001

[5] Wolfgang Heidrich. High quality shading and lighting for hardware-accelerated rendering. http://www.cs.ubc.ca/~heidrich/Papers/phd.pdf. 1999.

[6] Mark Kilgard. GDC 2001 – Shadow mapping with today's OpenGL hardware. http://developer.nvidia.com. March 2001.

[7] Abraham Mammen. Transparency and antialiasing algorithms Implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9(4): 43-55, July 1989.

[8] NVIDIA OpenGL Extensions Specifications. http://developer.nvidia.com/view.asp?IO=nvidia_opengl_specs. March 2001.

[9] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 1.2.1). www.opengl.org

[10] Mark Segal, et al. Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH '92*, pages 249-252, 1992.

[11] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH '78*, pages 270-274, 1978.

Fast and Approximation Computation of Geometric Arrangements using GPUs I

Shankar Krishnan
AT & T Research Labs

Application of the Two-Sided Depth Test to CSG Rendering

Sudipto Guha
Univ. of Pennsylvania
sudipto@cis.upenn.edu

Shankar Krishnan
AT&T Labs–Research
krishnas@research.att.com

Kamesh Munagala*
Stanford University
kamesh@cs.stanford.edu

Suresh Venkatasubramanian
AT&T Labs–Research
suresh@research.att.com

Abstract

Shadow mapping is a technique for doing real-time shadowing. Recent work has shown that shadow mapping hardware can be used as a *second depth test* in addition to the z-test. In this paper, we explore the computational power provided by this second depth test by examining the problem of rendering objects described as CSG (Constructive Solid Geometry) expressions. We provide an algorithm that asymptotically improves the number of rendering passes required to display a CSG object by a factor of n by exploiting the two-sided depth test. Interestingly, a matching lower bound can be proved demonstrating that our algorithm is optimal.

Keywords: Constructive solid geometry, Graphics hardware, Z-buffer, Shadow mapping

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems

1 Introduction

In recent years, the increased power of graphics rendering hardware has led to the use of the graphics pipeline for general purpose stream computations. One of the early examples of this was the use of hierarchical z-buffering for visibility calculations [Greene et al. 1993], and subsequently in programmable vertex shaders [Peercy et al. 2000; Lindholm et al. 2001; Proudfoot et al. 2001]. Other uses of the graphics pipeline as a general purpose stream computing engine have been demonstrated in computational geometry[Hoff III et al. 1999; Mustafa et al. 2001; Krishnan et al. 2002], robotics[Hoff et al. 2000], numerical analysis[Larsen and McAllister 2001], and ray tracing[Purcell et al. 2002].

In a recent development, work by Everitt *et al.* [2002] has shown that the shadow mapping hardware (supported in the nVidia GeForce3 and newer architectures) can be used to perform order-independent transparency. They demonstrate this by using the shadow mapping phase in the pipeline to filter out fragments that have a z-value less than (or greater than) values stored in a depth texture. This operation, combined with the standard z-test, provides a two-sided depth test on fragments. This feature is exploited in a technique they call *depth peeling* that can "peel" off layers of a scene one by one. Interestingly enough, the idea of using two-sided depth tests to implement depth peeling was proposed earlier by Mammen [1989], who used the idea of *virtual pixel maps*. The key observation by Everitt *et al.* was that existing shadow mapping hardware can be used to simulate this test.

Our Contributions In this paper, we study the computational power of the two-sided depth test in the context of rendering objects represented as CSG trees.

- We show that the two-sided depth test can be used to render CSG trees with a factor of n (number of primitives in the CSG expression) fewer passes than the best known OpenGL-based algorithms (see Section 2 for more details).
- Our algorithm can render arbitrary CSG objects, and does not require the explicit precalculation of levels that prior results did.
- We use no external storage or readbacks; all computations are performed directly on the GPU.
- Our algorithm works by performing a *topological sweep* over the arrangement of the objects; this technique may be of independent interest.

Paper Outline The rest of the paper is organized as follows. We discuss prior work in Section 2. We define the problem of rendering a CSG tree in Section 3, and present our solution for a single product in Section 4. The solution is extended to arbitrary CSG expressions in Section 5. We discuss implementation details and present our algorithm performance in Section 6 and we conclude in Section 7.

2 Prior Work

There has been extensive work on the problem of rendering solid objects defined in terms of CSG trees. A general survey of CSG methods is beyond the scope of this paper. We will focus solely on methods that make use of the graphics hardware.

Goldfeather *et al.* [1986] presented an algorithm for rendering a CSG tree of *convex* objects (and subsequently [1989] for non-convex objects) using an extension of the Pixel-Planes graphics hardware [Fuchs and Poulton 1981]. This algorithm was refined and implemented on modern graphics hardware by Wiegand [1996]. The running time of the algorithm, expressed as the number of rendering passes required, is essentially quadratic in the number of primitives (the running time also includes a quadratic term that depends on the *convexity* of the objects).

The Trickle algorithm, developed by Epstein *et al.* [1989] and later refined by Rossignac and Wu [1992], takes a different approach using "depth-interval buffers" (which essentially provide the functionality of a two-sided test) to do the rendering. Their approach requires three depth buffers, the two-sided depth test and two color buffers, and thus is not readily adaptable to current OpenGL-based architectures. Although they do not analyze their algorithm in terms of rendering passes, we believe that their approach (for each product) requires number of passes proportional to the depth complexity from the given viewpoint.

*Supported by NSF grant CCR-0113217

Stewart *et al.* [1998] presented an improvement to the Goldfeather *et al.*algorithm that takes into account the fact that objects may be disjoint and thus can be rendered in parallel. If the depth complexity of a collection of n primitives is k, then the modification proposed by Stewart *et al.*requires $O(kn)$ rendering passes. In the case when the primitives are sufficiently disjoint in screen space (and thus $k < n$), this algorithm is superior. Erhart and Tobler [2000] provide a modification to this algorithm that yields more accurate results (in terms of depth tests). However, in the worst case, their algorithm again requires $O(n^2)$ passes.

More recently, Stewart *et al.* [2000; 2002] present improvements that compute a CSG product in a constant number of passes when all the primitives are convex. They use a universal sequence to model the depth ordering of the primitives without having to compute an explicit front-to-back ordering. The caveat with this approach is that a quadratic number of objects are rendered in each pass (because primitives are duplicated).

All of the algorithms above compute a union of objects by merging the partial depth buffers obtained for each product. This merging step is performed via the use of readbacks, and is thus slow.

3 CSG Trees and Normalization

A three dimensional object can be described as the result of performing set operations ($\cup, \cap, \setminus$) on a ground set of shapes (called *primitives*). A *CSG tree* can be used to define an object by defining the sequence of operations that are performed.

The CSG tree is usually assumed to be in a canonical form to aid in rendering. A CSG tree is said to be in *sum-of-products* form if the expression it defines can be written as a union of intersections/subtractions (a sum of products). Such a tree is said to be *normalized*. Goldfeather *et al.* [1986] provide an algorithm for normalizing a CSG tree; we use their technique, and the rest of the paper assumes without loss of generality that the CSG tree has been normalized.

Given a normalized CSG tree and a procedure to compute the product of a set of primitives, unions can be computed easily by merging the results of individual products in the depth buffer. The above mentioned algorithms make use of this observation, and thus focus on the problem of rendering a CSG tree denoted by a single product. For clarity of presentation, we will explain the working of our algorithm on a single product, and subsequently we will show how the same ideas can be extended to render a sum of products.

3.1 Notation and Preliminaries

We denote a normalized CSG expression as $P_1 \cup \cdots \cup P_m$, where each P_i is a product of primitives. A single product is a general expression involving intersections and complementations. Consider a single product $P = (((\mathbf{o}_1 \cap \mathbf{o}_2) \setminus \mathbf{o}_3) \cap \mathbf{o}_4)$. P can be rewritten as $\mathbf{o}_1 \cap \mathbf{o}_2 \cap \overline{\mathbf{o}_3} \cap \mathbf{o}_4$. Thus each product is the intersection of a set of (possibly complemented) objects. For a product P, let $U(P)$ denote the set of uncomplemented objects and $C(P)$ denote the set of complemented objects. In this example, $U(P) = \{\mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_4\}$ and $C(P) = \{\mathbf{o}_3\}$.

Every object $\mathbf{o}$ is a collection of alternating front and back faces (or *layers* [Goldfeather et al. 1989]) as seen from the viewpoint. The *depth* $d(P)$ of a product P is the maximum number of layers in P (with respect to the viewpoint).

4 Rendering A Product

Consider a product P. Our algorithm works by traversing the layers of the primitives in P in a front-to-back order. It is easy to see that only the front faces of uncomplemented primitives and back faces of complemented primitives contribute to the visible portions of P. Hence, only these faces (termed *appropriate primitives*) are considered by our algorithm in the front-to-back traversal. Once a pixel in P is found, no further updates are made for this pixel. Thus the algorithm maintains a FRONT of all pixels (with associated depth values in the z-buffer) for which membership in P has not yet been determined. In each step,

1. We test which pixels in the FRONT satisfy membership in P.
2. If the pixel fails the membership test then its depth is updated to the depth of the next face in the front-to-back ordering – this is called *advancing the* FRONT .
3. If the pixel passes the membership test, a mask is applied to ensure its depth value is not updated in subsequent steps.

After the algorithm has traversed all layers the z-buffer holds the depth field of P. We then render all the objects with depth test set to `EQUAL` to obtain P. We illustrate the working of the algorithm in Figure 1.

Testing Product membership Assume that a point p is in the product and its depth is stored in the z-buffer. Goldfeather *et al.* [1986] made the observation that (a) if a primitive $\mathbf{o}$ occurs uncomplemented in a product, the number of layers of that primitive (both front and back) that have depth greater than p must be *odd*, and similarly (b) the number of layers (with depth greater than p) must be *even* if the object occurs complemented.

Let $f(\mathbf{o}, p)$ denote the number of front faces of $\mathbf{o}$ of depth greater than the depth of p. Similarly let $b(\mathbf{o}, p)$ denote the number of back faces of $\mathbf{o}$ satisfying the same depth condition. Since all objects are simple and thus have no self-intersections, each front face of $\mathbf{o}$ is followed immediately by a back face of $\mathbf{o}$, and thus $f(\mathbf{o}, p) \leq b(\mathbf{o}, p) \leq f(\mathbf{o}, p) + 1$. For an uncomplemented object, $b(\mathbf{o}, p) - f(\mathbf{o}, p) = 1$ and for a complemented object, $b(\mathbf{o}, p) - f(\mathbf{o}, p) = 0$. For a general product P, we can summarize the $|P|$ equations in a single condition as follows:

$$\sum_{\mathbf{o} \in C(P)} b(\mathbf{o}, p) - \sum_{\mathbf{o} \in C(P)} f(\mathbf{o}, p) + \sum_{\mathbf{o} \in U(P)} b(\mathbf{o}, p) - \sum_{\mathbf{o} \in U(P)} f(\mathbf{o}, p) = |U(P)|$$

It is not difficult to show that only points in the final product will satisfy the above equation. Moreover, this equation is crucial because it allows us to check membership for a point p in two rendering passes (instead of n). We use the stencil buffer to implement this test. We group together the back (front) faces of $U(P)$ and $C(P)$ in a single pass to increment (decrement) the stencil buffer. Pixels whose stencil value is $|U(P)|$ have passed the membership test, and are masked with a suitable value to prevent future depth updates.

Advancing the Front The FRONT is maintained as depth values in the z-buffer. The initial front is obtained by rendering the *appropriate primitives* with the depth-test set to `LESS`. To advance the front, we copy the depth buffer to a shadow buffer, and invoke the *depth peeling* subroutine to pass only those fragments whose depth is greater than the value in the shadow buffer using the alpha test. We refer the reader to [Everitt 2002] for details of the depth peeling algorithm. This test, coupled with the normal z-test (depth test set to `LESS`), provides fragments whose depth value is immediately behind the current front. In our case, we selectively advance the FRONT using the stencil mask. Observe the crucial role of the second depth test provided by the depth peeling routine. Without this, we would be unable to implement the two-sided depth test, $a \leq$ `Zvalue` $< b$, and would not be able to advance the front in a single rendering pass.

The full algorithm is as follows. The sentinel N is used to mask points which have been determined to be in the product: the front is not updated for these pixels.

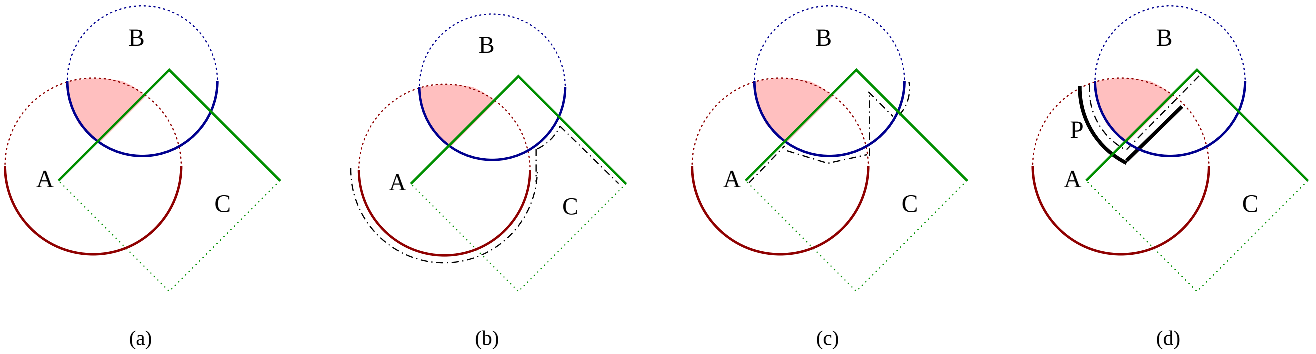


Figure 1: An illustration of the algorithm for a single product. (a) A product $P = A \cap B - C$. The original primitives are dotted and the appropriate primitives are drawn solid. P is shaded. (b) The initial contents of the front (dashed). (c) The front after a single step of the algorithm. (d) Final step: pixels marked solid are in P

Algorithm 1 Algorithm for a single product P

```
Initialize z-buffer to first front.
Initialize stencil buffer with 0
while stencil buffer contains 0 do
    Test points on front for membership in $P$
    Set stencil buffer to $N$ $(> n)$ for pixels passing test
    Set stencil buffer to 0 for pixels not passing test
    Advance front
end while
```

5 Computing Unions of Products

We now describe the computation of a sum of products. Let the products be $P_1, P_2, \dots P_m$. The computation proceeds incrementally. Assume that we have correctly computed $P_1 \cup \dots \cup P_{i-1}$. At the start of the i^{th} step, the z-buffer contains the depth values for $P_1 \cup \dots \cup P_{i-1}$ (denoted by D_{i-1}) and the color buffer contains the appropriate values (denoted by C_{i-1}).

We first copy the content of the z-buffer into a second shadow buffer *buf*. Then, we compute the depth field for P_i using the algorithm described in the previous section. This sets the color buffer appropriately as well. We now need to merge the two depth fields, retaining the minimum value at each pixel. Let the depth and color field for P_i be denoted by d_i and c_i respectively. Thus the new depth field $D_i = min(D_{i-1}, d_i)$. The new color field is

$$C_i = \begin{cases} C_{i-1} & \text{if } D_{i-1} < d_i, \\ c_i & \text{otherwise.} \end{cases}$$

This is accomplished in two phases. In the first phase, we identify those pixels where $d_i > D_{i-1}$ and tag them appropriately using the stencil buffer to be updated in the next phase. This is accomplished by setting the shadow test to pass fragments whose depth is *greater* than D_{i-1} and setting the depth test to EQUAL. The stencil function is set to clear the stencil bits if the depth test passes (i.e the fragment depth is equal to that in d_i). Intuitively, this encodes the two-sided test $D_{i-1} < d = d_i$, where d is the fragment depth. Now, rendering the faces of P_i has the effect of clearing the stencil buffer in all pixels for which the minimum depth is achieved by D_{i-1} i.e all pixels for which $D_i = D_{i-1}$. Note that this is precisely the set of pixels for which the current depth buffer contents are incorrect. The contents of the color buffer can be updated to reflect $P_1 \cup \dots \cup P_i$ in this phase by going through one extra rendering of the faces of P_i at places where the stencil buffer is not cleared.

In the second phase, update the depth buffer to that in D_{i-1} wherever the stencil bits are cleared in the previous phase. We set the shadow test to pass fragments whose depth is *at most* $(\leq)$ D_{i-1}. The depth test is set to GREATER. By rendering all objects in $P_1 \cup \dots \cup P_{i-1}$, this two-sided depth test passes only fragments whose depth value is D_{i-1}. This completes the z-buffer update, and it now contains D_i. The union algorithm can be summarized as follows:

Algorithm 2 Algorithm for a union of products $P_1, \dots, P_m$

```
Initialize shadow buffer buf to 1.
for $i$ = 1 to $m$ do
    Compute product $P_i$

    Set shadow test to greater
    Set depth test to equal
    Set stencil buffer to 0 on depth pass
    Render $P_i$ and update depth and color buffer

    Set shadow test to less_or_equal
    Set depth test to greater
    Set stencil test to equal to 0
    Render $P_1, \dots, P_{i-1}$ and update depth buffer
    Copy depth buffer to shadow buffer buf
end for
```

Running Time Analysis The total number of rendering passes is the number of passes taken to compute each product, plus m passes to compute the union. Therefore the total number of passes is $m + 2\sum_i d(P_i) = O(\sum_i d(P_i))$. This running time is asymptotically superior to all prior techniques by a factor of n, where n is (on average) the number of primitives appearing in each product. Moreover, in our algorithm, we only render while there are pixels whose correct depth is yet to be determined. Therefore, in practice, our algorithm takes much fewer passes than predicted by the above worst-case expression. In contrast, the running times of the previous algorithms is "worst-case": the number of rendering passes required in any run is always the (same) worst-case.

6 Implementation Details

All our code was implemented using C++/OpenGL on a 1.8Ghz/1GB PC running Red Hat 7.3. The graphics card is an nVidia GeForce4 Ti4600. Our system performs no readbacks and uses no intermediate software buffers, while being able to handle *arbitrary* sums of products. For the purpose of this study, we considered four objects described by CSG trees: they are depicted in the right-most lines of Figure 2. In order to evaluate the performance of our scheme, we used two variants of our algorithm; **BASIC** is the algorithm described above, and **CONV** is a modification that processes convex objects more efficiently (however the overall algorithm structure remains the same). We compare both these methods to the algorithm used by Stewart *et al.* [2002], which we refer to as **SCS**. All running times are reported in frames per second. Table 1 summarizes the nature of the inputs and the running times obtained.

Object	#Products/	CSG Rendering		
	#Primitives	**BASIC**	**CONV**	**SCS**
GRID	1/26	26	57	32
HELIX	1/4	38	38	48^{**}
CUBE	2/4	7	21	1.23^{*}
HOLLOW	3/6	12	40	0.6^{**}

Table 1: Performance of our algorithms on some CSG models, in comparison to earlier work. Running times are reported in frames/second. An asterisk denotes artifacts in the solution and a double asterisk denotes an incorrect answer.

The table indicates two things: firstly that our algorithms (**BASIC,CONV**) obtain (overall) significantly better frame rates than **SCS**. Moreover, there are far fewer artifacts in our approach: this is possible due to the fewer number of EQUAL tests we perform in the z-buffer.

Figure 2 demonstrates the working of our system on the above models. For each object, the lefthand-most image displays all the primitive objects involved in the CSG operations. As we go from left to right, each image displays the portion of the final answer rendered at that layer. In the case of HOLLOW, the original CSG objects would not be visible in a direct superimposition, and so we render the set of primitive as two distinct figures (the two left-most ones) for ease of viewing. We also emphasize that we place *no* convexity restrictions on our primitives; HELIX contains nonconvex objects.

7 Conclusions

In this paper, we demonstrate that the two-sided depth test, as realized by using the shadow buffer, is a powerful operator in the graphics pipeline. We studied the specific problem of rendering objects represented as CSG trees and provided an algorithm that asymptotically improves the number of rendering passes by a factor of n. It is likely that there are many other problems for which specific aspects of the graphics hardware provides a tremendous advantage. In general, with the increasing power of graphics hardware, theoretical studies that attempt to ascertain the potential and the limits of this pipeline as a general purpose stream engine will be invaluable.

Acknowledgements We thank Nigel Stewart for making his implementation of the algorithm in [Stewart et al. 1998] available on his web page. We also thank nVidia for providing a sample implementation of depth peeling and helping us with the Linux implementation of it.

References

EPSTEIN, D., JANSEN, F., AND ROSSIGNAC, J. 1989. Z-buffer rendering from CSG: The Trickle algorithm. Research Report RC 15182, IBM.

ERHART, G., AND TOBLER, R. 2000. General purpose z-buffer CSG rendering with consumer level hardware. Tech. Rep. VRVis 003, VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH.

EVERITT, C., REGE, A., AND CEBENOYAN, C. 2002. Hardware shadow mapping. In *ACM SIGGRAPH 2002 Tutorial Course #31: Interactive Geometric Computations using graphics hardware*, ACM, F38–F51.

EVERITT, C. 2002. Interactive order-independent transparency. Tech. rep., Nvidia Corporation. http://developer.nvidia.com.

FUCHS, H., AND POULTON, J. 1981. Pixel-planes: a VLSI-oriented design for 3-D raster graphics. *Proc. of the 7th Canadian Man-Computer Communications Conf.*, 343–347.

GOLDFEATHER, J., HULTQUIST, J. P. M., AND FUCHS, H. 1986. Fast constructive-solid geometry display in the pixel-powers graphics system. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, ACM Press, ACM, 107–116.

GOLDFEATHER, J., MOLNAR, S., TURK, G., AND FUCHS, H. 1989. Near realtime CSG rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications 9*, 3 (May), 20–28.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical Z-buffer visibility. *Computer Graphics 27*, Annual Conference Series, 231–238.

HOFF, K., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. 2000. Interactive motion planning using hardware-accelerated computation of generalized Voronoi diagrams. In *Proc. IEEE International Conf. on Robotics and Automation*.

HOFF III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. *Computer Graphics 33*, Annual Conference Series, 277–286.

KRISHNAN, S., MUSTAFA, N., AND VENKATASUBRAMANIAN, S. 2002. Hardware-assisted computation of depth contours. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms*, 558–567.

LARSEN, E. S., AND MCALLISTER, D. 2001. Fast matrix multiples using graphics hardware. In *Supercomputing*.

LINDHOLM, E., KILGARD, M., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proc. ACM SIGGRAPH 2001*.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9*, 4 (July), 43–55.

MUSTAFA, N., KOUTSOFIOS, E., KRISHNAN, S., AND VENKATASUBRAMANIAN, S. 2001. Hardware assisted view dependent map simplification. In *17th ACM Symposium on Computational Geometry*, 50–59.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *Proc. ACM SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., 425–432.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proc. ACM SIGGRAPH 2001*.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 703–712.

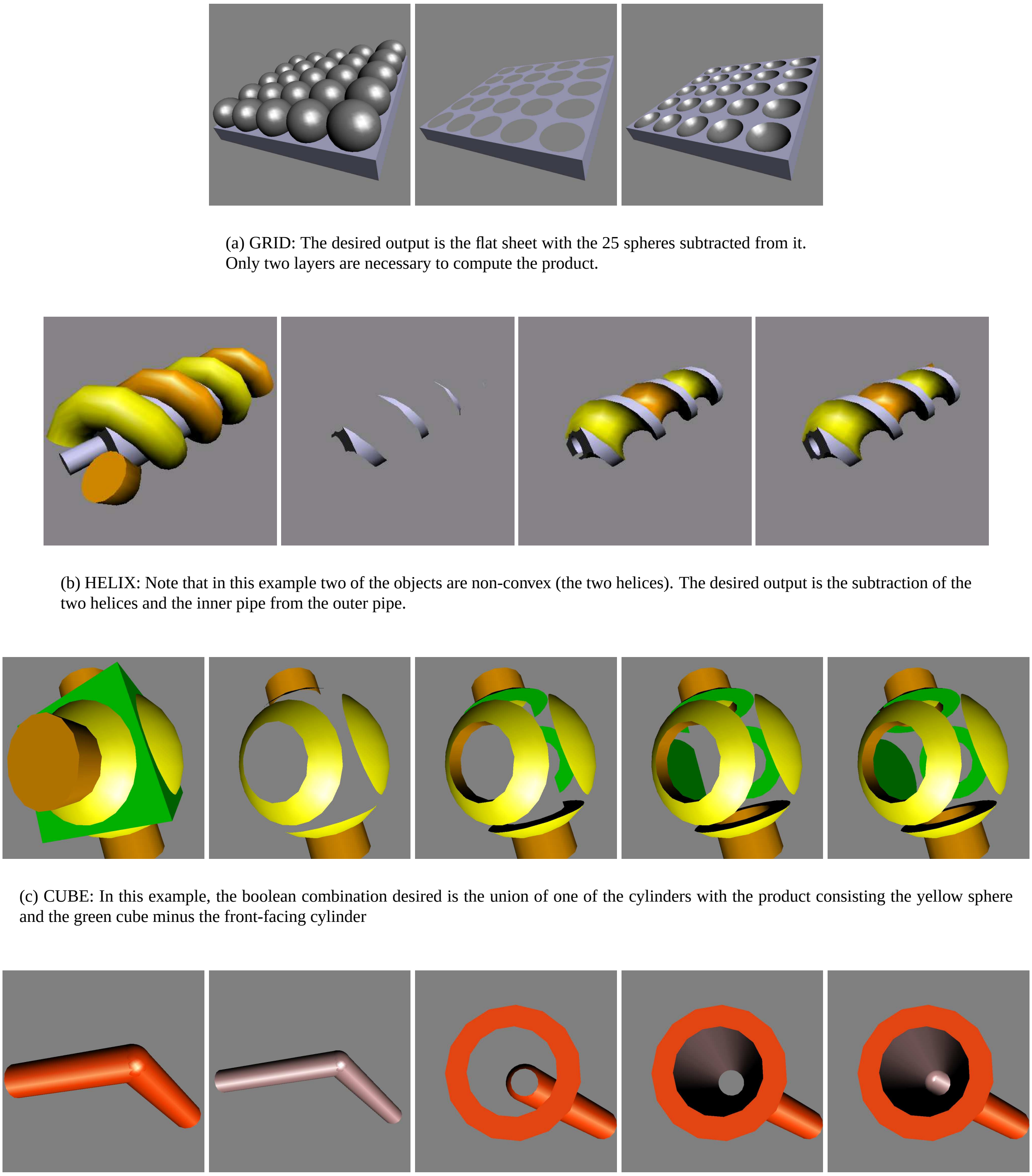
ROSSIGNAC, J., AND WU, J. 1992. Correct shading of regularized CSG solids using a depth-interval buffer. In *Advanced Computer Graphics Hardware V: Rendering, Ray Tracing and Visualization Systems*, R. L. Grimsdale and A. Kaufman, Eds., Eurographics Seminars. Springer-Verlag, 117–138.

STEWART, N., LEACH, G., AND JOHN, S. 1998. An improved z-buffer CSG rendering algorithm. In *Proc. Eurographics/Siggraph Workshop on Graphics Hardware*, 25–30.

STEWART, N., LEACH, G., AND JOHN, S. 2000. A CSG rendering algorithm for convex objects. In *8th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media - WSCG 2000*, vol. II, 369–372.

STEWART, N., LEACH, G., AND JOHN, S. 2002. Linear-time CSG rendering of intersected convex objects. In *10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision - WSCG 2002*, vol. II, 437–444.

WIEGAND, T. F. 1996. Interactive rendering of CSG models. *Computer Graphics Forum 15*, 4, 249–261.



(a) GRID: The desired output is the flat sheet with the 25 spheres subtracted from it. Only two layers are necessary to compute the product.

(b) HELIX: Note that in this example two of the objects are non-convex (the two helices). The desired output is the subtraction of the two helices and the inner pipe from the outer pipe.

(c) CUBE: In this example, the boolean combination desired is the union of one of the cylinders with the product consisting the yellow sphere and the green cube minus the front-facing cylinder

(d) HOLLOW PIPE: In this example, for ease of viewing we show the original objects in the two left-most figures. The output should be a hollow pipe formed by the subtraction of the inner tube (colored in pink) from the outer tube (in red)

Figure 2: Examples of CSG renderings produced by our algorithm. In each example, the left-most figure depicts all the primitives prior to any boolean operations. Each subsequent figure depicts the rendered output after successive steps of the algorithm, and the right-most figure shows the final answer.

Submitted to the European Symposium on Algorithms 2003

Streaming Geometric Optimization using Graphics Hardware [*]

Pankaj K. Agarwal[†] Shankar Krishnan[‡] Nabil H. Mustafa[§] Suresh Venkatasubramanian[¶]

Abstract

The need for analyzing and processing massive data in real time has led to a flurry of activity related to performing computations on a *data stream*. In this paper we propose algorithms for solving a variety of geometric optimization problems on a stream of points in $\mathbb{R}^2$ or $\mathbb{R}^3$. In particular, we study the problems of computing various extent measures (e.g. diameter, width, smallest enclosing disk), collision detection (penetration depth and distance between polytopes), and shape fitting (minimum width annulus, circle/line fitting).

The main contribution of this paper is a *unified* approach to solving all of the above problems efficiently using modern graphics hardware All the above problems can be approximated using a constant number of passes over the data stream. All of our algorithms are easily implemented, and our empirical study demonstrates that the running times of our programs are comparable to the best implementations for the above problems. Another significant property of our results is that although the best known implementations for the above problems are quite different from each other, our algorithms all draw upon the same set of tools, making their implementation significantly easier.

Our graphics-hardware based technique can also be used to solve a number of other geometric optimization problems (problems in layered manufacturing, Hausdorff distance between planar point sets), which do not necessarily arise in the streaming model.

[*] P.A. and N.M. are supported by by NSF grants ITR–333–1050, EIA–9870724, EIA–997287, and CCR–02–04118, and by a grant from the U.S.-Israeli Binational Science Foundation.

[†] Department of Computer Science, Duke University, Durham, NC 27708-0129, U.S.A. Email: pankaj@cs.duke.edu

[‡] AT&T Research, 180 Park Ave, Florham Park, NJ 07932. Email: krishnas@research.att.com

[§] Department of Computer Science, Duke University, Durham, NC 27708-0129, U.S.A Email: nabil@cs.duke.edu

[¶] AT&T Research, 180 Park Ave, Florham Park, NJ 07932. Email: suresh@research.att.com

1 Introduction

Motivated by various applications (e.g., data warehousing, data mining, query optimization), the need for analyzing and processing massive data in real time has led to a flurry of activity related to performing computations on a *data stream*, such as the computation of frequently occurring items in a stream. Several techniques have been developed for computing statistical aggregates and histograms over data streams, which scan the input a few times (e.g. a small constant number of times) and compute the desired information using very little space. In general, these methods achieve efficiency by returning approximate answers to queries.

In this paper we propose algorithms for solving a variety of geometric optimization problems over a stream of two or three dimensional geometric data (e.g. points, lines, polygons). In particular, we study three classes of problems: (a) **Extent Measures:** computing various *extent measures* (e.g. diameter, width, smallest enclosing circle) of a stream of points in $\mathbb{R}^2$ or $\mathbb{R}^3$, (b) **Collision Detection:** computing the penetration depth of a pair of convex polyhedra in three dimensions and **Shape Fitting:** approximating a set of points by simple shapes like circles or annuli.

Many of the problems we study can be formulated as computing and/or overlaying lower and upper envelopes of certain functions. We will be considering approximate solutions, and thus it suffices to compute the value of these envelopes at a set of uniformly sampled points, i.e., on a grid. This allows us to exploit recent developments in graphics hardware accelerators. Almost all modern graphics cards (examples include the nVidia GeForce and ATI Radeon series) provide hardware support for computing the envelope of a stream of bivariate functions at a uniform sample of points in $[-1, +1]^2$ and for performing various arithmetic and logical operations on each of these computed values, which makes them ideal for our applications. We therefore study the above streaming problems in the context of graphics hardware.

Related Work. Data streams were first referred to in the early 1980s and were formalized in the late 1990s through a series of papers [5, 21, 11]. In the standard streaming model, the input $\{x_1, \ldots, x_n\}$ is written in sequence on an *input tape*. The algorithm has a *read head*, and works in passes. In each pass, the read head makes one sequential scan over the input tape, and then returns to the beginning. It is not permitted to move backwards in the course of a scan. The algorithm is allowed a work space of size $o(n)$ on which it can perform any arbitrary computation, and also has an *output tape* on which it writes the result of the computation. The efficiency of an algorithm is measured in terms of the size of the working space, the number of passes, and the time it spends on performing the computation. Typically, algorithms are deemed to be efficient if they work in *one* or a few passes, and use $O(n^\epsilon), \epsilon < 1$ workspace memory. Efficient streaming algorithms for computing the mean and median [31, 30, 15], histograms of time-series data [14], the k-center and k-median [8, 18], and a number of other problems have been proposed. Most of these algorithms provide a tradeoff between efficiency and accuracy. Lower bounds on various problems in the streaming model have also been proposed; see [33, 6] and the references therein. Recently, Korn *et al.* [25] developed a streaming algorithm for the reverse nearest-neighbor problem, in which given a fixed set of red points and a stream of blue points, the algorithm is required to compute, for each red point r, the number of blue points for which r is their nearest neighbor.

Traditionally, the graphics hardware has been used for rendering three-dimensional scenes. But the growing sophistication of graphics cards and their relatively low cost has led researchers to use their power for a variety of other problems, including constructive solid geometry [17], robotics [22], GIS [35], and scientific computing [27, 34]. In the context of geometric computing, it has been successfully used for computing Voronoi diagrams [23], map simplification [32], depth contours [26], and other problems. These algorithms use various frame buffers (e.g. color, depth, stencil, and accumulation buffers), texture memory, and pixel processors in clever ways, exploiting the streaming SIMD architecture of graphics cards. Fournier and Fussel [13] were the first to study general stream computations on graphics cards; a recent paper [16] shows lower bounds on the number of passes required by hardware-based k^{th}-element selection operations, as well as showing the necessity of certain hardware functions in reducing the number of passes in selection from $\Omega(n)$ to $O(\log n)$.

There has been extensive work in computational geometry and computing extent measures and shape fitting [3]. The most relevant work in a recent result by Agarwal *et al.* [2] in which they present an algorithm for computing a small size "core set" C of a given set S of points in $\mathbb{R}^d$ whose extent approximates the extent of S. As a corollary, their algorithm can compute the diameter of S in time $O(n + 1/\varepsilon^{3(d-1)/2})$ and the smallest spherical shell containing S in time $O(n + 1/\varepsilon^{2d} \log 1/\varepsilon)$. Their algorithm can be adapted to the streaming model, in the sense that C can be computed by performing one pass over S, after which one can compute an ε-approximation of the desired extent measure in $1/\varepsilon^{O(1)}$ time using $1/\varepsilon^{O(1)}$ memory. Recently, Feigenbaum *et al.* [12] studied the computation of the diameter of a set of points in the streaming and sliding-window models.

Our Work. In this paper, we demonstrate a large class of geometric optimization problems that can be approximated efficiently using graphics hardware. A unifying theme of the problems that we solve is that they can be expressed in terms of minimizations over envelopes of bivariate functions.

Extent Problems: Table 1 summarizes our results for computing the diameter and width (in two and three dimensions) and the smallest enclosing ball (in two dimensions) of a set of points. All the algorithms are approximate, and compute the desired answer in a constant number of passes. We note here that although the number of passes is more than one, each pass does not use any information from prior passes and the computation effectively runs in a single pass. For reasons that will be made clear in Section 4, the graphics pipeline requires us to perform a series of passes that explore different regions of the search space.

In addition, the smallest bounding box of a planar point set can also be approximated in a constant number of passes; computing the smallest bounding box in three dimensions can be done in $1/\sqrt{\alpha - 1}$ passes, where α is an approximation parameter.

Problem	Approximation	Number of passes
Diameter (2D/3D)	α·OPT	6
1-center (2D)	α·OPT	2
Width (2D/3D)	$\alpha \cdot \text{OPT} + \beta$	6
Bounding Box (2D)	$\alpha \cdot \text{OPT} + \beta$	4
Bounding Box (3D)	$\alpha \cdot \text{OPT} + \beta$	$1/\sqrt{\alpha - 1}$
Hausdorff Distance (2D)	$\alpha \cdot \text{OPT} + \beta$	2

Table 1: Results for Extent Problems. In all cases, the algorithm works for any choice of $\alpha > 1, \beta > 0$

Collision detection: We present a hardware-based algorithm for approximating the *penetration depth* between two convex polytopes. In general, our method can compute any inner product-based distance between any two convex polyhedra (intersecting or not). Our approach can also be used to compute the Minkowski sum of two convex polygons in two dimensions.

Shape fitting and other problems: We also present heuristics for a variety of shape fitting problems in the plane: computing the minimum width annulus, best fit circle, and best-fit line for a set of points, and computing the Hausdorff distance between two sets of points. Our methods are also applicable to many problems in computer-aided design; most notably, we can address with a single approach the multi-criteria optimization problems in *layered manufacturing* [28].

Experimental results: An important practical consequence of our unified approach to solving these problems is that all our implementations make use of the same underlying procedures, and thus a single implementation provides much of the code for all of the problems we consider. We present an empirical study that compares our algorithms to existing implementations for representative problems from the above list; in all cases we are comparable, and in many cases we are far superior to existing software-based implementations.

Paper Outline In Section 2 we introduce the graphics pipeline, the basic primitive operations that it can perform, and some of the tools that we will use in our algorithms. Section 3 discusses Gauss maps, duality, and how we use the two-dimensional grid to represent a Gauss map. Sections 4,5 and 6 present our results on extent measures, collision detection and shape fitting. We present a detailed experimental study in Section 7. Finally, Appendix A describes the pseudocode for our general bivariate envelope computation.

2 Preliminaries

The Graphics Pipeline. The graphics pipeline is primarily used as a rendering (or "drawing") engine to facilitate interactive display of complex three-dimensional geometry. The input to the pipeline is a set of geometric primitives and images to be "drawn" on a two-dimensional grid of pixels known as the *frame buffer*. The frame buffer is a collection of several individual dedicated buffers (color, stencil, depth buffers etc.). The user interacts with the pipeline via a standardized software interface such as OpenGL or DirectX that is designed to mimic the graphics subsystem.

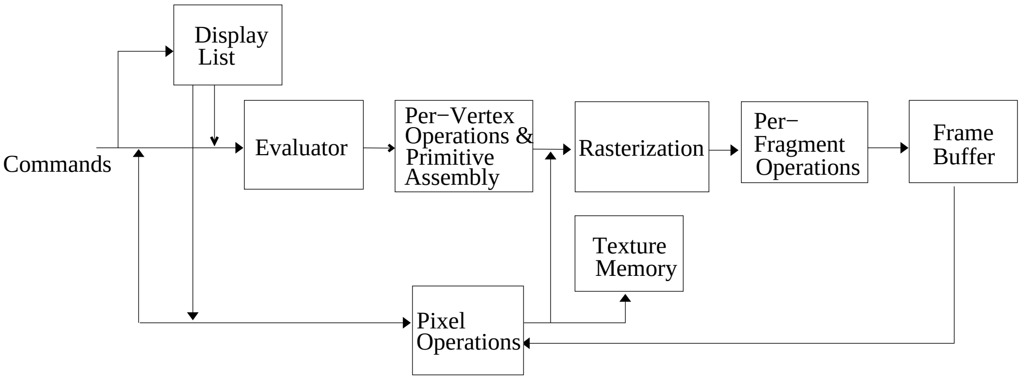


Figure 1: The Graphics Pipeline [36]

We describe some of the key elements of the pipeline here (see Figure 1). For more details, the reader may refer to The OpenGL Programming Guide [36]. Inputs to the pipeline are usually in one of two forms — geometry and images. *Per-vertex operations* take geometric primitives (described by points, line segments, and polygons) as input. The results of this stage are transformed and clipped vertices (with respect to a viewing volume) with related color, depth, and sometimes texture-coordinate values. In the next phase, *rasterization*, geometric data is rendered to produce an array of *fragments* corresponding to a two-dimensional description of the geometry.The pipeline also provides special pixel copy operations to copy data to/from the framebuffer and texture memory.

Next, operations on individual fragments are performed before they finally alter the framebuffer. This is the stage of the pipeline that we exploit for our computations: the operations performed include conditional updates into the framebuffer based on incoming and previously stored depth or stencil values, blending of incoming fragment colors with stored colors, as well as masking and other logical operations on fragment values. Table 2 lists a subset of the per-fragment operations. Consider a pixel P_{ij} at position (i,j) in the framebuffer with color, stencil and depth values denoted by CB_{ij}, SB_{ij} and DB_{ij} respectively. Let us assume that an input fragment arrives at position (i,j) with depth value z and color RGB, and a user-specified constant K. *SB op* and *DB op* are any stencil buffer or depth buffer operations.

Finally, contents (or some computed statistics like histograms or min/max) of the framebuffer can be transferred to the system memory through a single OpenGL call. This is usually termed the *readback* stage. The cost

Color buffer operations:	**Stencil buffer operations:**	**Depth buffer operations:**
$CB_{ij} \leftarrow \{\min, \max\}(CB_{ij}, RGB)$	$SB_{ij} \leftarrow K, SB_{ij} \leftarrow 0$	$DB_{ij} \leftarrow z$
$CB_{ij} \leftarrow CB_{ij} \odot RGB, \odot = \{+, -, \times, \wedge, \vee\}$	$SB_{ij} \leftarrow SB_{ij} + \{1, -1, 0\}$	$DB_{ij} \leftarrow \{\min, \max\}(DB_{ij}, z)$
$CB_{ij} \leftarrow RGB$	$SB_{ij} \leftarrow !SB_{ij}$	$DB_{ij} \leftarrow DB_{ij}$

Table 2: Framebuffer Operations

of a readback is directly proportional to the bandwidth requirement for this data transfer and can be significantly larger (by orders of magnitude) than the cost of sending geometric objects to the hardware. Thus an efficient algorithm attempts to minimize the number of readbacks (which is closely related to the number of passes).

Computing Envelopes. Let $F = \{f_1, \ldots, f_n\}$ be a set of d-variate functions. The *lower envelope* of F is defined as $E^-_F(x) = \min_i f_i(x)$, and the *upper envelope* of F is defined as $E^+_F(x) = \max_i f_i(x)$. The projection of E^-_F (resp. E^+_F) is called the *minimization* (resp. *maximization*) diagram of S. Set $f^-_F(x)$ (resp. $f^+_F(x)$) to be the index of a function of F that appears on its lower (resp. upper) envelope. Finally, define $I_F(x) = E^+_F(x) - E^-_F(x)$. We will omit the subscript F when it is obvious from the context.

If F is a family of piecewise-linear bivariate functions, we can compute E^-, E^+, f^-, f^+ for each pixel $x \in [-1, +1]^2$, using the graphics hardware. We will assume that function $f_i(x)$ can be described accurately as a collection of triangles. Pseudocode for the computations below is given in Appendix A.

Computing E^- (E^+): Each vertex v_{ij} is assigned a color equal to its z-coordinate (depth) (or function value). The graphics hardware generates color values across the face of a triangle by performing bilinear interpolation of the colors at the vertices. Therefore, the color value at each pixel correctly encodes the function value. We disable the stencil test, set the depth test to $\min$ (resp. $\max$). After rendering all the functions, the color values in the framebuffer contains their lower (resp. upper) envelope. In the light of recent developments in the programmability of the graphics hardware, nonlinear functions can be encoded as part of a shading language (or fragment program) to compute their envelopes as well.

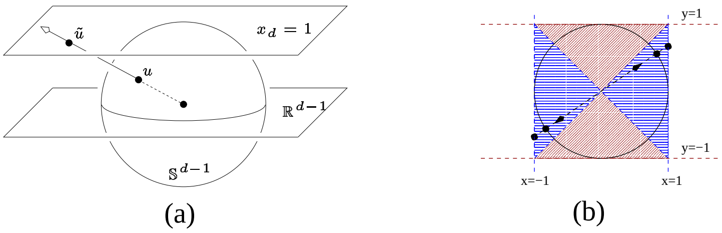
Computing f^- (f^+): Each vertex v_{ij} of function f_i is assigned the color c_i (in most cases, c_i is determined by the problem). By setting the graphics state similar to the previous case, we can compute f^- and f^+.

In many of the problems we address, we will compute envelopes of distance functions. That is, given a distance function $\delta(\cdot, \cdot)$ and a set $S = \{p_1, \ldots, p_n\}$ of points in $\mathbb{R}^2$, we define $F = \{f_i(x) \equiv \delta(x, p_i) \mid 1 \leq i \leq n\}$, and we wish to compute the lower and upper envelopes of F. For the Euclidean metric, the graph of each f_i is a cone whose axis is parallel to the z-axis and whose sides are at an angle of $\pi/4$ to the xy-plane. For the square Euclidean metric, it is a paraboloid symmetric around a vertical line. Such surfaces can be approximated to any desired degree of approximation by triangulations ([23]).

Approximations. For purposes of computation, the two-dimensional plane is divided into pixels. This discretization of the plane makes our algorithms approximate by necessity. Thus, for a given problem, the cost of a solution is a function both of the algorithm and the screen resolution. We define a (β, g)-approximation algorithm to be one that provides a solution of cost at most β times the optimal solution, with a grid cell size of $g = g(I, \beta)$, where I is the instance of the problem. This definition implies that different instances of the same problem may require different grid resolutions.

3 Gauss Maps And Duality

Let $S = \{p_1, \ldots, p_n\}$ be a set of n points in $\mathbb{R}^d$. A direction in $\mathbb{R}^d$ can be represented by a unit vector $u \in \mathbb{S}^{d-1}$. For $u \in \mathbb{S}^{d-1}$, let $\hat{u}$ be its *central* projection, i.e., the intersection point of the ray $\vec{ou}$ with the hyperplane $x_d = 1$ (resp. $x_d = -1$) if u lies in the positive (resp. negative) hemisphere.



(a) (b)

Figure 2: (a) An illustration of central projection (b) Two duals used to capture the Gauss Map

For a direction u, we define the *extremal point* in direction u to be $\lambda(u, S) = \arg\max_{p \in S} \langle \hat{u}, p \rangle$, where $\langle \cdot, \cdot \rangle$ is the inner product. We refer to $\omega(u, s) = \max_{p \in S} \langle \hat{u}, p \rangle - \min_{p \in S} \langle \hat{u}, p \rangle$ as the *directional width* of S. The *Gaussian map* of the convex hull of S is the decomposition of $\mathbb{S}^{d-1}$ into maximal connected regions so that the the extremal point is the same for all directions within one region.

For a point $p = (p_1, \ldots, p_d)$, we define its *dual* to be the hyperplane $p^* : x_d = p_1 x_1 + \cdots + p_{d-1} x_{d-1} + p_d$. Let $H = \{p^* \mid p \in S\}$ be the set of hyperplanes dual to the points in S. The following is easy to prove.

Lemma 3.1. *For $u \in \mathbb{S}^{d-1}$, $\lambda(u, S) = f_H^+(\hat{u}_1, \ldots, \hat{u}_{d-1})$ if u lies in the positive hemisphere, and $\lambda(u, S) = f_H^-(\hat{u}_1, \ldots, \hat{u}_{d-1})$ if u lies in the negative hemisphere; here $\hat{u} = (\hat{u}_1, \ldots, \hat{u}_d)$.*

Hence, we can compute $\lambda(u, S)$ using f_H^+ and f_H^-. Note that the central projection of the portion of the Gaussian map of S in the upper (resp. lower) hemisphere is the maximization (resp. minimization) diagram of H. Thus, for $d = 3$ we can compute portion of the Gaussian map of S whose central projection lies in the square $[-1, +1]^2$, using graphics hardware, as described in Section 2. In other words, we can compute the extremal points of S for all u such that $\hat{u} \in [-1, 1]^2 \times \{1, -1\}$.

If we also take the central projection of a vector $u \in \mathbb{S}^2$ onto the planes $y = 1$ and $x = 1$, then at least one of the central projections of u lies in the square $[-1, +1]^2$ of the corresponding plane. Let R_x (resp. R_y) be the rotation transform that maps the unit vector $(1, 0, 0)$ (resp. $(0, 1, 0)$) to $(0, 0, 1)$. Let H_x (resp. H_y) be the set of planes dual to the point set $R_x(S)$ (resp. $R_y(S)$). If we compute $f_{H_x}^+$, $f_{H_x}^-$, $f_{H_y}^+$, and $f_{H_y}^-$ for all $x \in [-1, +1]^2$, then we can guarantee that we have compute extremal points in all directions (see Fig. 2(b) for an example in two dimensions).

In general, vertices of the arrangement of dual hyperplanes may not lie in a $[-1, +1]^3$ box. A generalization of the above idea can be used to compute a family of three duals such that any vertex of the dual arrangement is guaranteed to lie in the region $[-1, +1]^2 \times [-n, n]$ in *some* dual. Such a family of duals can be used to compute more general functions on arrangements using the graphics hardware; a special case of this result in two dimensions was proved in [26]. In general, the idea of using a family of duals to maintain boundedness of the arrangement can be extended to d dimensions. We defer these more general results to a full version of the paper.

4 Extent Measures

Let $S = \{p_1, \ldots, p_n\}$ be a set of points in $\mathbb{R}^d$. We describe streaming algorithms for computing the diameter and width of S for $d \leq 3$ and the smallest enclosing box and ball of S for $d = 2$.

Diameter. In this section we describe a six-pass algorithm for computing the diameter of a set S (the maximum distance between any two points of S) of n points in $\mathbb{R}^3$. It is well known that the diameter of S is realized by a pair of antipodal points, i.e., there exists a direction u in the positive hemisphere of $\mathbb{S}^2$ such that $\mathrm{diam}(S) = \|\lambda(u, S) - \lambda(-u, S)\| = \|f_H^+(\hat{u}_1, \hat{u}_2) - f_H^-(\hat{u}_1, \hat{u}_2)\|$, where H is the set of planes dual to the points in S. In order to compute $\|f_H^+(x) - f_H^-(x)\|$, we assign the RGB values of the color of a plane p_i^* to be the coordinates of p_i. The first pass computes f_H^+, so after the pass, the pixel x in the color buffer contains the coordinates of $f_H^+(x)$. We copy this buffer to the texture memory and compute f_H^- in the second pass. We then compute $\|f_H^+(x) - f_H^-(x)\|$ for each pixel. Since the hardware computes these values for $x \in [-1, +1]^2$, we repeat these steps for $R_x(S)$ and $R_y(S)$ as well. Detailed pseudo-code for the diameter computation is presented in Appendix A. Since our algorithm operates in the dual plane, the discretization incurred is in terms of the directions, yielding the following result.

Theorem 4.1. *Given a point set $S \subset \mathbb{R}^3, \alpha > 1$, there is a six-pass $(\alpha, g(\alpha))$-approximation algorithm for computing the diameter of S.*

Width. Let S be a set of n points in $\mathbb{R}^3$. The *width* of S is the minimum distance between two parallel planes that enclose P between them, i.e., $\mathrm{width}(S) = \min_{u \in \mathbb{S}^2} \omega(u, S)$. The proof of the following lemma is relatively straightforward.

Lemma 4.1. *Let R_x, R_y be the rotation transforms as described earlier, and let H (resp. H_x, H_y) be the set of planes dual to the points in S (resp. $R_x(S)$, $R_y(S)$). Then*

$$\mathrm{width}(S) = \min_{p \in [-1,+1]^2} \frac{1}{\|(p,1)\|} \min\{I_H(p), I_{H_x}(p), I_{H_y}(p)\}.$$

This lemma implies that the algorithm for width can be implemented similar to the algorithm for diameter. Consider a set of coplanar points in $\mathbb{R}^3$. No discretized set of directions can yield a good approximation to the width of this set (which is zero). Hence, we can only prove a slightly weaker approximation result, based on knowing a lower bound on the optimal width. We omit the details from this version and conclude the following.

Theorem 4.2. *Given a point set $S \subset \mathbb{R}^3$, $\alpha > 1$, and $\tilde{w} \leq w^*$, there is a six-pass $(\alpha, g(\alpha, \tilde{w}))$-approximation algorithm for computing the width of S.*

1-center The 1-center of a point set S in $\mathbb{R}^2$ is a point $c \in \mathbb{R}^2$ minimizing $\max_{p \in P} d(c, p)$. This is an envelope computation, but in the primal plane. For each point $p \in S$, we render the colored distance cone as described in Section 2. The 1-center is then the point in the upper envelope of the distance cones with the smallest distance value. The center of the smallest enclosing ball will always lie inside conv(S). The radius of the smallest enclosing ball is at least half the diameter of S. Thus, if we compute the farthest point Voronoi diagram on a grid of cell size $g = \alpha\Delta/2$, the value we obtain is a $(1 + \alpha)$-approximation to the radius of the smallest enclosing ball. An approximate diameter computation gives us $\tilde{\Delta} \leq 2\Delta$, and thus a grid size of $\alpha\tilde{\Delta}/4$ will obtain the desired result.

Theorem 4.3. *Given a point set S in $\mathbb{R}^2$ and a parameter $\alpha > 1$, there is a two-pass $(\alpha, g(\alpha))$-approximation algorithm for computing the smallest-area disk enclosing S.*

Smallest Bounding Box. Let S be a set of points in $\mathbb{R}^2$. A rectangle enclosing S consists of two pairs of parallel lines, each of which are orthogonal to the other. For a direction $u \in \mathbb{S}^1$, let $u^\perp$ be the direction normal to u. Then the side lengths of the smallest rectangle whose edges are in directions u and $u^\perp$ that contains S are $W(u) = \omega(u, S)$ and $H(u) = \omega(u^\perp, S)$. Hence, the area of the smallest rectangle containing S is $\min_{u \in \mathbb{S}^1} W(u) \cdot H(u)$. The algorithm to compute the minimum-area two-dimensional bounding box can now be viewed as computing the minimum widths in two orthogonal directions and taking their product. Similarly, we can compute a minimum-perimeter rectangle containing S. Since the algorithm is very similar to computing the width, we omit all the details and conclude the following.

Theorem 4.4. *Given a point set S in $\mathbb{R}^2$, $\alpha > 1$, and a lower bound $\tilde{a}$ on the area of the smallest bounding box, there is a four-pass $(\alpha, g(\alpha, a))$-approximation algorithm for computing the smallest enclosing bounding box.*

It is not clear how to extend this algorithm to $\mathbb{R}^3$ using a constant number of passes since the set of directions normal to a given direction is $\mathbb{S}^1$. However, by sampling the possible choices of orthogonal directions, we can get a $(1+\alpha)$-approximation in $1/\sqrt{\alpha - 1}$ passes. Omitting all the details, we obtain the following.

Theorem 4.5. *Given point set $S \subset \mathbb{R}^3$, $\alpha > 1$ and lower bound $\tilde{a}$ on the area of the smallest bounding box, there is an $O(1/\sqrt{\alpha - 1})$-pass $(\alpha, g(\alpha, a))$-approximation algorithm for computing the smallest bounding box.*

5 Collision Detection

Given two convex polytopes P and Q in $\mathbb{R}^3$, their penetration depth, denoted $PD(P, Q)$ is defined as the translation vector t such that P and $Q + t$ are disjoint. We can specify a placement of Q by fixing a reference point $q \in Q$ and specifying its coordinates. Assume that initially q is at the origin o. Since $M = P \oplus -Q$ is the set of placements of Q at which Q intersects P, $PD(P, Q) = \min_{z \in \partial M} d(o, z)$ For a direction $u \in \mathbb{S}^2$, let $h_M(u)$ be the tangent plane of M normal to direction u. As shown in [1], $PD(P, Q) = \min_{u \in \mathbb{S}^2} d(o, h_M(u))$

Let A be a convex polytope in $\mathbb{R}^3$ and let V be the set of vertices in A. For a direction $u \in \mathbb{S}^2$, let $g_A(u) = \max_{p \in V} \langle p, \hat{u} \rangle$. It can be verified that the tangent plane of A in direction u is $h_A(u) : \langle \hat{u}, x \rangle = g_A(u)$. Therefore $PD(P, Q) = \min_{u \in \mathbb{S}^2} \frac{g_M(u)}{\|\hat{u}\|}$. The following lemma shows how to compute $h_M(u)$ from $h_P(u)$ and $h_{-Q}(u)$.

Lemma 5.1. *For any $u \in \mathbb{S}^2$,*

$$g_M(u) = g_P(u) + g_{-Q}(u)$$

This lemma follows from the fact that for convex P and Q, the point of M extreme in direction u is the sum of the points of P and Q extreme in direction u. Therefore,

$$PD(P, Q) = \min_{u \in \mathbb{S}^2} \frac{g_P(u) + g_{-Q}(u)}{\|u\|}$$

Hence, we discretize the set of directions in $\mathbb{S}^2$, compute $g_P(u), g_{-Q}(u)$, $(g_P(u)+g_{-Q}(u))/\|\hat{u}\|$ and compute their minimum. Since g_P and g_{-Q} are upper envelopes of a set of linear functions, they can be computed at a set of directions by the graphics hardware in six passes, as described in Section 4. Pseudocode for this computation is described in Appendix A

We note here that the above approach can be generalized to compute any inner product-based distance between two non-intersecting convex polytopes in three dimensions. It can also be used to compute the Minkowski sum of polygons in two dimensions.

6 Shape Fitting

We now present hardware-based heuristics for shape analysis problems. These problems are solved in the primal, by computing envelopes of distance functions.

Circle fitting. The *minimum width annulus* of a point set $P \subset \mathbb{R}^2$ is a pair of concentric disks R_1, R_2 of radii $r_1 > r_2$ such that P lies in the region $R_1 \setminus R_2$ and $r_1 - r_2$ is minimized. It is well-known [10] that the center of the minimum-width annulus lies on a vertex of the overlay of the nearest and farthest-neighbor Voronoi diagrams.

Note that the center of the minimum width annulus could be arbitrarily far away from the point set (for example, the degenerate case of points on a line). Furthermore, when the minimum width annulus is *thin*, the pixelization induces large errors which cannot be bounded. Therefore, we look at the special case when the annulus is not thin, i.e. $r_1 \geq (1+\varepsilon)r_2$. For this case, Chan [7] presents a $(1+\varepsilon)$ approximation algorithm as follows: lay a uniform grid of resolution $\varepsilon \cdot w$ on the pointset, where w is some constant factor approximation to the minimum width, snap each point to the nearest grid point and remove duplicates. Chan shows that the grid dimensions are at most $1/\varepsilon^2 \times 1/\varepsilon^2$ and the center realizing the approximation is one of the grid points. This algorithm can be implemented efficiently in hardware as follows: set the buffer B to be of size $1/\varepsilon^2 \times 1/\varepsilon^2$; for each point p_i, draw its Euclidean distance cone C_i as described in Section 2. Let $C = \{C_1, C_2, \ldots, C_n\}$ be the collection of distance functions. Then the minimum width annulus can be computed as $\min_{x \in B} I_C(x)$ with center $\arg\min_{x \in B} I_C(x)$. This approach yields a fast streaming $(1+\varepsilon)$-approximation algorithm for the minimum-width annulus (and for the minimum-area annulus as well, by using paraboloids instead of cones).

The *best-fit circle* of a set of points $P = \{p_1, p_2, \ldots, p_n\} \subset \mathbb{R}^2$ is a circle $C(c, r)$ of radius r centered at c such that the expression $\sum_{p \in P} d^2(p, C)$ is minimized. For a fixed center c, elementary calculus arguments show that the optimal r is given by $r^* = 1/n \sum_{p \in P} d(p, c)$. Let $d_i = \|p_i - c\|$. The cost of the best fit circle of radius r^* centered at c can be shown to be $\sum_{i \leq n} d_i^2 - (1/n)(\sum_{i \leq n} d_i)^2$.

Once again, this function can be represented as an overlay of distance cones, and thus for each grid point, the cost of the optimal circle centered at this grid point can be computed. Unfortunately, this fails to yield an approximation guarantee for the same reasons as above.

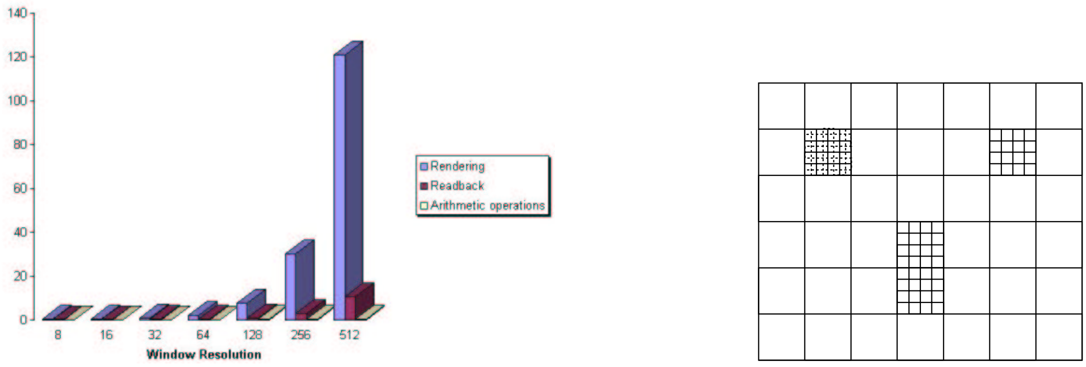
Hausdorff distance. Given two point sets $P, Q \subset \mathbb{R}^2$, the *Hausdorff distance* d_H from P to Q is $\max_{p \in P} \min_{q \in Q} d(p, q)$. Once again, we draw distance cones for each point in Q, and compute the lower envelope of this arrangement of surfaces *restricted to points in* P. Now each grid point corresponding to a point of P has a value equal to the distance to the closest point in Q. A maximization over this set yields the desired result. For this problem, it is easy to see that as for the width, given any lower bound on the Hausdorff distance we can compute a $(\beta, g(\beta))$-approximation to the Hausdorff distance.

7 Experiments

In this section we describe some implementation specific details, and report empirical results of our algorithms, and compare their performance with software-based approximation algorithms.

Cost bottleneck. The costs of operations can be divided into two types: geometric operations, and fragment operations. Most current graphics cards have a number of *geometry engines* and *raster managers* to handle multiple vertex and fragment operations in parallel. Therefore, we can typically assume that the geometry transformation and each buffer operation takes constant time. This assumption breaks down when the input causes certain parts of the graphics pipeline to act as bottlenecks. Most graphics cards describe their performance in terms of the number of triangles processed per second (since there is a fixed cost associated with transforming input objects) and in their processing rate of fragments (called the *fill rate*) in the imaging pipeline [4]. Typical numbers for current cards range between 50-100 millions of triangles per second and a fill rate of 0.5-2 *billion* fragments per second, and these numbers are constantly increasing. For the sort of geometric problems that we address, the fill rate is the main bottleneck.

Figure 3(a) shows the running times of the different components of our algorithm for width for various grid sizes. As the plot shows, the rendering stage bottleneck is roughly unchanged till we saturate the fill-rate, at which point performance degrades severely. We now propose a hierarchical method that circumvents the fill limitation by doing refined local searches for the solution.



(a)

(b)

Figure 3: (a) Breakdown of the width algorithm under varying window sizes (b)Tree of height three produced by the hierarchical computation

Hierarchical Refinement. One way to reduce the fill-rate bottleneck is to produce fewer fragments per plane. Instead of sampling the search space with a uniform grid, we instead perform adaptive sampling by constructing a coarse grid, computing the solution value for each grid point and then recursively refining candidate points. See Figure 3(b) for a tree of height three. The advantage of using adaptive refinement is that not all the grid cells need to be refined to a high resolution. However, the local search performed by this selective refinement could fail to find an approximate solution with the guarantee implied by this higher resolution. In our experiments, we will compare the results obtained from this approach with those obtained by software-based methods.

Empirical Results. In this section we report on the performance of our algorithms. All our algorithms were implemented in C++ and OpenGL, and run on a 2.4GHz Pentium IV Linux PC with an ATI Radeon 9700 graphics card and 512 MB Memory. Our experiments were run on three types of inputs: (i) randomly generated convex shapes [19] (ii) large geometric models of various objects, available at `http://www.cc.gatech.edu/graphmodels/` and (iii) randomly generated input using `rbox` (a component of `qhull`). In all our algorithms below, we use hierarchical refinement (with depth two) to achieve more accurate solutions.

Penetration Depth. We compare our implementation of penetration depth (called `HwPD`) with our own implementation of an exact algorithm (called `SwPD`) based on Minkowski sums which exhibits quadratic complexity and with DEEP [24], which to the best of our knowledge is the only other implementation for penetration depth. DEEP is an incremental approach based on local search, specially suited to maintain the penetration depth when the objects are in motion. The software implementation generates the quadratic vertices from the original polytopes, computes its convex hull and then determines the closest distance from the origin. We used the convex polytopes available at [19], as well as random polytopes found by computing the convex hull of points on random ellipsoids as inputs to test our code. The performance of the algorithms on the input set is presented in Table 3: column two and three gives the sizes of the two input polytopes. `HwPD` always outperforms `SwPD` in running time, in some cases by over three orders of magnitude. With regard to DEEP, the situation is less clear. DEEP performs significant preprocessing on its input, so a single number is not representative of the running times for either program. Hence, we report both preprocessing times and query times (for our code, preprocessing time is merely reading the input). We note that DEEP crashed on some of our inputs; we mark those entries with an asterisk. it is instructive to see that the penetration depth values we get are close to the right answer in all the cases we were able to compare.

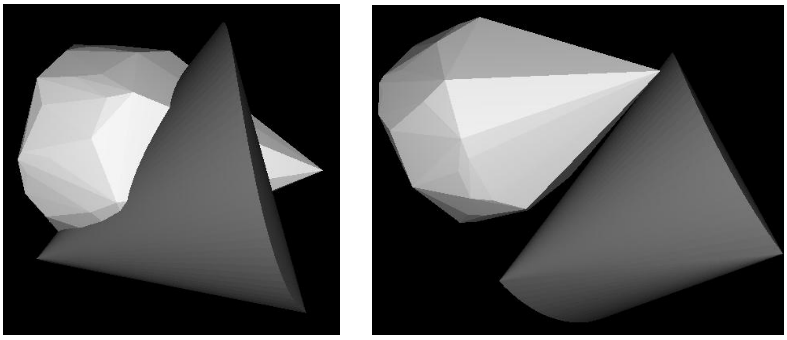
Polygon		HwPD			DEEP			SwPD	
Size	Size	Preproc.	Query Time	Pen. Depth	Preproc.	Query Time	Pen. Depth	Time	Pen. Depth
500	500	**0**	**0.04**	1.278747	**0.15**	**0**	1.29432	**27.69**	1.289027
750	750	**0**	**0.08**	1.053032	**0.25**	**0**	1.07359	**117.13**	1.071013
789	1001	**0.01**	**0.067**	1.349714	*	*	*	**148.87**	1.364840
789	5001	**0.01**	**0.17**	1.360394	*	*	*	-	-
5001	4000	**0.02**	**0.30**	1.362190	*	*	*	-	-
10000	5000	**0.04**	**0.55**	1.359534	**3.28**	**0**	1.4443	-	-

Table 3: Comparison of running times for penetration depth. On the last three datasets, we stopped SwPD after it ran for over 25 minutes. Asterisks mark inputs where DEEP crashed.

Error:		$\epsilon^2 = 0.002$			
		HAnnWidth		SAnnWidth	
Dataset	size	Time	Width	Time	Width
R-Circle-0.1	(1,000)	**0.36**	0.099882	**0.53**	0.099789
R-Circle-0.2	(1,000)	**0.35**	0.199764	**0.42**	0.199442
R-Circle-0.1	(2,000)	**0.66**	0.099882	**0.63**	0.099816
R-Circle-0.1	(5,000)	**1.58**	0.099882	**26.44**	0.099999
R-Circle-0.1	(10,000)	**3.12**	0.099882	**0.93**	0.099999

Table 4: Comparison of running time and approximation quality for 2D-Min Width Annulus

As an example, consider the two polytope shown in Figure 4(a). After running our algorithm, the penetration depth obtained was 1.118479 and the vector achieving this was $(-0.038276, -0.833938, -0.550529)$. Figure 4(b) shows the results of translating the first polytope accordingly (the viewpoint is rotated slightly). ***2D***



(a) The original polytopes

(b) After translation

Figure 4: An illustration of the working of our algorithm for penetration depth

Minimum Width Annulus. We compare our implementation of minimum width annulus (called HAnnWidth) with the software implementation of the grid based method, called SAnnWidth: the software implementation lays a grid of $1/\varepsilon^2 \times 1/\varepsilon^2$, snaps the points to the grid (removing redundant points), and finds the nearest and furthest neighbour of each grid point. The input point sets to the programs were synthetically generated using rbox: R-Circle-r refers to a set of points with minimum width annulus r and is generated by sampling points from a circle and introducing small perturbations. See Table 4 for the timings results.

3D Width. We compare our implementation of width (called HWidth) with the code of Duncan *et al.* [9]

Dataset	size	Error ϵ	HWidth Time	HWidth Width	DGRWidth Time	DGRWidth Width
Club	(16,864)	0.250	**0.45**	0.300694	**0.77**	0.312883
Bunny	(35,947)	0.060	**0.95**	1.276196	**2.70**	1.29231
Phone	(83,034)	0.125	**2.55**	0.686938	**6.17**	0.697306
Human	(254,721)	0.180	**6.53**	0.375069	**18.91**	0.374423
Hand	(327,323)	0.090	**8.66**	0.479850	**21.64**	0.499391
Dragon	(437,645)	0.075	**10.88**	0.813487	**39.34**	0.803875
Buddha	(543,652)	0.075	**13.77**	0.794050	**50.32**	0.809624
Blade	(882,954)	0.090	**23.45**	0.715578	**66.71**	0.726137

Table 5: Comparison of running time and approximation quality for 3D-width

		Error: $\epsilon = 0.015$					
		HDiam		MBDiam		PDiam	
Dataset	size	Time	Diam	Time	Diam	Time	Diam
Club	(16,864)	**0.023**	2.326992	**0.0**	2.32462	**0.00**	2.32462
Bunny	(35,947)	**0.045**	2.549351	**0.75**	2.54772	**0.03**	2.54772
Phone	(83,034)	**0.11**	2.416497	**0.01**	2.4115	**0.07**	2.4115
Human	(254,721)	**0.32**	2.020594	**3.5**	2.01984	**0.04**	2.01938
Hand	(327,323)	**0.41**	2.120115	**0.06**	2.11791	**0.09**	2.11499
Dragon	(437,645)	**0.55**	2.063075	**17.27**	2.05843	**0.21**	2.05715
Buddha	(543,652)	**0.68**	2.113198	**7.75**	2.10768	**0.14**	2.09697
Blade	(882,954)	**1.10**	2.246725	**0.1**	2.23939	**0.22**	2.22407

Table 6: Comparison of running time and approximation quality for 3D-diameter

(DGRWidth). Algorithm DGRWidth reduces the computation of the width to $O(1/\epsilon)$ linear programs. It then tries certain pruning heuristics to reduce the number of linear programs solved in practice. The performance of both the algorithms on a set of real graphical models is presented in Table 5: column four gives the $(1+\epsilon)$-approximate value of the width computed by the two algorithms for the ϵ given in the second column (this ϵ value dictates the window size required by our algorithm, as explained previously, and the number of linear programs solved by DGRWidth). HWidth always outperforms DGRWidth in running time, in some cases by more than a factor of five.

3D Diameter. We compare our implementation (HDiam) with the approximation algorithm of Malandain and Boissonnat [29] (MBDiam), and Har-Peled [20] (PDiam). PDiam maintains a hierarchical decomposition of the point set, and iteratively throws away pairs that are not candidate for the diameter until an approximate distance is achieved by a pair of points. MBDiam is a further improvement on PDiam. Table 6 reports the timing and approximation comparisons for two error measures for graphical models. Although our running times in this case are worse than the software implementations, they are comparable even for very large input, illustrating the generality of our approach.

References

[1] Agarwal, P. K., Guibas, L. J., Har-Peled, S., Rabinovitch, A., and Sharir, M. Computing the penetration depth of two convex polytopes in 3d. In *Scandinavian Workshop on Algorithm Theory* (2000), pp. 328–338.

[2] Agarwal, P. K., Har-Peled, S., and Varadarajan, K. Approximating extent measures of points. Unpublished manuscript, 2002.

[3] AGARWAL, P. K., AND SHARIR, M. Efficient algorithms for geometric optimization. *ACM Comput. Surv. 30* (1998), 412–458.

[4] AKELEY, K., AND JERMOLUK, T. High-performance polygon rendering. *Computer Graphics (Siggraph '88 Proceedings) 22*, 4 (1988), 239–246.

[5] ALON, N., MATIAS, Y., AND SZEGEDY, M. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (1996), pp. 20–29.

[6] BAR-YOSSEF, Z., JAYRAM, T. S., KUMAR, R., AND SIVAKUMAR, D. Information theory methods in communication complexity. In *Proc. IEEE Complexity* (2002).

[7] CHAN, T. M. Approximating the diameter, width, smallest enclosing cylinder, and minimum-width annulus. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.* (2000), pp. 300–309.

[8] CHARIKAR, M., CHEKURI, C., FEDER, T., AND MOTWANI, R. Incremental clustering and dynamic information retrieval. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (1997), ACM Press, pp. 626–635.

[9] DUNCAN, C., GOODRICH, M., AND RAMOS, E. Efficient approximation and optimization algorithms for computational metrology. In *ACM-SIAM Symposium on Discrete Algorithms* (1997).

[10] EBARA, H., FUKUYAMA, N., NAKANO, H., AND NAKANISHI, Y. Roundness algorithms using the Voronoi diagrams. In *Proc. 1st Canad. Conf. Comput. Geom.* (1989).

[11] FEIGENBAUM, J., KANNAN, S., STRAUSS, M., AND VISWANATHAN, M. An approximate l1-difference algorithm for massive data streams. In *Proc. 40th IEEE Symp. Foundations of Computer Science* (1999).

[12] FEIGENBAUM, J., KANNAN, S., AND ZHANG, J. Computing diameter in the streaming and sliding-window models. DIMACS Working Group on Streaming Data Analysis II, 2003.

[13] FOURNIER, A., AND FUSSELL, D. On the power of the frame buffer. *ACM Transactions on Graphics* (1988), 103–128.

[14] GILBERT, A. C., GUHA, S., INDYK, P., KOTIDIS, Y., MUTHUKRISHNAN, S., AND STRAUSS, M. J. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing* (2002), ACM Press, pp. 389–398.

[15] GREENWALD, M., AND KHANNA, S. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data* (2001), ACM Press, pp. 58–66.

[16] GUHA, S., KRISHNAN, S., MUNAGALA, K., AND VENKATASUBRAMANIAN, S. The power of a two-sided depth test and its application to csg rendering and depth extraction. Tech. rep., AT&T, 2002.

[17] GUHA, S., KRISHNAN, S., MUNAGALA, K., AND VENKATASUBRAMANIAN, S. Application of the two-sided depth test to csg rendering. In *2003 ACM SIGGRAPH Symposium on Interactive 3D Graphics* (April 2003).

[18] GUHA, S., MISHRA, N., MOTWANI, R., AND O'CALLAGHAN, L. Clustering data streams. In *Proc. 41st IEEE Symp. Foundations of Computer Science* (Nov. 2000).

[19] HAR-PELED, S. `http://valis.cs.uiuc.edu/~sariel/research/papers/99/nav/nav.html`.

[20] HAR-PELED, S. A practical approach for computing the diameter of a point-set. In *ACM Symposium on Computational Geometry* (2001), pp. 177–186.

[21] HEZINGER, M., RAGHAVAN, P., AND RAJAGOPALAN, S. Computing on data streams. Tech. Rep. 11, DEC, 1998.

[22] HOFF, K., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. Interactive motion planning using hardware-accelerated computation of generalized voronoi diagrams. In *Proc. IEEE International Conf. on Robotics and Automation* (2000).

[23] HOFF III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. Fast computation of generalized Voronoi diagrams using graphics hardware. *Computer Graphics 33*, Annual Conference Series (1999), 277–286.

[24] KIM, Y. J., LIN, M. C., AND MANOCHA, D. Fast penetration depth estimation between polyhedral models using hierarchical refinement. In *International Workshop on Algorithmic Foundations of Robotics (to appear)* (2002).

[25] KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. Reverse nearest neighbour aggregates over data streams. In *Proc. 28th Conference on Very Large Databases (VLDB)* (2002).

[26] KRISHNAN, S., MUSTAFA, N., AND VENKATASUBRAMANIAN, S. Hardware-assisted computation of depth contours. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms* (January 2002), pp. 558–567.

[27] LARSEN, E. S., AND MCALLISTER, D. Fast matrix multiples using graphics hardware. In *Supercomputing* (2001).

[28] MAJHI, J., JANARDAN, R., SMID, M., AND SCHWERDT, J. Multi-criteria geometric optimization problems in layered manufacturing. In *Proceedings of the fourteenth annual symposium on Computational geometry* (1998), ACM Press, pp. 19–28.

[29] MALANDAIN, G., AND BOISSONNAT, J.-D. Computing the diameter of a point set. In *Discrete Geometry for Computer Imagery (DGCI 2002)* (Bordeaux, France, 2002), A. Braquelaire, J.-O. Lachaud, and A. Vialard, Eds., vol. 2301 of *LNCS*, Springer. also INRIA research report RR-4233.

[30] MANKU, G. S., RAJAGOPALAN, S., AND LINDSAY, B. G. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA* (1999), A. Delis, C. Faloutsos, and S. Ghandeharizadeh, Eds., ACM Press, pp. 251–262.

[31] MUNRO, J. I., AND PATERSON., M. S. Selection and sorting with limited storage. *Theor. Comp. Sci. 12* (1980.), 315–323.

[32] MUSTAFA, N., KOUTSOFIOS, E., KRISHNAN, S., AND VENKATASUBRAMANIAN, S. Hardware assisted view dependent map simplification. In *17th ACM Symposium on Computational Geometry* (2001), pp. 50–59.

[33] SAKS, M., AND SUN, X. Space lower bounds for distance approximation in the data stream model. In *Proc. 34th ACM Symp. Theory of Computing* (2002).

[34] STRZODKA, R., AND RUMPF., M. Using graphics cards for quantized fem computations. In *Proc. IASTED Intnl. Conf. Visualization, Imaging and Image processing (VIIP)* (2001).

[35] SUN, C., AGRAWAL, D., AND ABBADI, A. E. Hardware acceleration for spatial selection and join. Tech. Rep. 17, U. California, Santa Barbara, 2002.

[36] WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, 3 ed. Addison-Wesley, 1999.

A Pseudo-code

Algorithm A.1 Pseudocode for the lower (upper) envelope computation

```
for $i = 1$ to $n$ do {/* For each piecewise-linear function $f_i$ */}
  for $j = 1$ to $m_i$ do {/* For each vertex in function $f_i$ */}
    /* Let vertex $v_{ij} = (x_{ij}, y_{ij}, z_{ij})$ */
    Set color value of $v_{ij}$ appropriately depending on problem
  end for
end for
/* Compute lower (upper) envelope in color buffer */
Set depth test to $\min$ ($\max$)
Render all functions in $F$
Readback the color buffer
/* Color value at each pixel determines envelope */
```

Algorithm A.2 Pseudocode for penetration depth computation

```
/* Given two convex polytopes $P$ and $Q$ */
for $i = 1$ to $3$ do {/* For each dual */}
  Compute dual planes $\mathbb{D}^p$ and $\mathbb{D}^q$ for vertices of $P$ and $Q$
  Set color of each vertex based on its $z$-coordinate
  Compute lower envelope of $\mathbb{D}^p$ in color buffer
  Copy color buffer to texture memory
  Compute lower envelope of $\mathbb{D}^q$ in color buffer
  Enable blending and set blending function to ADD
  Use texture mapping to access contents of texture
  Compute component-wise sum with the color buffer
  Readback the color buffer
  /* Repeat this for upper envelopes */
  for each pixel in color buffer do
    Compute distance from origin based on color value
    Track minimum
  end for
end for
/* Minimum value and pixel location determine penetration depth and direction */
```

Short Note on Algorithm for Penetration Depth using Graphics Hardware

Shankar Krishnan*

Abstract

This note discussed the use of the graphics hardware to perform proximity queries and penetration depth computations on convex polytopes. We use the idea of duality mapping to find this information without explicitly constructing the Minkowski sum.

1 Introduction

The ability to track closest feature pairs and collisions in an environment with moving objects has been an important and well studied problem in computer graphics and computational geometry. They find applications games and simulation-based design. In order to resolve collisions, a useful way to quantify the contact is called *penetration depth.* The penetration depth of a pair of intersecting objects is the shortest vector over which one object needs to be translated in order that the pair become disjoint. Over the last two decades, a number of algorithms and software systems have been developed to perform collision detection and computation of penetration depth. A complete survey of all the previous work in this area is beyond the scope of this note. We refer the reader to [3, 4, 2] for surveys and recent work on these topics.

In this draft, we will explain an algorithm that takes a different approach to compute the closest distance or the penetration depth between convex polytopes. All the computation required is done purely on the graphics hardware. Another feature of our algorithm is that, unlike other methods for penetration depth computation, we do not explicitly compute the Minkowski sum of the polytopes which is the main computational bottleneck.

We will briefly review some definitions of point-hyperplane duality and Minkowski sums in the next section. We then provide our algorithm for computing the penetration depth and justify its correctness.

2 Definitions

Duality The principle of duality has been known in computational geometry for a long time. The main idea is that points in one space can be mapped into hypeplanes in another space keeping certain geometric atributes invariant. Consider the case in two dimensions. A point $p = (p_x, p_y)$ in the plane can be mapped to the line $p^* : y = -p_x x + p_y$ in the dual plane. In what follows, we use the superscript x^* to denote objects in the dual plane. Similarly, a line in the plane is mapped to a point whose coordinates depend on the coefficients of the line. Some of the main properties of duality are:

- it preserves incidences *i.e.,* if a point p lies on (above) a line l, then the point l^* lies on (below) the line p^* in the dual plane.
- Given a set of point $P = \{p_1, p_2, \ldots, p_n\}$, the set of lines $P^* = \{p_1^*, p_2^*, \ldots, p_n^*\}$ in the dual plane forms an arrangement. The convex hull of P can be obtained from the upper and lower envelope of the arrangement P^*. Figure 1 shows a set of points in the primal plane and the corresponding dual arrangement.

[*] AT&T Research. Email: krishnas@research.att.com

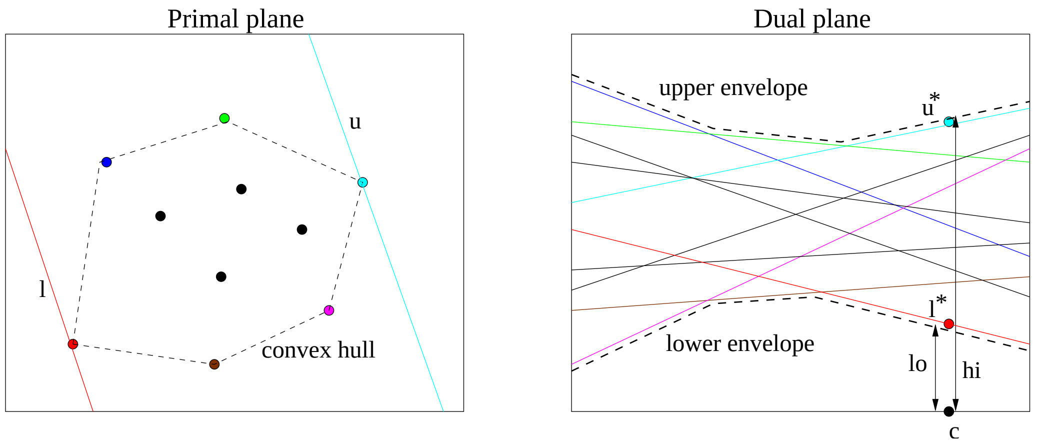


Figure 1: Illustration of point-line duality

- Consider a point $l^* = (\mathbf{c}, lo)$ in the lower envelope and the corresponding point $u^* = (\mathbf{c}, hi)$ in the upper envelope of the dual arrangement P^*. In $\mathbb{R}^d$, $\mathbf{c}$ is a $(d-1)$-dimensional point, and l and u are the heights of the lower and upper envelope (think of the envelopes as terrains). Then the corresponding lines l and u in the original plane are parallel to each other and they are tangential to the convex hull of P (as shown in the figure).

The above facts are true in any dimension.

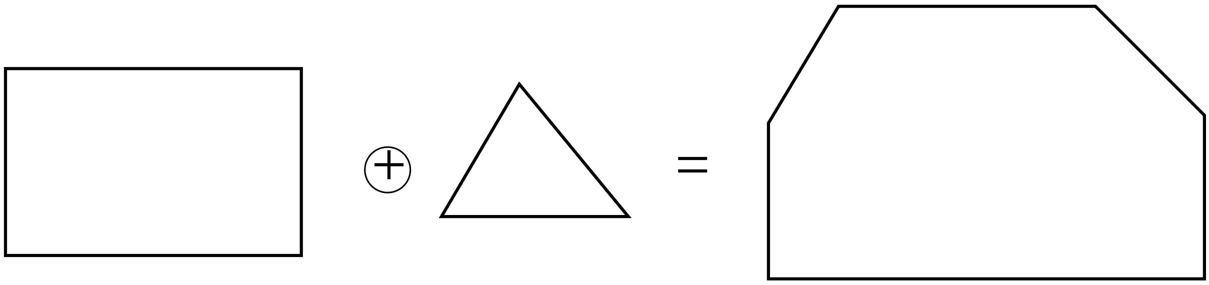


Figure 2: Minkowski sum of two convex polyhedra

Minkowski Sum Given two objects P and Q, the Minkowski sum $M = P \oplus Q$ of is given by

$$M = \{\mathbf{p} + \mathbf{q} : \mathbf{p} \in P, \mathbf{q} \in Q\} \tag{2.1}$$

Figure 2 shows an example of two convex polygons and its Minkowski sum. If P and Q are convex, then so is M. In fact, closure under Minkowski sums extends to the class of star-shaped objects. The algorithm to compute the Minkowski sum of two convex polytopes is relatively straightforward but exhibits quadratic complexity. Therefore, software approaches to compute the Minkowski sum can be fairly time consuming.

Given two convex polytopes P and Q in $\mathbb{R}^3$ and let V_p and V_q be their respective set of vertices. V_p^* (V_q^*) is the dual arrangements of the vertices of P (Q).

Then the upper (lower) envelope in the dual arrangement of the vertices of $M = P \oplus Q$ can be obtained by *adding the height values* of the upper (lower) envelopes of V_p^* and V_q^*. The main intuition in this observation is that the extremal point of M in any particular direction is obtained by the sum of extremal points of P and Q in the same direction. This observation is true only in the case of convex polytopes.

3 Penetration Depth

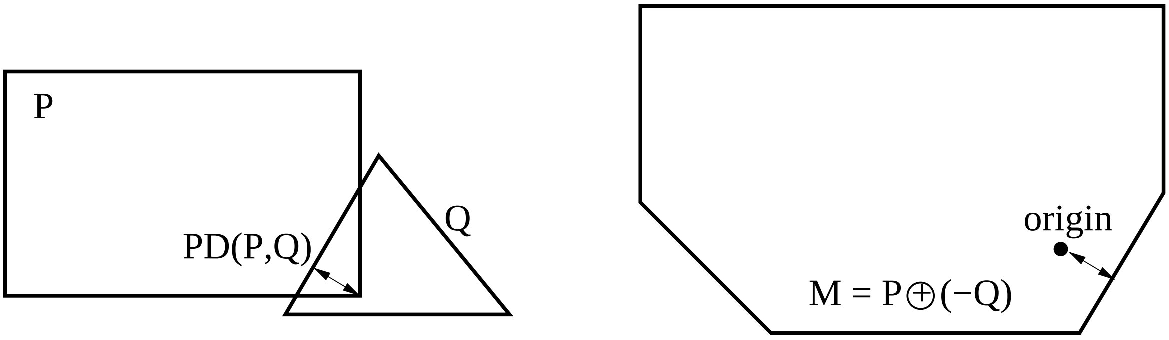


Figure 3: Penetration depth of two convex polyhedra and its relation to Minkowski sum

Given two convex polytopes P and Q, their penetration depth, denoted $PD(P,Q)$ is defined as the smallest translation vector t such that P and $Q + t$ are disjoint. It is known that penetration depth is the minimum distance from the origin to the polytope $M = P \oplus -Q$.

$$PD(P,Q) = \min_{z \in \partial M} d(o,z),$$

where o is the origin and ∂M denotes the boundary of M. It was shown by [1] that it is equivalent to compute the distance from the origin to the set of all tangent planes to the polytope M.

We will now use this fact to compute the penetration depth without explicitly computing the Minkowski sum of P and $-Q$. We had observed in the section on Duality that the set of all tangent planes to a polytope A can be obtained by looking at the lower and upper envelopes of the dual arrangement of V_a^* and that for the case of Minkowski sums this arrangement can be obtained by adding the corresponding envelope “heights” of the polytopes P and $-Q$. Our algorithm for penetration depth is:

1. Compute vertices of $-Q$.

2. For each vertex of P and $-Q$, compute its dual hyperplane

3. Compute the upper and lower envelope of the dual arrangement V_p^*

4. Compute the upper and lower envelope of the dual arrangement V_{-q}^*

5. Add the heights of the individual upper and lower envelopes

6. Find tangent planes to the polytope M and compute distance from the origin.

7. Take minimum

Use of graphics hardware Most of the operations detailed above can be solved remarkably well using the graphics hardware. It is specially suited to compute lower and upper envelopes by a simple use of the depth buffer. Moreover, the technique is very efficient because the primitives to be rendered are simple quads. However, we do have a required readback to compute the minimum over all the distances. A simple pseudocode to compute envelopes and the penetration depth using the hardware is provided in Algorithms 3.1 and 3.2 respectively.

Algorithm 3.1 Pseudocode for the lower (upper) envelope computation

```
for $i = 1$ to $n$ do {/* For each piecewise-linear function $f_i$ */}
  for $j = 1$ to $m_i$ do {/* For each vertex in function $f_i$ */}
    /* Let vertex $v_{ij} = (x_{ij}, y_{ij}, z_{ij})$ */
    Set color value of $v_{ij}$ appropriately depending on problem
  end for
end for
/* Compute lower (upper) envelope in color buffer */
Set depth test to $\min$ $(\max)$
Render all functions in $F$
Readback the color buffer
/* Color value at each pixel determines envelope */
```

Algorithm 3.2 Pseudocode for penetration depth computation

```
/* Given two convex polytopes $P$ and $Q$ */
for $i = 1$ to $3$ do {/* For each dual */}
  Compute dual planes $\mathbb{D}^p$ and $\mathbb{D}^q$ for vertices of $P$ and $Q$
  Set color of each vertex based on its $z$-coordinate
  Compute lower envelope of $\mathbb{D}^p$ in color buffer
  Copy color buffer to texture memory
  Compute lower envelope of $\mathbb{D}^q$ in color buffer
  Enable blending and set blending function to ADD
  Use texture mapping to access contents of texture
  Compute component-wise sum with the color buffer
  Readback the color buffer
  /* Repeat this for upper envelopes */
  for each pixel in color buffer do
    Compute distance from origin based on color value
    Track minimum
  end for
end for
/* Minimum value and pixel location determine penetration depth and direction */
```

This algorithm has been implemented on a standard graphics card using OpenGL. A naive implementation of the above algorithm is competitive with some of the best known software solutions, and in most cases outperforming them. For a detailed description of our experimental results, refer to the accompanying paper on “Geometric Optimization”.

4 Conclusion

This note was meant to provide an easy-to-read description of the penetration depth algorithm that was described in the paper accompanying the course notes. The algorithm presented here can easily be extended to maintain closest feature pairs between convex polytopes.

References

[1] AGARWAL, P. K., GUIBAS, L. J., HAR-PELED, S., RABINOVITCH, A., AND SHARIR, M. Computing the penetration depth of two convex polytopes in 3d. In *Scandinavian Workshop on Algorithm Theory* (2000), pp. 328–338.

[2] KIM, Y. J., LIN, M. C., AND MANOCHA, D. Fast penetration depth estimation between polyhedral models using hierarchical refinement. In *International Workshop on Algorithmic Foundations of Robotics (to appear)* (2002).

[3] LIN, M. C., AND GOTTSCHALK, S. Collision detection between geometric models: A survey. *Proc. of IMA Conference on Mathematics of Surfaces* (1998). Available at: `http://www.cs.unc.edu/~geom/papers/subject.shtml`.

[4] VAN DEN BERGEN, G. Proximity queries and penetration depth computation on 3d game objects. *Proc. of Game Developer's Conference* (2001). Available at: `http://www.win.tue.nl/~gino/solid/index.html`.

Fast and Approximation Computation of Geometric Arrangements using GPUs II

Dinesh Manocha
University of North Carolina at Chapel Hill

Efficient Computation of A Simplified Medial Axis

Appeared in the Proceedings of the 8th ACM Symposium on Solid Modeling and Applications, June 16-20, 2003

Mark Foskey
foskey@cs.unc.edu
Department of Radiology

Ming C. Lin
lin@cs.unc.edu
Department of Computer Science

Dinesh Manocha
dm@cs.unc.edu
Department of Computer Science

University of North Carolina at Chapel Hill
http://www.cs.unc.edu/ geom/smx/

ABSTRACT

Applications of of the medial axis have been limited because of its instability and algebraic complexity. In this paper, we use a simplification of the medial axis, the θ-SMA, that is parameterized by a separation angle (θ) formed by the vectors connecting a point on the medial axis to the closest points on the boundary. We present a formal characterization of the degree of simplification of the θ-SMA as a function of θ, and we quantify the degree to which the simplified medial axis retains the features of the original polyhedron.

We present a fast algorithm to compute an approximation of the θ-SMA. It is based on a spatial subdivision scheme, and uses fast computation of the distance field and its gradient using interpolation-based rasterization hardware. The complexity of the overall algorithm varies based on the error threshold used by the approximation scheme and is a linear function of the input size. We have applied this algorithm to approximate the SMA of complex models composed of tens or hundreds of thousands of triangles. Its running time varies from a few seconds, for a model consisting of hundreds of triangles, to minutes for highly complex models on a 2-GHz PC.

Categories and Subject Descriptors

I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Curve, surface, solid, and object representations*; I.4 [**Image Processing and Computer Vision**]: Reconstruction, Image Representation

General Terms

Algorithms, Experimentation, Performance, Theory

Keywords

Distance field, Medial axis

1. INTRODUCTION

The medial axis [5] of a solid, defined as the set of centers of maximal balls contained in the solid, has been proposed as a tool for shape analysis, surface reconstruction, motion planning, and many other applications. It is useful because it provides a local lower-dimensional characterization of the solid. In particular, for a solid in 3D the medial axis consists of a union of surfaces that provide information about the shape and topology of the solid. If the distance to the boundary is also stored for each medial axis point, the resulting structure is known as the medial axis transform (MAT) and the entire boundary representation can be reconstructed from it.

The use of the medial axis has been limited mainly by two significant drawbacks: It is *unstable*, in that small deformations in the boundary of the solid can lead to large changes in the medial axis. It is also difficult to compute because of the underlying algebraic complexity. For a polyhedron, the surfaces constituting the medial axis are quadrics, and the seam curves can have degree four. For solids with curved boundaries, the medial axis sheets and seam curves can have much higher degree. Geometric computation with primitives of such high degree is is hard to make both reliable and fast.

There have been numerous approaches to these problems. While exact algorithms have been proposed to compute the MAT for relatively simple polyhedra, it is non-trivial to scale them to very complex models composed of tens or hundreds of thousands of faces. As a result, most of the practical algorithms attempt to compute an approximation to the MAT. Different approximation algorithms, based on using a uniform grid, a spatial subdivision, or a point sampling of the surface, have been proposed in the literature. A number of techniques have also been proposed to simplify these approximations, in terms of reducing the number of geometric primitives or pruning away portions that can cause instability. We will give a review of this literature in Section 2.

In this paper, we primarily deal with a subset of the medial axis, which we call the *θ-simplified medial axis*, or θ-SMA. The θ refers to the angle formed by the vectors connecting a point on the medial axis to its corresponding closest points on the object boundary. We call this angle the *separation angle*, and the θ-SMA is simply the set of medial axis points for which the separation angle exceeds θ. The relationship between the stability of the medial axis and the separation angle has been known in the literature and used in many applications including surface reconstruction and skeleton-based modeling [1, 13, 26].

Main Results: We present novel properties of the θ-SMA and a fast algorithm to compute an approximation of the θ-SMA of a complex polyhedron. The θ-SMA, as indicated above, is parameterized by a minimum separation angle θ. It has the property that $M_{\theta_i} \subset M_{\theta_j}$ whenever $\theta_i > \theta_j$. Moreover, M_θ more closely approximates the medial axis as $\theta \to 0$, and becomes more stable as $\theta \to \pi$. We describe a formal characterization of the simplification of the medial axis as a function of θ. Given the distance function at each point on M_θ, an approximation to the boundary of the original solid can be reconstructed, and we give a formula relating the tightness of this approximation to θ.

We also present a novel and fast algorithm to compute an approximation to M_θ at an adjustable resolution ϵ. The ϵ determines the maximum error between the computed approximation and θ-SMA. It is based on efficient computation of a distance field and its gradient using a spatial decomposition. The complexity of the resulting algorithm is $\Theta(n/\epsilon^3)$ where n is the number of primitives in the model and ϵ is the resolution (voxel width). We describe an adaptive subdivision scheme for computing a bounded-error approximation. Moreover, we present a number of techniques to improve the quality of the approximation by smoothing operations and accelerate the performance of the overall algorithm.

The algorithm has been implemented and applied to complex polyhedra composed of tens or hundreds of thousands of triangles. Its running time ranges from a few seconds for a model composed of hundreds of triangles to minutes for highly complex models on a 2 GHz PC with an nVidia GeForce 4 graphics card.

As compared to other approximate schemes, our approach offers the following advantages:

- **Complex Models:** It can handle very large and complex models as the running time is a linear function of the input size.
- **Efficiency:** We use fast algorithms for computing the distance field and its gradient based on interpolation-based rasterization hardware. As a result, our algorithm can handle complex models composed of tens of thousands of polygons in a few minutes.
- **Approximation:** The ϵ-approximation to the θ-SMA is everywhere within $\sqrt{3}/2\ \epsilon$ of the medial axis, and it converges to the true θ-SMA as $\epsilon \to 0$.
- **Stability:** The criterion for simplification is scale-invariant, so that small shallow bulges are ignored, but thin, extended features in the medial axis are represented.
- **Simplification:** The simplification criterion is rather intuitive, depending only on the separation angle.

The rest of the paper is organized as follows. In Section 2 we give an overview of related work. In Section 3 we define M_θ and present some of its properties. In Section 4 we present our algorithm, and in Section 5 we analyze the time complexity of our algorithms and various sources of error in the approximation. We describe our implementation in Section 6 and highlight its performance on a number of complex models. In Section 7 we compare our approach to others in the literature, and we conclude in Section 8.

2. RELATED WORK

There is an extensive literature on both the computation and the simplification of the MAT and related constructions. In this section, we give a brief overview of exact and approximate algorithms for MAT computation as well as simplification.

2.1 Medial Axis Computation

At a broad level, algorithms for medial-axis computation can be classified into four categories: thinning algorithms, distance field based algorithms, algebraic methods, and surface-sampling approaches. These categories differ in terms of the underlying representations used for the medial-axis as well as how they compute it.

2.1.1 Thinning Algorithms

Thinning algorithms use a voxel-based representation of the initial figure, and perform erosion operations to arrive at a set of voxels approximating the medial axis. Lam et al. [19] give a survey of these approaches, and Zhang et al. [32] compare various methods. These methods are significant in the areas of image processing and pattern recognition, since the input data is represented as a discrete grid. We also use a voxel-based spatial decomposition to localize regions containing the medial surfaces. This is followed by an extraction step to represent the medial axis as a union of polygonal surfaces.

2.1.2 Distance Field Computations

Many approaches compute an approximation of the medial axis based on distance fields. Danielsson [12], uses a scanning approach in 2D to create an image in which each pixel contains the Euclidean distance to the nearest pixel on the boundary of the figure being analyzed. Moreover, the resulting distance map can be analyzed for local directional maxima to get an approximation of the medial axis. This algorithm has also been extended to three and higher dimensions [22].

Vleugels and Overmars [30] use a spatial subdivision to represent the medial axis, relying on nearest-neighbor queries to determine whether a cell must be further subdivided. They subdivide if the cell has vertices in different Voronoi regions and is larger than a certain threshold.

Hoff et al. [17] use graphics hardware to render a polygonal approximation of the distance field. The interpolation-based rasterization hardware is used to store the distance field in the depth buffer. We have extended this algorithm to compute the gradient of the distance field, also using rasterization hardware. We then use the gradient field for fast computation of the medial axis.

Siddiqi et al. [25] have also presented an approximate algorithm based on distance fields. Their analysis is based on a differential equation simulating the inward progress of a front starting at the boundary of the object. They compute a vector field that, at every point $\mathbf{p}$, is equal to the vector from the nearest point on the surface, to $\mathbf{p}$. Given the fact that this vector field points towards the medial axis from both sides, Siddiqi et al. consider a point to be on the medial axis if the mean flux of the vector field, entering a neighborhood of the point, is positive. This algorithm has been designed assuming that the input is represented in terms of voxels.

2.1.3 Algebraic Methods

There is a family of methods that rely fundamentally on the fact that the algebraic form is explicitly known for each surface patch (i.e., each sheet) of the medial axis of a polyhedron.

Etzion and Rappoport [16] represent the curves and surfaces symbolically, but use a spatial subdivision to resolve the connectivity of the curves. They use algebraic tests to determine whether the surfaces pass into the cells of the subdivision, and subdivide until either the proper connectivity is determined, or a minimum cell size is reached. The presence of a minimum cell size means that it is not always possible to isolate all vertices and fully resolve the local connectivity of seams (and hence the surfaces they bound).

Most algorithms that represent the medial axis symbolically use a *tracing* approach [21, 23]. Starting from a junction point on the medial axis, a seam emanating from the junction is followed. The seam terminates at another junction and the process is applied recursively. Chiang [8] describes an algorithm for computing the medial axis of a planar region bounded by piecewise C^2 curves. The algorithm involves tracing branches using systems of polynomial equations. Sherbrooke et al. [24] present a variation on the algorithm. Their method explicitly traces along the seam, creating a piecewise-linear approximation to the seam curves.

Culver et al. [11] use exact computation to represent the curves and surfaces of the 3D medial axis. Their method is a tracing approach that computes an exact representation of the medial axis of a polyhedron provided there are no degeneracies (such as more than four or seams intersecting at a point). They also use a spatial subdivision technique to improve the running time of the overall algorithm. Dutta and Hoffmann [15] and Hoffmann [18] present an approach to compute the medial axes of constructive solid geometry (CSG) models.

All of the methods in this family have been applied to polyhedra composed of only a few hundred faces. It is not clear whether they can be either applied to complex models composed of tens or hundreds of thousands of faces. Either their running time is more than $O(n^2)$, where n is the number of faces, or these algorithms are susceptible to accuracy and robustness problems.

2.1.4 Surface Sampling Approaches

Surface sampling methods represent the initial figure as a dense cloud of sample points presumed to be on or near the boundary. The medial axis of the figure is approximated by a subset of the Voronoi diagram of the point cloud. Different algorithms based on this approach use different methods for selecting the desired subset of the Voronoi diagram. Many such variations have been proposed. Boissonnat [6] classified certain triangles of the Delaunay tetrahedralization of the point cloud as interior to the model; the Voronoi vertices dual to those tetrahedra approximate the medial axis.

Using a similar approach, Amenta et al. [1] construct an approximate, simplified medial axis which they use as a stage in a surface reconstruction from the original point cloud, a common application for this approach. Dey and Zhao [13, 14] also create a simplified surface model of a medial axis. Turkiyyah et al. [29] focus on improved accuracy rather than simplification. They follow the initial approximation with a numerical optimization step to move the sample points so that the Voronoi vertices are closer to the true medial axis. Both [1] and [13] have given good surveys of the literature on surface sampling medial axis approaches.

These algorithms have been applied to models composed of tens of thousands of points. One of the main issues when applying these algorithms to polyhedral models is in generating appropriate point samples on the boundary to ensure a tight approximation of the medial axis. In general, the worst-case running time of these algorithms can be $O(n^2)$, where n is the number of point samples. Recently, Attali and Boissonnat [2] have shown that the running time is only linear when the points are distributed on a fixed number of well-sampled facets. However, the point sampling of the surface has to satisfy certain criteria.

2.2 Medial Axis Simplification

A fundamental problem with the medial axis as a tool in shape analysis and surface reconstruction is that it is *unstable*, in the sense that small perturbations in the surface model lead to large changes in the structure of the medial axis. For a polyhedral model, every pair of adjacent faces produces a medial axis sheet extending to the edge connecting the two faces, producing a cluttered and uninformative medial axis. A number of methods for simplifying the medial axis have been proposed.

One of the criteria to identify parts of a medial axis that are stable is what we call the separation angle $S(\mathbf{x})$. It is the maximum angle formed by the vectors connecting the medial axis point $\mathbf{x}$ to its closest points on the boundary, and portions of the medial axis with a larger separation angle tend to be more stable. This has been noted by several researchers based on analyzing functions on the boundary surface [4], investigating the effect of noise [3] or samples [7] on the medial axis or in other skeleton-based applications [26]. Amenta et al. [1] use a similar criterion to determine whether a point on the medial axis is stable.

Dey and Zhao [13, 14] use a pair of criteria to retain faces from the Voronoi diagram of a set of points. For one criterion, they consider the angle between an approximate inward-pointing surface normal and a Delaunay edge (dual to a Voronoi face). If that angle is small, the Voronoi face is retained. The other criterion retains Voronoi faces if they are much farther from the surface sample points than the sample points are from each other.

Styner et al. [27] iteratively merge and prune sheets according to a pair of cost functions designed to minimize the change to the reconstructed model. They achieve a substantial reduction in medial axis complexity while retaining better than 98% volume overlap with the original model.

Choi and Seidel [10] study the stability of the medial axis and derive a bound on one measure of the instability of the medial axis for solids satisfying certain hypotheses.

3. θ-SIMPLIFIED MEDIAL AXIS

In this section we formally define the θ-SMA and give some of its properties. While the relationship between the separation angle and stability is well known, we are not

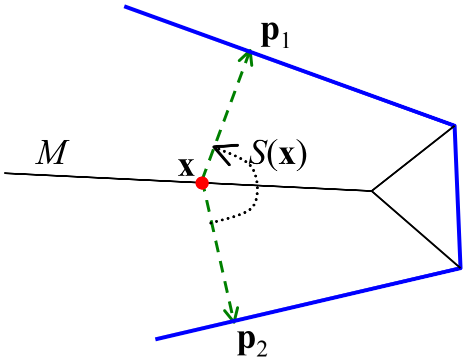


Figure 1: The separation angle $S(\mathbf{x})$ for a point on the medial axis. The thick border is the boundary of X.

aware of this particular subset of the medial axis being studied as an object in its own right. We show the degree to which it is more stable than the medial axis. Moreover, we define the θ-SMAT, which includes the distance information just as the MAT does, and show that the original model can be reconstructed from the θ-SMAT to an accuracy that depends in a simple way on θ. The significance of this relationship is that it is a way of quantifying the importance of the portion of the medial axis retained in the θ-SMA. If the original model can be reconstructed with reasonable accuracy, then one can argue that the most significant portions of the medial axis are being preserved.

Notation and Terminology: In this paper, vectors and points will be in boldface. Sets and functions will generally be denoted by capital letters. Unless otherwise specified, X will denote a solid with a polyhedral boundary.

Given a set of geometric primitives $S = \{P_i\}$, the *Voronoi region* of a primitive P_i is the set of points that are at least as close to P_i as to any other primitive. The collection of Voronoi regions is the *generalized Voronoi diagram*, or GVD. The medial axis of a polyhedron is a subset of the GVD of its faces, edges, and vertices.

We will say that an edge or vertex of X is *reflex* if its incident faces are not coplanar and it intersects the boundary of a ball whose interior lies in the interior of X.

Let X be a polyhedral solid with medial axis M. Recall that M can be characterized as the closure of the set of points in the interior of X having at least two nearest neighbors on the boundary of X. (Sometimes the requirement that the points be in the interior of X is relaxed.) Consider a point $\mathbf{x} \in M$, and let $NS(\mathbf{x})$ denote the set of its nearest neighboring points on the boundary of X. There is a sphere centered at $\mathbf{x}$ that does not cross the boundary of X, but that touches it at just the points of $NS(\mathbf{x})$.

For each pair of points $\mathbf{p}_1, \mathbf{p}_2 \in NS(\mathbf{x})$, we can consider the angle $\angle \mathbf{p_1 x p_2}$. (We will treat all angles as values in $[0, \pi]$.) If $\mathbf{x}$ has more than two nearest neighbors, then we consider the largest angle subtended by a pair of nearest neighbors. We call this angle the *separation angle* $S(\mathbf{x})$ for the medial axis point $\mathbf{x}$:

$$S(\mathbf{x}) = \max_{\mathbf{p}_1,\mathbf{p}_2 \in NS_{\mathbf{x}}} (\angle \mathbf{p}_1\mathbf{x}\mathbf{p}_2)$$

(see Figure 1).

The intuitive motivation for this definition is as follows: If the separation angle is exactly π, then $\mathbf{x}$ is directly between its nearest neighbors, while if the angle is small, then both neighboring points are on the same side of $\mathbf{x}$, and there is space on the other side of $\mathbf{x}$ that is, in a natural sense, deeper in X.

Given an angle θ, define the *θ-simplified medial axis* M_θ of X to be the set of points of M with separation angle greater than θ. When we wish to emphasize the relationship of M_θ to a particular solid X, we will write $M_\theta(X)$. The following facts follow from the definitions of M and M_θ:

- The θ-SMAs are nested, with larger angles implying smaller subsets. That is, if $0 < \theta_1 < \theta_2 < \pi$, then $M_{\theta_2} \subset M_{\theta_1}$.
- $\overline{\bigcup_{\theta \in (0,\pi)} M_\theta} = M$, where $\overline{X}$ denotes the closure of X.

In this sense we can say that $M_\theta \to M$ as $\theta \to 0$. Note that, even though we specified X as a polyhedral solid, all of the above applies to any solid.

3.1 Quantifying the Significance of θ

In this section we derive a formula that quantifies the degree to which the θ-SMA retains the significant portions of the medial axis. It is well known that X can be reconstructed from M along with the radius values for each point on M. If we use M_θ instead of M in the reconstruction, then we get a subset of X, which we can call X_θ. The accuracy with which X_θ approximates X is a measure of the degree to which M_θ captures the important geometric features of X. Next, we formalize these notions, explaining what we mean by the accuracy of the approximation, and show how the accuracy is related to the separation angle θ, used as the angle cutoff in simplifying the original medial axis.

Formally, the *medial axis transform* is the *set* of all maximal balls contained in X. The centers of the balls constitute the medial axis, and retaining the balls is equivalent to retaining the radius information associated to each medial axis point. We define the *θ-simplified medial axis transform* (θ-SMAT) to be the subset of the MAT consisting of those balls centered on points of the θ-SMA. X can be reconstructed as the union of all the maximal balls in the MAT of X, and $X_\theta \subset X$ is the union of the balls in the θ-SMAT of X. By construction, M_θ is the medial axis of X_θ, but note that X_θ may not correspond to a polyhedron.

We can measure how closely X_θ approximates X in two ways. First, we can compare the volumes of the two spaces, computing the ratio $\mathrm{Vol}(X)/\mathrm{Vol}(X_\theta)$, where $\mathrm{Vol}(X)$ denotes the volume of X. Second, we can look at the distance between points on the boundary of X_θ and the nearest neighboring points on X. For each point $\mathbf{p}$ on the boundary of X_θ, there is a well-defined *local radius* $R(\mathbf{p})$ given by the radius of the smallest maximal ball touching $\mathbf{p}$ (see Figure 1). We can measure the local error as the distance from $\mathbf{p}$ to its nearest neighbor $\mathbf{p}'$ on the boundary of X, as compared to the local radius of $\mathbf{p}$. That is, the local error $E(\mathbf{p})$ is defined by

$$E(\mathbf{p}) = \frac{\|\mathbf{p} - \mathbf{p}'\|}{R(\mathbf{p})},$$

where $\mathbf{p}'$ is the point on the boundary of X that is nearest to $\mathbf{p}$.

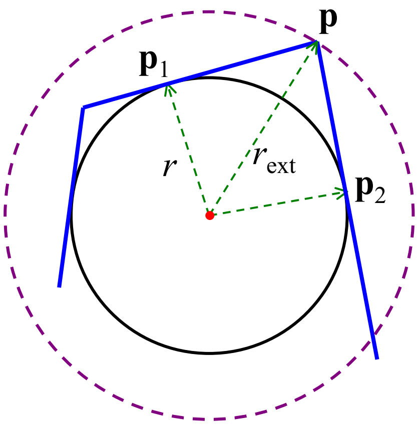


Figure 2: Computing the error bound. The angle subtended by $\mathbf{p}_1$ and $\mathbf{p}_2$ is equal to θ, so no circle tangent only to the edges containing $\mathbf{p}_1$ and $\mathbf{p}_2$ is represented in M_θ. If the solid circle is enlarged into the dashed circle, then the vertex $\mathbf{p}$ will be included. The radius r is equal to the local radius $R(\mathbf{p}_1) = R(\mathbf{p}_2)$.

The following theorem shows how well X_θ approximates X, as a function of θ.

Theorem 1. *Let*

$$g(\theta) = \frac{1}{\sqrt{1 - \frac{4}{3}\sin^2\frac{\theta}{2}}}.$$

Then $\mathrm{Vol}(X)/\mathrm{Vol}(X_\theta) \leq g(\theta)^3$ *and, for each point* $\mathbf{p}$ *on the boundary of* X_θ*,* $E(\mathbf{p}) \leq g(\theta) - 1$.

Proof. We claim that, if all the balls of the θ-SMAT are enlarged by a factor of $g(\theta)$, then their union will contain X. The largest local feature that can be excluded from X_θ is a corner such that the normals to the respective faces differ by an angle no greater than θ. If all the balls are enlarged by an appropriate ratio to include such corners, then their union will include all of X. We will argue that $g(\theta)$ as defined above is the required ratio by which all the balls of the θ-SMAT must be enlarged to include all of X.

First consider the 2D analogue (Figure 2). If there are two adjacent edges whose normals differ by an angle equal to the threshold angle θ, then no disk tangent only to those two edges will be added to the medial axis transform. Hence a disk such as the one shown will be the medial axis disk that is closest to the vertex $\mathbf{p}$, and the external radius r_{ext} is the radius to which that disk must be enlarged to contain all of that corner. Thus, in two dimensions, $g(\theta) = r_{\text{ext}}/r = \sec(\theta/2)$.

In three dimensions, the corresponding situation consists of three faces coming together at $\mathbf{p}$ such that each pair of normals differs by θ. Let $\mathbf{x}$ be the center of the maximal ball closest to the extremal vertex $\mathbf{p}$, and let $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ be the points nearest to $\mathbf{x}$ on each of the three faces meeting at $\mathbf{p}$ (see Figure 3).

Consider the planes passing through $\mathbf{x}$, $\mathbf{p}$, and $\mathbf{p}_i$ for $i = 1$, 2, and 3. Since each pair of normals differs by the same angle, these planes must have dihedral angles of $2\pi/3$ to one another, and the points $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ form an equilateral triangle in a plane orthogonal to $\overline{\mathbf{xp}}$. Let $\mathbf{q}$ be the point where this plane crosses $\overline{\mathbf{xp}}$, and let $\mathbf{q}'$ be the midpoint of $\overline{\mathbf{p}_1\mathbf{p}_2}$. Let $r_{\text{ext}} = \|\mathbf{p} - \mathbf{x}\|$, and $r = \|\mathbf{p}_1 - \mathbf{x}\|$. Then our goal is to compute the ratio r_{ext}/r. For convenience, assume without loss of generality that $r = 1$, so that we only need to compute r_{ext}.

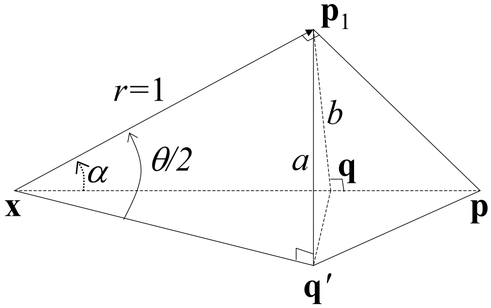


Figure 3: Computing $r_{\text{ext}} = \|\mathbf{p} - \mathbf{x}\|$. The vector $(\mathbf{p}_1 - \mathbf{x})$ is a normal vector from $\mathbf{x}$ to a face on the boundary of X. There are two other such faces; the endpoints of the normal vectors to those faces, $\mathbf{p}_2$ and $\mathbf{p}_3$, are not shown. The point $\mathbf{q}$ bisects $\overline{\mathbf{p}_1\mathbf{p}_2}$, which is one side of an equilateral triangle.

If we denote $\angle\mathbf{p}_1\mathbf{x}\mathbf{p}$ by α, then $r_{\text{ext}} = 1/\cos\alpha$. We will compute $\sin\alpha$. Let $a = \|\mathbf{p}_1 - \mathbf{q}'\|$, and $b = \|\mathbf{p}_1 - \mathbf{q}\|$. Then $b = \sin\alpha$, and $a = \sin(\theta/2)$. (This is because $\angle\mathbf{p}_1\mathbf{x}\mathbf{p}_2 = \theta$.) Also, $a = b\sqrt{3}/2$ because $\overline{\mathbf{p}_1\mathbf{q}'}$ is perpendicular to $\overline{\mathbf{q}\mathbf{q}'}$, and $m\angle\mathbf{p}_1\mathbf{q}\mathbf{q}' = \pi/3$. Bear in mind that $\mathbf{q}$ is the center of the equilateral triangle $\triangle\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$. Thus,

$$\sin\alpha = b = \frac{2}{\sqrt{3}}a = \frac{2}{\sqrt{3}}\sin\theta/2.$$

Therefore,

$$r_{\text{ext}} = \frac{1}{\sqrt{1 - \sin^2\alpha}} = \frac{1}{\sqrt{1 - \frac{4}{3}\sin^2\frac{\theta}{2}}}$$

□

3.2 Stability and Connectivity

In this section we discuss the stability of M_θ, that is, how much it is altered by small changes in X. We also observe that M_θ is not guaranteed to preserve the connectivity properties of X.

One of the benefits of the θ-SMA is that it is more stable than the medial axis. The medial axis of a finely tessellated polyhedron will have a sheet for every adjacent pair of faces, and many other pairs as well. The θ-SMA will only retain sheets for pairs of faces whose normals differ by an angle greater than θ, and thus, whose respective dihedral angles are less than $\pi - \theta$. Thus, introducing new vertices to generate a finer tessellation of the model will not create new sheets of the θ-SMA unless the new faces that are introduced to the polyhedron create sufficiently small angles with each other or with other faces in the model.

However, by design the θ-SMA detects small, elongated features, as in Figure 4. If such features are expected to arise as noise, then the θ-SMA will be affected by the noise. The relationship of θ to the stability of the simplified medial axis is illustrated by Figure 5.

The θ-SMA does not in general preserve the homotopy type of the model. It can be disconnected and have holes, even if X is simply connected. In Figure 4, $M_\theta(X)$ is shown for $\theta \approx \pi/3$. The point $\mathbf{x}$ is on the medial axis of the space

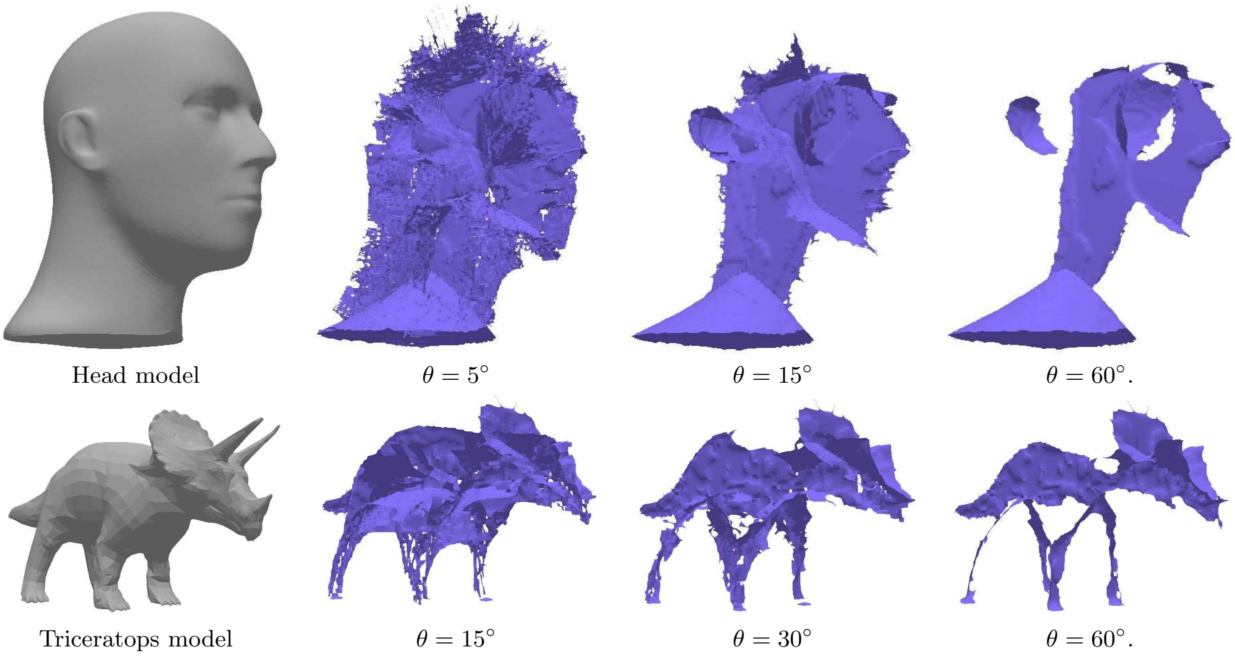


Figure 5: Different Θ-SMA for the same model. As the separation angle increases, the number of high frequency or sharp components decreases.

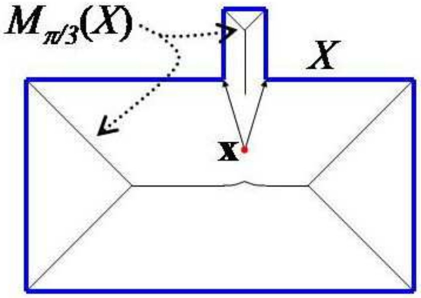


Figure 4: Disconnectedness. The point $\mathbf{x}$ is on the medial axis but has a small separation angle.

X, but not on $M_{\pi/3}$ because its separation angle is too low. If $\mathbf{x}$ were to move towards the rectangular feature at the top, the separation angle would increase until it exceeded the threshold angle, at which point $\mathbf{x}$ would be on $M_{\pi/3}$.

This lack of a connectivity guarantee can be problematic for some applications. However, for others, what is desired is a characterization of the geometric properties of an object whose connectivity may already be understood. Also, it is possible to use a larger value of θ to select significant components of the object, and then compute again with a smaller value of θ to achieve improved connectivity. The more connected version can then be pruned, retaining just enough information to connect the components corresponding to the larger value of θ.

Finally, if it is desired to simplify X, one can remove small detached components of M_θ, yielding a pruned version of the θ-SMA. After that one can reconstruct an approximation to X from the pruned θ-SMA.

4. ALGORITHM

In this section we present a fast algorithm to approximate $M_\theta(X)$. The algorithm has two variations, one based on a uniform voxel grid, and the other on an adaptive subdivision of space. We first give an overview of the algorithm. We then describe in more detail the criterion we use to determine whether to add a face to the representation of M_θ, after which we describe the different spatial subdivision approaches. We conclude by describing two approaches to improving the surface representation of the θ-SMA.

The algorithm is based on a vector field that we call the *neighbor direction field* of X, denoted N_X. If $\mathbf{x}$ is a point having a unique nearest neighbor $\mathbf{p}$ on the boundary of X, then

$$N_X(\mathbf{x}) = \frac{1}{\|\mathbf{p} - \mathbf{x}\|}(\mathbf{p} - \mathbf{x}).$$

This field consists of the negated gradients of the distance field defined by the boundary of X, and it is well-defined everywhere outside the boundary and medial axis of X.

Using N_X, we define a *separation criterion* to determine whether an arbitrary line segment in the interior of X crosses a sheet of the medial axis. The essence of the criterion is that two points $\mathbf{x}_1$ and $\mathbf{x}_2$ are taken to be on opposite sides of a medial axis sheet if $N_X(\mathbf{x}_1)$ and $N_X(\mathbf{x}_2)$ diverge. We use this criterion to test either the centers of the voxels of a uniform grid, or the cell vertices of an adaptive subdivision.

When a pair of points passes the separation criterion, we add a facet between them to our model of M_θ. Once the polygonal model is generated, it can be filtered to improve the fit of the represented sheets to those of the actual θ-

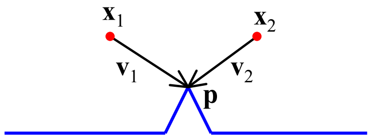


Figure 6: The direction vectors at neighboring voxels can differ by a large angle even when the voxels are not on different sides of the medial axis.

SMA.

4.1 The Separation Criterion

For a given pair of points we first determine the angle between the respective direction vectors given by N_X. If the angle is not greater than the threshold θ then we reject the pair. However, if it is greater than the threshold, we need to be careful to avoid false positives. If, say, a reflex vertex is the nearest neighbor to both points in a pair, then both direction vectors will converge towards the vertex (see Figure 6).

If the points are close enough to the vertex, then the angle between the vectors can be greater than the threshold, even though the segment between the points does not cross the medial axis.

To avoid this error, we need to check whether the vectors diverge. We check this condition by ensuring that the heads of the vectors are at least as far apart as the tails, where the lengths of the vectors are scaled to equal the separation between the neighboring points.

Given the separation criterion, we present algorithms for two spatial subdivision schemes, namely a uniform grid and an adaptive grid.

4.2 Uniform Subdivision

The simplest spatial subdivision is a uniform grid. There are efficient ways to compute a distance field and its gradient that make use of the uniformity of the grid [17, 12]. We extend the algorithm presented in [17] for fast computation of the distance field.

Our goal is to create a uniform sampling of the direction field of the model X. We divide the volume into an axis-aligned voxel grid, referring to a set of voxels with a constant z-value as a *slice*. The algorithm we use relies on the parallel nature of interpolation-based graphics hardware to perform the computation efficiently for one slice at a time. The algorithm simultaneously computes a distance field and a direction field over a uniform 2D grid for each slice. We will describe the computation of the distance field first and then explain how we use it for direction computation. For each slice, the distance field is a scalar function $D_X : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. If we decompose X into sub-objects X_i, then D_X is determined by the lower envelope (or minima) of the set of all the distance functions D_{X_i}. We thus decompose X into its faces, edges, and vertices and compute the lower envelope of the distance fields of each of these primitives.

The distance functions of these primitives can be represented in a simple form. We highlight these functions for points, lines and planes. For edges and triangular faces, these definitions are combined in piecewise fashion to represent the full distance field for the primitive. We describe formulas for the slice $z = 0$ and for primitives placed in particularly convenient configurations. The general forms can be derived by simple coordinate transformations.

For a point $\mathbf{p} = (0, 0, c)$ and the slice $z = 0$, the distance field is the hyperboloid

$$D_{\mathbf{p}}(x, y) = \sqrt{x^2 + y^2 + c^2}.$$

For a point with arbitrary coordinates we perform a translation on the distance field.

For the line L in the xz plane given parametrically by $(ta, 0, tc)$ with $a^2 + c^2 = 1$, the distance is given by the elliptical cone

$$D_L(x, y) = \sqrt{x^2c^2 + y^2}.$$

For a general line, we perform a translation and a rotation.

Finally, let F be the plane defined by the equation $ax + by + cz + d = 0$. If we assume that a, b, and c are chosen so that $a^2 + b^2 + c^2 = 1$, then the distance from a point to the plane is simply found by evaluating the left-hand side of the equation at that point. Thus, for the slice $z = 0$ we have

$$D_F(x, y) = ax + by + d.$$

In 3-space, the Voronoi region of the interior of a triangular face is defined by the three planes perpendicular to the face and passing through the edges. Points in this region are closer to the interior of the triangle than any of its edges or vertices. Similarly, the Voronoi region of the interior of a segment is defined by the planes normal to the segment and passing through the endpoints. For points outside the Voronoi region of the interior of a face or segment, we define the distance to be infinite by convention. Then, when the lower envelope of the distance fields for all the faces, edges and vertices is taken, the proper nearest neighbor will be determined for each point.

As we generate the distance field for each primitive, we also generate the direction field for that particular primitive. The distance field allows the lower envelope to be defined, and the lower envelope determines, for each point in the volume, which primitive defines the direction field at that point. With the point $\mathbf{p}$, the line L, and the plane F defined as above, unnormalized direction fields are given by

$$\begin{aligned} N_{\mathbf{p}}(x, y) &= (-x, -y, c) \\ N_L(x, y) &= (-xc^2, -y, xac) \\ N_F(x, y) &= -(ax + by + d)(a, b, c) \end{aligned}$$

To extract M_θ, we construct N_X for each slice by combining the direction fields for the primitives of X. We then evaluate each pair of voxels in the x, y, and z directions, adding a face to the approximate θ-SMA for each pair that passes the separation test. The computation of the distance field and the direction field maps very well to the rasterization hardware. More details are given in Section 6.

4.3 Adaptive Subdivision

Given the non-linear nature of the medial axis, in many applications it is possible to compute a better approximation by using a non-uniform grid. We present an algorithm based on octree subdivision of the space. This approach requires two primitive operations. First, one needs to evaluate the neighbor direction field $N_X(\mathbf{x})$ at an arbitrary point $\mathbf{x}$ in the volume of interest. Second, one must be able to determine whether an axis-aligned box contains, overlaps, or is contained by the object X. Both of these tests can be performed quickly using either a spatial subdivision to index the faces of the boundary of X, or by using a bounding volume hierarchy of X. There are standard collision detection packages that also provide the capability for distance queries, an example being PQP [20]. While these algorithms have been designed for object-object distance computation, it is straightforward to modify them to handle point-object computation. For instance, given a bounding volume hierarchy of the object, one can compute the feature on X that is closest to $\mathbf{x}$ by computing the distance from $\mathbf{x}$ to the bounding volumes at different levels in the hierarchy. Given the closest feature, the algorithm also computes the direction vector from $\mathbf{x}$.

Using these two primitive operations, the algorithm is as follows:

1. Begin with a single cell containing X.
2. Until a chosen cell size is reached, iteratively subdivide the cells that either
   (a) contain at least part of X but are not contained in X, or
   (b) are contained in X and have a pair of neighboring cell vertices that meet the separation criterion.
3. For each pair of vertices meeting the separation criterion, add a face to the medial axis as in the uniform grid approach.

This algorithm is more memory efficient, as well as more time efficient (in terms of operation count), than the the uniform grid algorithm. However, the uniform subdivision scheme is simpler to implement and maps well to the rasterization hardware.

4.4 Refining the Medial-Axis Approximation

The polyhedral approximation generated by the spatial subdivision schemes represents the θ-SMA up to a specified resolution. However, the sheets of the medial axis (which correspond to a portion of a quadric surface) are not well approximated by the axis-aligned facets of the voxel grid. In this section, we present two methods to refine the medial axis approximation.

Smoothing. When we use uniform subdivision of space, the algorithm we we use to compute the distance field produces distance values with a bounded error, with a bound equal to half the diagonal width of a voxel. For this reason, we cannot use the distance mesh to achieve subpixel accuracy in placing the faces of the medial axis mesh. However, we use the smoothing algorithm proposed by Taubin [28], a fast, non-shrinking smoothing filter. Because this filter is non-shrinking, it retains the shape of the medial axis sheets, while avoiding the stair-stepping appearance of the axis-aligned faces.

Iterative Retraction. We have described an approach to perform distance queries (to the boundary) and direction computation at arbitrary points in the space. Based on this information, we can quickly find points that are very close to the medial axis. Let $\mathbf{x}$ be a point in the interior of X, but not on the medial axis. Let $\mathbf{p}$ be the unique point on the boundary of X nearest to $\mathbf{x}$, and let $\mathbf{p}'$ be any other point on the boundary of X. Then $\mathbf{p}'$ places an upper bound on how far $\mathbf{x}$ can be from the medial axis. Let $S_{\mathbf{x},\mathbf{p}}$ be the sphere centered at $\mathbf{x}$ and passing through $\mathbf{p}$, with radius $\|\mathbf{x}-\mathbf{p}\|$. Since $\mathbf{x}$ is not on the medial axis, $S_{\mathbf{x},\mathbf{p}}$ is not a maximal sphere. Let $S_{\max}$ be the maximal sphere containing $S_{\mathbf{x},\mathbf{p}}$. Then $S_{\max}$ exhibits the following properties:

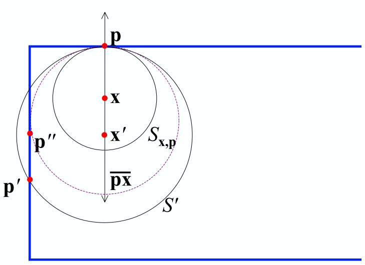


Figure 7: The first step in an iterative refinement of the approximate θ-SMA. $\mathbf{x}$ is the initial guess, and $\mathbf{p}$ is the nearest neighbor of $\mathbf{x}$ on the boundary. $S_{\mathbf{x},\mathbf{p}}$ is the center centered on $\mathbf{x}$ and passing through $\mathbf{p}$. There is a maximal circle $S_{\max}$ (not shown) that is contained in X and contains $S_{\mathbf{x},\mathbf{p}}$. We approach the medial axis by approaching $S_{\max}$. $S_{\max}$ touches the boundary in at least two places. $\mathbf{p}'$ and $\mathbf{p}''$ are successive approximations to to the second point where $S_{\max}$ touches the boundary of X.

- It will be tangent to $S_{\mathbf{x},\mathbf{p}}$, and hence centered on the line $\overline{\mathbf{px}}$ passing through $\mathbf{p}$ and $\mathbf{x}$.
- It will be no larger than the unique sphere S' centered on $\overline{\mathbf{px}}$ and passing through both $\mathbf{p}$ and $\mathbf{p}'$, because S' already touches two points on the boundary of X.

The center $\mathbf{x}'$ of S' is the most distant possible point from $\mathbf{x}$ where $\overline{\mathbf{px}}$ could cross the medial axis. See Figure 7.

In this way, points on the medial axis are computed using an iterative algorithm. The algorithm proceeds in the following manner: Once S' is found, we can define $\mathbf{x}'$ to be its center, and choose a new point $\mathbf{p}''$ on the boundary of X as the boundary point nearest $\mathbf{x}'$. Using $\mathbf{p}''$ in place of $\mathbf{p}'$, we construct S'', and repeat the process. As we perform more iterations, a sequence of circles is constructed that approaches $S_{\max}$. This algorithm fits quite well with the adaptive subdivision approach as now we can compute vertices that are very close to the medial axis. Our method is similar to a method used by Wilmarth, Amato, and Stiller [31] to retract randomly generated sample points to the medial axis.

5. ANALYSIS

In this section, we analyze the performance of our algorithm. This includes the accuracy of our approximation as well as the running time.

5.1 Accuracy

In this section we show that the discrete approximation computed by our algorithm converges to the actual θ-SMA as the resolution becomes arbitrarily fine. As before, let X be a polyhedral subspace of $\mathbb{R}^3$, and let M be the medial axis of X. Let $\dot{X}$ denote the interior of X. For a given $\epsilon > 0$, let θ-simplified medial axis, $M_{\epsilon,\theta}$, be the approximation produced by our algorithm at the resolution ϵ.

The idea of our argument is that our algorithm estimates the set of points over which the neighborhood direction field

N_X is discontinuous. The following theorem says that, inside X, the direction field can only be discontinuous at the medial axis. Since the direction field is not defined on the medial axis, it follows that, in $\dot{X}$, the medial axis is precisely the set of discontinuities of the direction field. We prove it based on the following theorem.

THEOREM 2. *The neighbor direction field N_X is continuous on the space $\dot{X} \setminus M$.*

PROOF. Let $\mathbf{x} \in \dot{X}$ be a point not on M. Either $\mathbf{x}$ is in a Voronoi cell of one of the faces, edges, or vertices of the boundary of X, or it is on the boundary between the Voronoi cells of a reflex edge and a face, or between the cells of a reflex vertex and a reflex edge. For each of these cases the distance field can be computed explicitly and is shown to be continuous in a neighborhood of $\mathbf{x}$. □

The result does hold for all curvilinear shapes of practical interest as well, but there are pathological cases where it fails. Choi, Choi, and Moon [9] give examples of such pathological cases in two dimensions, along with easily-satisfied criteria to ensure that a region does not exhibit such behavior.

THEOREM 3. *For a given $\epsilon > 0$, the ϵ, θ-SMA is within a Hausdorff distance of $\sqrt{3}\epsilon/2$ from a subset of the medial axis of X.*

PROOF. Let F be any face of $M_{\epsilon,\theta}$. F is a square face separating two voxels that satisfied the separation criterion. Let $\mathbf{x}_1$ and $\mathbf{x}_2$ denote the centers of those two voxels. Note that F is nowhere more than $\sqrt{3}\epsilon/2$ from the most distant point on the segment $\overline{\mathbf{x}_1\mathbf{x}_2}$, because each cubical voxel has side ϵ, and the distance from the center of a cube to the farthest point on its face is $\sqrt{3}/2$ times the side of the cube. We will show that the medial axis of X passes between $\mathbf{x}_1$ and $\mathbf{x}_2$. It follows that no point on F is farther than $\sqrt{3}\epsilon/2$ from some point on the medial axis.

Let $\mathbf{p}_i$ be the point on the boundary of X that is nearest $\mathbf{x}_i$, for $i = 1, 2$. The separation criterion ensures that the $\mathbf{p}_i$ are farther apart than the $\mathbf{x}_i$, which implies that the $\mathbf{p}_i$ are on different features (faces, edges, or vertices) of the polyhedral boundary of X. This inference follows by considering each type of feature in turn. Certainly the $\mathbf{p}_i$ cannot be on the same vertex. If the $\mathbf{p}_i$ are on the same edge, then the lines L_i containing $\mathbf{p}_i$ and $\mathbf{x}_i$ for each i are perpendicular to the edge. Thus the distance from $\mathbf{p}_1$ to $\mathbf{p}_2$ is the nearest distance from L_1 to L_2, and so $\mathbf{x}_1$ can be no closer to $\mathbf{x}_2$. The same reasoning applies to show that the $\mathbf{p}_i$ cannot be on the same face.

For each t with $0 \leq t \leq 1$, define $\mathbf{x}(t)$ to be $(1-t)\mathbf{x}_1 + t\mathbf{x}_2$, so that $\mathbf{x}(t)$ traverses the segment $\overline{\mathbf{x}_1\mathbf{x}_2}$. For each t, let $\mathbf{p}(t)$ be the nearest neighbor to $\mathbf{x}(t)$, if the neighbor is unique. There cannot be a path traversed by $\mathbf{p}(t)$ from $\mathbf{p}_1$ to $\mathbf{p}_2$ that only crosses reflex vertices and edges, since the direction vectors converge towards such edges, and the vectors at the endpoints of the segment diverge. Hence, there must be an intervening convex edge or vertex, resulting in a discontinuity in the direction field. Therefore, $\overline{\mathbf{x}_1\mathbf{x}_2}$ crosses the medial axis, and hence F is entirely within the specified bound. □

We have shown $M_{\epsilon,\theta}$ is within a bounded distance of the medial axis of X. It remains to show that it actually converges to M_θ.

THEOREM 4. *The ϵ, θ-SMA converges to the θ-SMA in Hausdorff distance as $\epsilon \rightarrow 0$.*

PROOF. We need only consider the sheets, since the distance from the seams to the sheets is zero. Consider a point $\mathbf{x}$ on a sheet of M_θ. Let $\mathbf{v}_1$ and $\mathbf{v}_2$ be the unit direction vectors to its two nearest neighbors. Recall that $S(\mathbf{x})$ denotes the separation angle for $\mathbf{x}$, that is, the angle between $\mathbf{v}_1$ and $\mathbf{v}_2$. Since $\mathbf{x}$ is on M_θ, $S(x) > \theta$. Let $\eta = (S(x) - \theta)/2$. By the continuity of N_X, there is a neighborhood B containing $\mathbf{x}$ such that the angle between $N_X(\mathbf{x}')$ and $\mathbf{v}_i$ is less than η for each $\mathbf{x}'$ in B and on the same side of the medial axis as v_i.

Now for sufficiently small ϵ, there will be adjacent voxels in the lattice of $M_{\epsilon,\theta}$ such that both voxels are contained in B, and the voxel centers are on opposite sides of the medial axis. Each such pair of voxels determines a face of $M_{\epsilon,\theta}$ that is contained in B. Since this applies for any sufficiently small neighborhood of $\mathbf{x}$, this shows that the minimum distance from $\mathbf{x}$ to $M_{\epsilon,\theta}$ can be made arbitrarily small. Combined with Theorem 3, this completes the proof. □

5.2 Time Complexity

In this section we discuss the time complexity of our algorithm. For the uniform grid approach, the analysis depends on the computational model that is used for the graphics hardware. If we assume that the hardware takes a a constant amount of time to render the distance field for each primitive, then the algorithm we use to compute the direction field requires time $\Theta(p/\epsilon)$ where ϵ is the resolution and p is the number of primitives (faces, edges, and vertices) in the model. Extracting the θ-SMA requires a single pass through the volume, requiring time proportional to $1/\epsilon^3$, so that the running time for the entire algorithm is $\Theta(p/\epsilon + 1/\epsilon^3)$. If we assume that rendering a slice of a distance field takes time proportional to the number of voxels in the slice, then the total time is $\Theta(p/\epsilon^3)$ or, equivalently, $\Theta(pv)$, where v is the number of voxels.

For the approach using an adaptive subdivision, the running time is highly output sensitive. We note that each distance query can require time logarithmic in the size of the model, using current techniques, but the constant factor is quite small.

6. IMPLEMENTATION AND RESULTS

In this section we describe the implementation of our algorithm and highlight its performance on a number of complex benchmarks.

6.1 Implementation

We implemented the system in C++ using Microsoft Visual Studio, with OpenGL as our graphics API. Our implementation for computing the distance field is based on the techniques described in Hoff et al. [17]. In that approach, a volume is processed one slice at a time. For each slice, and each geometric primitive in the model, a surface, called a *distance mesh*, is rendered such that the depth buffer contains the shortest 3-space distance from each point in the slice to the given geometric primitive (which may not be in the given slice). If a pixel from a given primitive's distance mesh passes the depth test, then the pixel is in that primitive's Voronoi region.

We extend this method to acquire direction information as well, by encoding the directions to the nearest primitive

Model	Tris	Resolution	$\mathbf{T}(N_{\mathbf{X}})$	T(SMA)
Bent Torus	2,000	127x128x42	5.42	0.321
Cassini PM	90,879	23x32x24	141	0.661
Cassini PM	90,879	94x128x96	1329	45.6
Buddha 1	15,536	55x128x55	35.7	5.5
Buddha 2	67,240	222x512x222	1634	48.8
Buddha 3	1,087,474	55x128x55	1588	1.17
Skel. Hand	654,666	79x106x127	602	0.07
Elbow Pipe	5,306	96x59x77	6.95	1.10
Elbow Pipe	5,306	128x79x103	10.8	3.87
Elbow Pipe	5,306	192x119x155	20.4	5.59
Elbow Pipe	5,306	256x159x207	33.1	8.24
Elbow Pipe	5,306	512x318x414	127	69.3
Bunny	69,451	64x63x50	77.4	0.12
Bunny	69,451	128x126x100	238	0.982
Bunny	69,451	256x253x200	794	2.51
Head	21,764	31x41x50	13.3	0.09
Head	21,764	79x106x127	57.8	0.22
Primer Anvil	4,340	128x73x112	8.99	0.61
Shell Charge	4,460	126x128x126	33.0	10.9

Table 1: Timings for some models at various resolutions. The Buddha model is shown at three different levels of detail. **Model**: Name of the model. **Tris**: Number of triangles in the model. **Resolution**: Number of voxels along each dimension. $\mathbf{T}(\mathbf{N_X})$: Time to compute the neighbor direction field. T(**SMA**): Time to extract the θ-SMA. All timings are in seconds on a 2Ghz Pentium 4 with an nVidia geForce 4 graphics card.

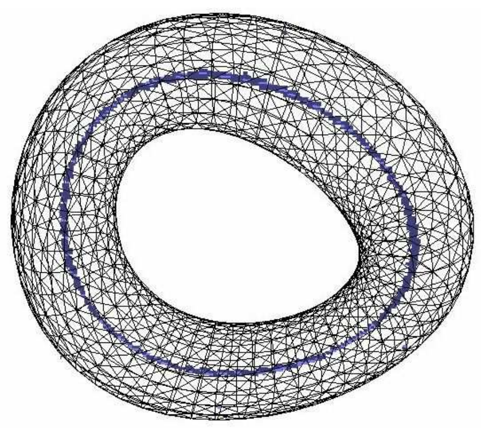


Figure 8: A torus and its θ-SMA. 2000 triangles. The grid resolution is 127x128x42.

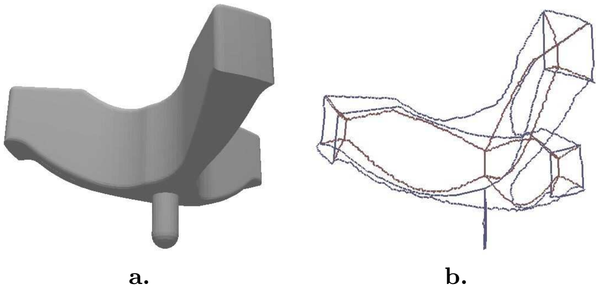


Figure 9: The "primer anvil" for a shotgun shell. 4,340 triangles, SMA computed at 128x73x112 resolution. (a) The model. (b) The θ-SMA. The seams and boundary curves of the θ-SMA are shown.

in the red, green, and blue channels of the color buffer. Both the directions and distances are linearly interpolated across each triangle of the distance mesh, which is a source of error that grows with the size of the triangles. The distance mesh is designed to adjust the size of the triangles to keep the error within acceptable bounds.

We encode the gradient vector at each vertex of the distance mesh. It is important to keep in mind that each triangle is part of a distance mesh associated both to a geometric primitive and a slice of the volume. The slice corresponds to a z-value in the volume, but the z-coordinates of the rendered triangles correspond to distances from the slice to the primitive. The *colors*, likewise, correspond to directions from points *on the slice* to the primitive. Henceforth, when we refer to a given triangle of a distance mesh, we imply the projection of the triangle onto the specified slice.

Each direction vector is of unit length, with the x, y, and z components represented by the red, green, and blue color components respectively. As the components are interpolated across the triangle, the magnitude differs from unit length, so that the vectors must be normalized after being read back and before computing a dot product to test the separation angle. The direction differs from the true direction. Consider a vertex which we can assume to be located at the origin, and a particular slice located at some depth z. Let $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ be the vertices of a mesh triangle that has been projected into the plane of the slice. Then a point in the triangle can be expressed as a sum $\sum t_i \mathbf{p}_i$ with t_i chosen so that $\sum t_i = 1$ (that is, in barycentric coordinates). The true unit vector pointing towards the vertex is

$$\mathbf{v} = -\frac{\sum t_i \mathbf{p}_i}{\|\sum t_i \mathbf{p}_i\|},$$

while the estimated vector is

$$\tilde{\mathbf{v}} = -\frac{\sum t_i \hat{\mathbf{p}}_i}{\|\sum t_i \hat{\mathbf{p}}_i\|},$$

where $\hat{\mathbf{p}} = \mathbf{p}/\|\mathbf{p}\|$. Then the error in the direction is given by the angle $\cos^{-1}(\mathbf{v} \cdot \tilde{\mathbf{v}})$. We do not have a good bound on this expression other than to say that it is bounded by the largest angle subtended by the triangle from the point of view of the primitive. We also note that, for triangles, which are treated as primitives separately from their edges and vertices, there is no interpolation error because the direction vector is constant. Thus, for a voxel inside a convex polyhedron, the only source of error in direction is the fact that each component of the vector can only be expressed with 8 bits of precision in the color buffer.

An alternative approach is to encode the full vector from each point on the slice to the given primitive, rather than a unit-length direction vector. This approach raised concerns with discretization error. However, with the advent of floating-point color buffers, that objection may not be a concern in the future.

6.2 Benchmark Models

We applied our algorithm to polygonal models of various sizes, ranging from 2,000 triangles to more than 1 million.

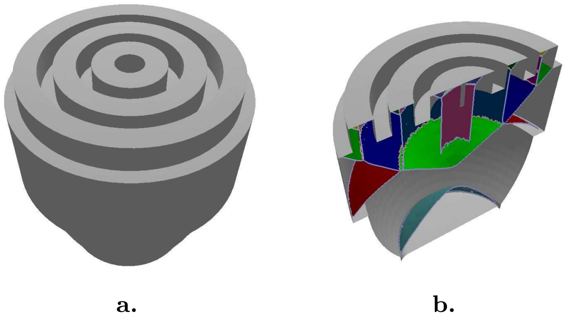


Figure 10: Shotgun shell "charge" with 4460 triangles. The grid resolution is 126x128x126. (a) The model. (b) Cross section, showing different sheets in different shades.

Some of the models were triangulations of scanned data, and others were CAD models. In general, scanned models have triangles with good aspect ratios and uniformly distributed over its boundary. However, many of these models have a high genus. On the other hand, the CAD models tend to have many sharp edges and uneven or high-aspect-ratio triangles.

In our analysis, X has been a solid with a polyhedral boundary. Our models are polygonal, and some of them do not bound solids. The definition of the medial axis extends naturally to such cases, and to solids for which we wish to analyze the exterior as well as the interior. For models that do have a well-defined interior, our implementation has an option to compute the medial axis for the interior only. As an optimization, when only the interior medial axis is being computed, distance meshes for convex vertices and edges are not rendered during distance field computation.

6.3 Performance

In our tests (Table 6.1), the bulk of the computation time is taken by the computation of distance fields. Comparing running times for different resolutions shows an increase that is more than linear but less than cubic in the number of voxels along one dimension.

Except where otherwise specified, the separation angle $\theta = 60^\circ$, and the θ-SMAs have been smoothed. The resolution is specified in terms of the dimensions of the scene. The relative dimensions of the volume were determined by slightly enlarging a tight bounding box for the model. Then the number of voxels along the longest dimension could be chosen, which governed the number of voxels along the other dimensions. All the voxels are cubical, as the dimensions of the bounding box were adjusted to be an integral number of voxels in each direction.

7. COMPARISON WITH OTHER APPROACHES

There are by now a large array of approaches for computing as well as simplifying the medial axis. Performance comparisons between them are difficult, because they make different assumptions about the input, and generate different kinds of medial axis approximations as output.

The two main features of our approach are, first, that it computes the θ-SMA and not the entire medial axis, and, second, that we use a fast algorithm based on uniform spatial subdivision to compute the distance field and its gradient. As a result, we are able to compute good approximations of the θ-SMA for complex models in a few minutes.

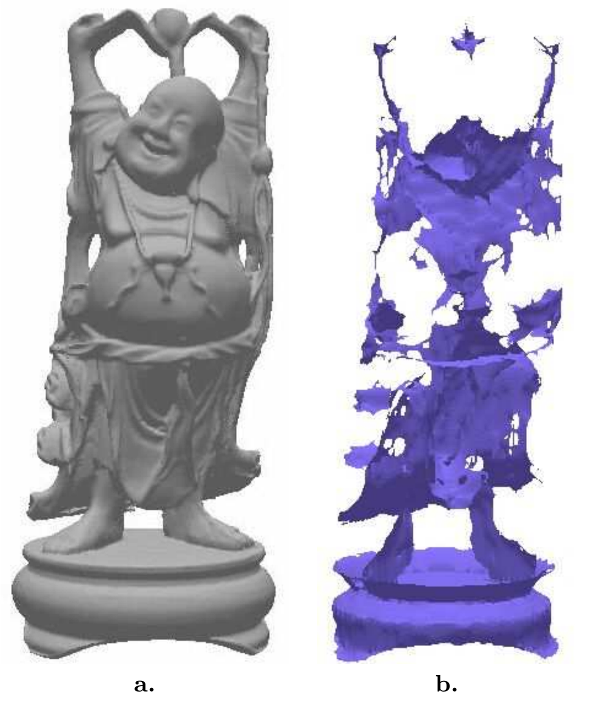


Figure 11: Buddha model with 1,087,474 triangles. The grid resolution is 55x128x55.

Tracing algorithms [11, 23, 24] are much more time consuming and have only been applied to models consisting of a few hundred triangles. The adaptive subdivision algorithms of [30] and [16] computed the generalized Voronoi diagram, rather than the medial axis. Their methods were only applied to models with up to a few hundred polygons. Our algorithm with adaptive subdivision is similar in approach to that of [30], though we use much faster sub-algorithms for distance computation.

The surface sampling approaches such as those of [6], [1], and [13], take point samples on the surface as input, rather than the boundary features of a polyhedron. The accuracy and topology of the resulting medial axis varies considerably based on the sampling criterion used to generate the point samples. This makes it difficult to compare the approaches. Amenta et al. [1] report times of roughly six minutes for models of around $30,000$ points, and Dey et al. [13] process around $122,000$ points in a little over five minutes.

8. CONCLUSION

We have presented a medial axis approximation, the θ-SMA, based on the idea of the separation angle for a point on the medial axis. The criterion characterizing the θ-SMA is easy to understand and analyze, and it results in a more stable structure than Blum's medial axis. In practice, it is able to detect and capture most of the sharp features of the original model. We have presented a formal characterization of the simplification of θ-SMA as a function of θ.

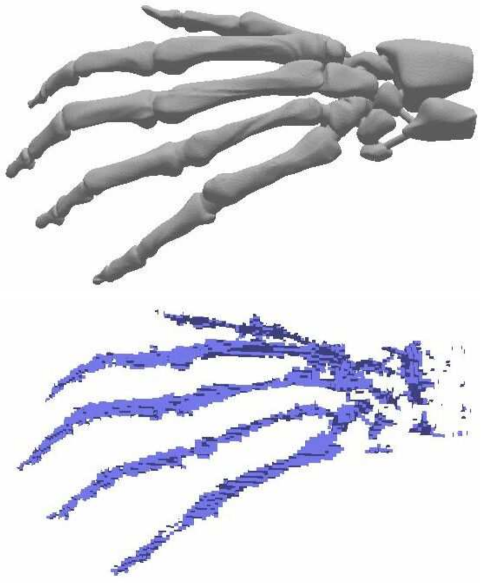


Figure 12: Skeleton hand with 654,666 triangles. The grid resolution is 79x106x127. No smoothing was performed.

We have described two algorithms for fast approximating the θ-SMA. One, using a uniform grid, is well-suited for implementation using the parallel features of modern graphics hardware. We have highlighted its performance on a number of complex benchmarks. The other algorithm uses an octree decomposition, in order to reduce the memory expense and make the time efficiency more output-dependent. Both algorithms fit into a consistent framework; both produce approximations that remain within a specified distance of some part of the full medial axis. We have analyzed the approximation errors produced by our algorithm.

There are a number of areas of future work. The key to our use of graphics hardware is that the direction vector field, stored as RGB triples, is associated to the scalar distance field, represented as depth values. This approach could be applied more generally to other pairs of associated vector and scalar fields. The use of graphics hardware for general computing purposes is currently an active area of research. We would like to compute the θ-SMA of solids with curved boundaries as well as procedural models. Moreover, we would like to use θ-SMA for different applications including mesh generation and shape analysis.

9. ACKNOWLEDGMENTS

We would like to thank Nancy Amato, the Stanford University Computer Graphics Laboratory, the Georgia Tech Large Models Archive, and NASA, for the use of polyhedral models. We also thank Kenny Hoff for his HAVOC software and for many helpful discussions, and we thank the reviewers for a number of suggestions on the exposition. Finally, this work has been supported in part by ARO Contract DAAD19-99-1-0162, NSF awards ACI 9876914, IIS-982167, ACI 0118743, ONR Contracts N00014-01-1-0067 and N00014-01-1-0496, by the National Library of Medicine, and by Intel Corporation.

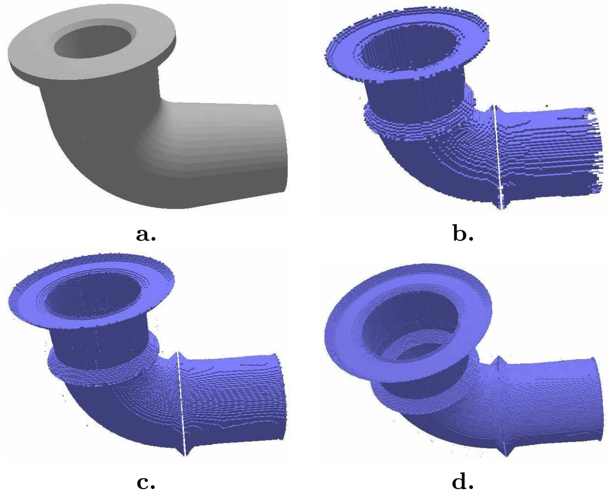


Figure 13: Elbow pipe, at varying resolutions. (a) The model. Figures (b), (c) and (d), correspond to 128, 256, and 512, voxels along the longest side, respectively. The gap visible in (b) and (c) shows where the interior of the pipe model is separated by a surface into two compartments. The gap is not visible in (d) only because the angle of the scene is slightly different. The θ-SMA is not smoothed.

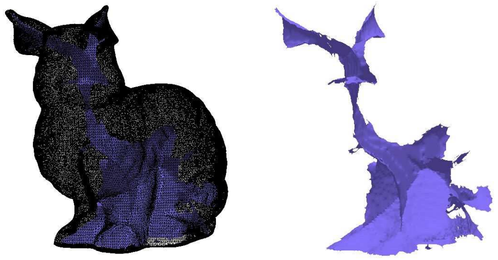


Figure 14: Bunny. 69,451 triangles, 128x126x100. (a) The bunny in wireframe, with the medial axis. (b) The θ-SMA.

10. REFERENCES

[1] N. Amenta, S. Choi, and R. Kolluri. The power crust. In *ACM Symposium on Solid Modeling and Applications*, pages 249–260, 2001.

[2] D. Attali and J. Boissonnat. A linear bound on the complexity of the delaunay triangulation of points on polyhedral surfaces. In *Proc. of ACM Solid Modeling*, pages 139–146, 2002.

[3] D. Attali and A. Montanvert. Computing and simplifying 2d and 3d continuous skeltons. *Computer Vision and Image Understanding*, 67(3):261–273, 1997.

[4] J. August, A. Tannebaum, and S. Zucker. On the evolution of the skelton. In *Proc. of Int. Conf. on Computer Vision*, 1999.

[5] H. Blum. A transformation for extracting new descriptors of shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, 1967.

[6] J.-D. Boissonnat. Geometric structures for three-dimensional shape representation. *ACM Trans. Graph.*, 3(4):266–286, 1984.

[7] J.-D. Boissonnat and F. Cazls. Smooth surface reconstruction via natural neighbour interpolation of distance functions. pages 223–232, 2000.

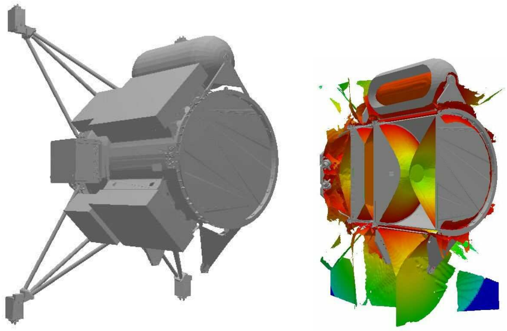


Figure 15: Propulsion module of the Cassini spacecraft. 90,879 triangles, 94x128x96. (a) The model. (b) The model in cross section, with the θ-SMA shaded based on the distance to the boundary. In this example, the θ-SMA is not restricted to the interior of the model.

[8] C.-S. Chiang. *The Euclidean distance transform*. Ph.D. thesis, Dept. Comput. Sci., Purdue Univ., West Lafayette, IN, Aug. 1992. Report CSD-TR 92-050.

[9] H. I. Choi, S. W. Choi, and H. P. Moon. Mathematical theory of medial axis transform. *Pacific Journal of Mathematics*, 181(1):56–88, 1997.

[10] S. W. Choi and H.-P. Seidel. Linear onesided stability of mat for weakly injective 3d domain. In *7th ACM Sympos. Solid Modeling Applications*, pages 344–355, 2002.

[11] T. Culver, J. Keyser, and D. Manocha. Accurate computation of the medial axis of a polyhedron. In *Proc. Symposium on Solid Modeling and Applications*, pages 179–190, 1999.

[12] P.-E. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

[13] T. K. Dey and W. Zhao. Approximate medial axis as a Voronoi subcomplex. In *7th ACM Sympos. Solid Modeling Applications*, 2002.

[14] T. K. Dey and W. Zhao. Approximating the medial axis from the Voronoi diagram with a convergence guarantee. In *European Symposium on Algorithms*, 2002.

[15] D. Dutta and C. M. Hoffmann. A geometric investigation of the skeleton of CSG objects. In *Proc. ASME Conf. Design Automation*, 1990.

[16] M. Etzion and A. Rappoport. Computing the Voronoi diagram of a 3-D polyhedron by separate computation of its symbolic and geometric parts. In *Proc. Symposium on Solid Modeling and Applications*, pages 167–178, 1999.

[17] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. *Proceedings of ACM SIGGRAPH 1999*, pages 277–286, 1999.

[18] C. M. Hoffmann. How to construct the skeleton of CSG objects. In *Proc. 4th IMA Conf. on The Mathematics of Surfaces*, Bath, UK, 1990. Oxford University Press.

[19] L. Lam, S.-W. Lee, and C. Y. Chen. Thinning methodologies—a comprehensive survey. *IEEE Trans. PAMI*, 14(9):869–885, 1992.

[20] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha. Fast proximity queries with swept sphere volumes. Technical Report TR99-018, Department of Computer Science, University of North Carolina, 1999.

[21] V. Milenkovic. Robust construction of the voronoi diagram of a polyhedron. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 473–478, 1993.

[22] I. Ragnemalm. The euclidean distance transformation in arbitrary dimensions. *Pattern Recognition Letters*, 14:883–888, 1993.

[23] J. Reddy and G. Turkiyyah. Computation of 3d skeltons using a generalized delaunay triangulation technique. *Computer-Aided Design*, 27:677–694, 1995.

[24] E. C. Sherbrooke, N. M. Patrikalakis, and E. Brisson. Computation of the medial axis transform of 3-D polyhedra. In *Proc. Symposium on Solid Modeling and Applications*, pages 187–199. ACM, 1995.

[25] K. Siddiqi, S. Bouix, A. Tannenbaum, and S. W. Zucker. The hamilton-jacobi skeleton. In *International Conference on Computer Vision*, pages 828–834, 1999.

[26] D. W. Storti, G. M. Turkiyyah, M. A. Ganter, C. T. Lim, and D. M. Stal. Skeleton-based modeling operations on solids. In *Proc. Symposium on Solid Modeling and Applications*, pages 141–154, 1997.

[27] M. Styner, G. Gerig, S. Joshi, and S. Pizer. Automatic and robust computatoin of 3D medial models incorporating object variability. *International Journal of Computer Vision*, To appear.

[28] G. Taubin. A signal processing approach to fair surface design. In *Proc. of ACM SIGGRAPH*, pages 351–358, 1995.

[29] G. M. Turkiyyah, D. W. Storti, M. Ganter, H. Chen, and M. Vimawala. An accelerated triangulation method for computing the skeletons of free-form solid models. *Comput. Aided Design*, 29(1):5–19, Jan. 1997.

[30] J. Vleugels and M. Overmars. Approximating generalized Voronoi diagrams in any dimension. Technical Report UU-CS-1995-14, Department of Computer Science, Utrecht University, 1995.

[31] S. A. Wilmarth, N. M. Amato, and P. F. Stiller. Motion planning for a rigid body using random networks on the medial axis of the free space. *Proc. of the 15th Annual ACM Symposium on Computational Geometry (SoCG'99)*, 1999.

[32] Y. Y. Zhang and P. S. P. Wang. Analytical comparison of thinning algorithms. *Int. J. Pattern Recognit. Artif. Intell.*, 7:1227–1246, 1993.

Fast Swept Volume Approximation of Complex Polyhedral Models

Young J. Kim Gokul Varadhan Ming C. Lin Dinesh Manocha

Department of Computer Science, UNC-Chapel Hill
{*youngkim,varadhan,lin,dm*}*@cs.unc.edu*

Abstract

We present an efficient algorithm to approximate the swept volume (SV) of a complex polyhedron along a given trajectory. Given the boundary description of the polyhedron and a path specified as a parametric curve, our algorithm enumerates a superset of the boundary surfaces of SV. It consists of ruled and developable surface primitives, and the SV corresponds to the *outer boundary* of their arrangement. We approximate this boundary by using a five-stage pipeline. This includes computing a bounded-error approximation of each surface primitive, computing unsigned distance fields on a uniform grid, classifying all grid points using fast marching front propagation, iso-surface reconstruction, and topological refinement. We also present a novel and fast algorithm for computing the signed distance of surface primitives as well as a number of techniques based on surface culling, fast marching level-set methods and rasterization hardware to improve the performance of the overall algorithm. We analyze different sources of error in our approximation algorithm and highlight its performance on complex models composed of thousands of polygons. In practice, it is able to compute a bounded-error approximation in tens of seconds for models composed of thousands of polygons sweeping along a complex trajectory.

Keywords: Computational geometry, Virtual environments and prototypes, Blends, sweeps, offsets & deformations, Geometric and topological representations

1 Introduction

Swept volume (SV) is the volume generated by sweeping a solid or a collection of surfaces in space along a smooth trajectory. The problem of SV computation arises in different applications, including NC machining verification [Blackmore et al. 1997; Boussac and Crosnier 1996], geometric modeling [Conkey and Joy 2000; Madrigal and Joy 1999], robot workspace analysis [Abrams and Allen 1995; Abdel-Malek and Yeh 1997a], collision detection [Kieffer and Litvin 1990; Xavier 1997], maintainability study [Law et al. 1998], ergonomic design [Abdel-Malek et al. 2002b], motion planning [Schwarzer et al. 2002], etc. A more extensive list of potential applications of SV can be found at [Abdel-Malek et al. 2002a].

The SV computation problem has been studied in different disciplines for more than four decades. This includes elegant work based on envelope theory, singularity theory, Lie groups, sweep differential equations on the characterization of the problem. As a result, the mathematical formulation of SV computation is relatively well-understood.

In many applications, the main goal of SV computation is to identify and extract the boundary of the SV, in particularly its outermost boundary. Most of the algorithms for computation of the boundary of SV are based, either explicitly or implicitly, on the following framework:

1. Find all the boundary primitives that contribute to the outermost boundary of SV.
2. Compute an arrangement of the boundary primitives by performing intersection and trimming computations.
3. Traverse the arrangement and extract the outer boundary. Here, the outer boundary of an arrangement is defined as the boundary of a cell, which is reachable from infinity following some continuous path, in the arrangement.

Most of the mathematical work has mainly dealt with characterizing the boundary primitives, given some assumptions on the sweeping path. There is a considerable amount of research in computational geometry on the combinatorial complexity of computing arrangements as well as on surface-surface intersection computations in geometric and solid modeling. However, the underlying combinatorial and algebraic complexity of exact SV computation is very high. Furthermore, the implementations of any algorithms for computing intersections and arrangements need to deal with accuracy and robustness issues. As a result, no practical algorithms are known for exact computation of the SV for any arbitrary polyhedron sweeping along a given smooth path.

Given the underlying complexity of exact SV computation, most of the earlier work has focussed on approximate techniques. Different algorithms can be characterized based on whether they are limited to 2D objects, or they only compute an image-space projection or visualization of the SV from a given viewpoint, or compute a relatively coarse discretization of the boundary primitives followed by union computation of different configurations of the polyhedra along the trajectory. These algorithms are either slow for practical applications, or suffer from robustness problems, or compute a rather coarse approximation of the SV.

Main Results We present an efficient algorithm to approximate the outermost boundary of SV's of complex polyhedral models along a given trajectory. The algorithm initially enumerates a superset of the boundary primitives of SV, which consists of *ruled* and *developable* surfaces [Weld and Leu 1990]. The ruled surface is generated by considering each edge in the original model as a ruling line and the trajectory as a directrix curve. The developable surface is obtained by applying the envelope theory to moving triangles. Given a formulation of the boundary elements, our algorithm computes an approximation to the resulting arrangement using a five-stage pipeline. Firstly, it computes a bounded-error polygonal approximation of each surface primitive. Secondly, it samples the surface primitives by computing unsigned, directed distance fields along the vertices of a grid. Next it classifies the grid points to be either inside or outside of the surfaces to obtain the signed distance field using a novel algorithm based on marching front propagation. This is followed by iso-surface reconstruction. Finally, the algorithm performs topological refinement, taking into account some of the characterizations of the SV computation. We also present a number of acceleration techniques based on culling of surface primitives, use of interpolation-based rasterization hardware for fast computation of distance field, and a variation of fast marching level-set method for classification of grid points.

Our algorithm computes a bounded-error approximation of the SV and we analyze all sources of error. We have implemented this algorithm on a commodity-based PC with nVidia GeForce 4 graphics card, and benchmarked its performance on complex benchmarks. The underlying polyhedral models consist of thousands of triangles and are sweeping along a complex trajectory corresponding to a parametric curve. The computation of SV takes a few tens of seconds on a 2.4GHz Pentium IV processor.

As compared to earlier approaches, the main advantages of our technique include:

- Generality: The algorithm can handle general 2-manifold polyhedral models, and makes no assumptions about the sweep path.

- Complex Models: The algorithm is directly applicable to complex models composed of a high number of features. Given a trajectory and a bound on the approximation error, the overall complexity increases as a linear function of the input size.

- Efficiency: The use of culling techniques and algorithms for signed distance field computation significantly improve the running time of the algorithm.

- Simplicity: The algorithm is relatively simple to implement and does not suffer from robustness problems or degeneracies.

- Good SV Approximation: Our preliminary application of the algorithm to different benchmarks indicates that it can compute a good, bounded-error approximation of the boundary.

Organization The rest of our paper is organized as follows. In Section 2, we briefly review the earlier work on SV computation. Section 3 provides the overview of our approach to SV computation. In Section 4, we present an algorithm to compute the boundary surface primitives of SV. Section 5 describes our approximation algorithm to compute the arrangement of the surface primitives using sampling and reconstruction. We analyze the performance of our algorithm in Section 6 and describe its implementation and performance in Section 7. In Section 8, we compare our algorithm with other earlier approaches.

2 Previous Work

In this section, we give a brief survey of the work related to SV computation, arrangements, and iso-surface reconstruction based on distance fields.

2.1 Swept Volume Computation

SV has been studied quite extensively over the years. We list some of the crucial development in the history of SV research here, but refer the readers to see [Abdel-Malek et al. 2002a] for more thorough survey of SV-related work.

Methodology The mathematical formulation of the SV problem has been investigated using singularity theory (or Jacobian rank deficiency method) [Abdel-Malek and Yeh 1997c; Abdel-Malek and Yeh 1997b; Abdel-Malek and Othman 1999], Sweep Differential Equation (SDE) [Blackmore and Leu 1990; Blackmore et al. 1997], Minkowski sums [Elber and Kim 1999], envelope theory [Martin and Stephenson 1990; Weld and Leu 1990], implicit modeling [Schroeder et al. 1994], and kinematics [Jüttler and Wagner 1996]. Moreover, most of this work deals with the SV of generic, free form objects.

Polyhedral Approximation Given the complexity of computing the exact SV, few algorithms have been developed to provide a polyhedral approximation of SV. In 2D, [Lee et al. 2002; Ahn et al. 1993] study an approximation of the general sweep for curved objects, and they have been applied to font design. In 3D, [Weld and Leu 1990] describe a geometric representation of SV for compact n-manifolds with application to polyhedral objects. [Schroeder et al. 1994] use discretized representations and iso-surface reconstruction to approximate SV, [Abrams and Allen 1995; Raab 1999] compute the arrangement of swept polyhedral surfaces based on their coarse approximation, [Baek et al. 2000] study a simple rotational sweep of exact SV. However, these 3D algorithms are either restricted to simple geometric primitives [Raab 1999] or simple sweep trajectory [Baek et al. 2000], or suffer from accuracy [Schroeder et al. 1994] and robustness problems [Abrams and Allen 1995].

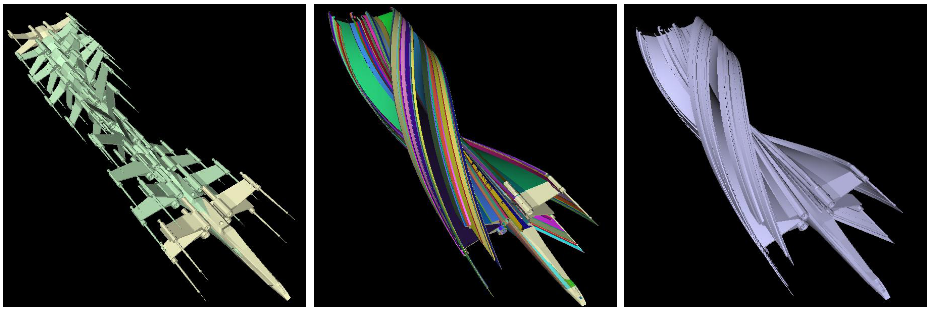
Visualization Many algorithms have been proposed to visualize the boundary of the SV using the rasterization hardware. These algorithms use the Z-buffer hardware to compute a 2D projection of the surface from a given viewpoint and not the actual boundary of the 3D SV. [Van Hook 1986; Huang and Oliver 1994; Hui 1994; Wang and Wang 1986] utilizes rasterization hardware to simulate NC machining display, [Conkey and Joy 2000] uses the Jacobian rank deficiency method to visualize a SV of trivariate tensor-product B-spline solids, and [Winter and Chen 2002] studies the SV computation of a 2D image.

2.2 Arrangement Computation

Given a finite collection of geometric objects in R^d, their arrangement is the decomposition of R^d into connected open cells [Halperin 1997]. The arrangement computation problem is ubiquitous by nature and arises in a number of applications. A survey of different algorithms and complexity bounds for arrangements computations is given in [Halperin 1997].

Complexity It is well known that the worst case combinatorial complexity of an arrangement of n surfaces in R^d is $O(n^d)$ [Halperin 1997], and there are such arrangements having $\theta(n^d)$ complexity, thus this bound is tight. In this analysis, each surface is assumed to have a bounded algebraic degree, and needs to be decomposed into *monotonic* patches as well.

Algorithms There are quite a few known algorithms to compute an arrangement using both deterministic algorithms and randomized algorithms. This includes an output-sensitive algorithm to compute an arrangement of surfaces



(a) Trajectory (b) Surfaces (c) $\partial SV(\mathbf{\Gamma})$

Figure 1: Complexity of SV Computation. *(a) shows a sweeping trajectory of a cubic polynomial curve for a X-Wing model. In (b), each surface primitive comprising in $\partial SV(\mathbf{\Gamma})$ (total 3793 surface primitives) is color-coded differently. (c) shows $\partial SV(\mathbf{\Gamma})$, an outer boundary of the surface elements.*

in 3-space and has $O(n\lambda_q(n)\log(n) + V\log(n))$ time complexity, where V is the combinatorial complexity of the vertical decomposition, q is a constant depending on the degree of the surfaces, and $\lambda_q(n)$ is the maximum length of (n, q) Davenport-Schinzel sequences [de Berg et al. 1996].

Implementation Issues Some of the major issues in the implementation of arrangement computation algorithms are accuracy and robustness problems. It is quite hard to enumerate all degenerate configurations, especially when the primitives are non-linear surfaces. [Raab 1999] enumerate 15 different possible degenerate cases for an arrangement of polyhedral surfaces. Moreover, [Raab 1999] proposed a controlled perturbation scheme, and applied it to polyhedral SV approximation. However, it can take a considerable amount of time for models composed of few hundred triangles. These problems get more severe when we are dealing with curved primitives.

2.3 Distance Field Computation and Iso-Surface Reconstruction

Recently, distance fields have been increasingly used in volumetric shape representation [Gibson 1998; Frisken et al. 2000], proximity computations based on rasterization hardware [Hoff et al. 2001], path planning [Kimmel et al. 1998], surface metamorphosis [Cohen-Or et al. 1998], and SV computation [Schroeder et al. 1994].

Grid-based iso-surface reconstruction has been extensively studied beginning from the seminal work of the Marching Cubes algorithm [Lorensen and Cline 1987], and has been extended to its variants such as the Enhanced Marching Cubes (EMC) [Kobbelt et al. 2001] or the dual contouring method [Ju et al. 2002]. [Wood et al. 2000] have used surface wavefront propagation techniques to extract semi-regular meshes from volumes.

3 Overview

In this section, we characterize the mathematical formulation of computing the SV of general polyhedral models and also give an overview of our approximation scheme.

3.1 Notation

We use bold-faced letters to distinguish a vector (e.g. $\boldsymbol{p}(t)$) from a scalar value (e.g. time t). f, v, e respectively denotes a face, a vertex, and an edge of a polyhedron. We use $f_k^{\mathbf{\Gamma}}$ to denote the kth face of a polyhedron $\mathbf{\Gamma}$.

3.2 Problem Formulation

Let $\mathbf{\Gamma}$, also known as a generator, be a polyhedron in R^3. Let the sweep trajectory $\boldsymbol{\tau}(t)$ be a tuple of $(\mathbf{\Psi}(t), \boldsymbol{R}(t))$, where $\mathbf{\Psi}(t)$ is a time-varying, differentiable vector in R^3 and $\boldsymbol{R}(t)$ is a time-varying, orthonormal matrix in $SO(3)$. Here, both $\mathbf{\Psi}(t)$ and $\boldsymbol{R}(t)$ depend on a single variable, the time $t \in [0, 1]$. Furthermore, $\mathbf{\Psi}(0)$ corresponds to the origin, and $\boldsymbol{R}(0)$ to the identity matrix. Then, consider the following sweep equation of $\mathbf{\Gamma}(t)$:

$$\mathbf{\Gamma}(t) = \mathbf{\Psi}(t) + \boldsymbol{R}(t)\mathbf{\Gamma} \qquad (1)$$

In our paper, the SV of the generator $\mathbf{\Gamma}$ along the trajectory $\boldsymbol{\tau}(\boldsymbol{t})$ is defined as:

$$SV(\mathbf{\Gamma}) = \{ \cup \mathbf{\Gamma}(t) \mid t \in [0, 1] \} \qquad (2)$$

Notice that our SV equation is allowed with only rigid motions (i.e., translation and rotation), even though, in general, $\boldsymbol{\tau}(t)$ can be any isotopy mapping [Weld and Leu 1990].

Our goal is to compute the boundary of $SV(\mathbf{\Gamma})$, $\partial SV(\mathbf{\Gamma})$, without internal voids. More formally, consider an arrangement $\mathcal{A}$ and a cell $\mathcal{C}$ in $\mathcal{A}$, which is reachable from infinity following some continuous path. Let us further define the *outer boundary* of $\mathcal{A}$ as the boundary of $\mathcal{C}$. Then, we want to compute the outer boundary[1] of $\mathcal{A}$ induced by the surface elements in $SV(\mathbf{\Gamma})$. We use the following theorems [Weld and Leu 1990] to characterize the boundary of SV:

THEOREM 3.1 *If during the sweep* $\mathbf{\Gamma}(\boldsymbol{t_i}) \cap \mathbf{\Gamma}(\boldsymbol{t_j}) = \phi$ *for* $t_i \neq t_j$*, then* $SV(\mathbf{\Gamma}) = \{ \cup_{k=1}^{n} SV(f_k^{\mathbf{\Gamma}}) \mid$ *n is the number of faces in* Γ *}.*

THEOREM 3.2 $SV(f_k^{\mathbf{\Gamma}})$ *consists of:*

[1] Throughout the paper, we interchangeably use the outer boundary of $\mathcal{A}$ and the outer boundary of $SV(\mathbf{\Gamma})$ to describe $\partial SV(\mathbf{\Gamma})$.

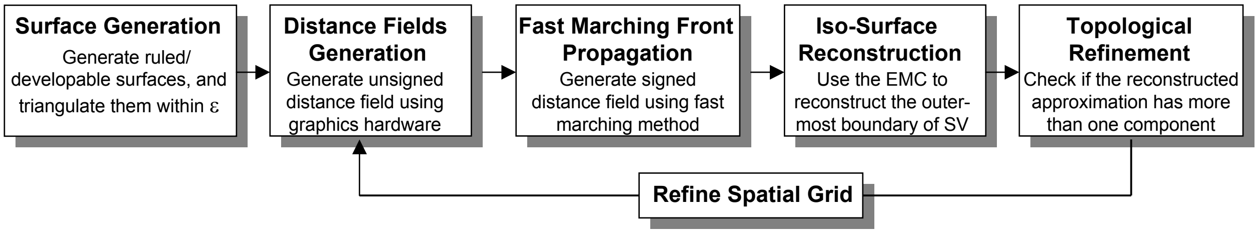

Surface Generation
Generate ruled/ developable surfaces, and triangulate them within ε
Distance Fields Generation
Generate unsigned distance field using graphics hardware
Fast Marching Front Propagation
Generate signed distance field using fast marching method
Iso-Surface Reconstruction
Use the EMC to reconstruct the outer-most boundary of SV
Topological Refinement
Check if the reconstructed approximation has more than one component
Refine Spatial Grid

Figure 2: Our Swept Volume Computation Pipeline

- *Faces in $f_k^{\mathbf{\Gamma(0)}}$ and $f_k^{\mathbf{\Gamma(1)}}$*

- *Ruled surfaces using the edges of $f_k^{\mathbf{\Gamma}(t)}$ as a ruling line along the directrix $\boldsymbol{\tau}$*

- *Developable surfaces as an envelope of $f_k^{\mathbf{\Gamma}(t)}$ along $\boldsymbol{\tau}$.*

Therefore, computing the boundary of $\partial SV(\mathbf{\Gamma})$ boils down to computing ruled and developable surface primitives, and finally computing the outer boundary of their arrangement.

3.3 Approximation Algorithm

Our goal is to compute the outermost boundary, $\partial SV(\mathbf{\Gamma})$ where the complexity of $\mathbf{\Gamma}$ is relatively high, e.g. thousands of triangles. The major difficulty of the computation lies in the arrangement computation, as its computational and combinatorial complexity can be super-quadratic and its implementation is rather non-trivial due to the accuracy and robustness problems. Given the complexity of surface-surface intersection problem, it is very hard to robustly compute all the intersections between thousands of ruled and developable surface primitives within a reasonable time. For example, in Fig. 1, in order to exactly compute the SV of the X-Wing model consisting of 2496 triangles, we need to compute an arrangement of 3793 surfaces including calculating their intersection curves of as high as degree nine. Thus, instead of computing $\partial SV(\mathbf{\Gamma})$ exactly, we approximate it using an implicit modeling technique based on discretized representations and iso-surface-based reconstruction methods.

The main idea of our approximation approach is to compute the polyhedral approximation of ruled and developable surface primitives, generate their signed distance field, and reconstruct the outer boundary of the arrangement of the discretized surfaces. To accelerate this pipeline, we prune redundant surfaces in $SV(f_k^{\mathbf{\Gamma}})$, and perform fast distance field computation. As a result, the basic steps of our algorithm are as follows:

1. Given an error threshold of Hausdorff distance ϵ, we formulate the ruled and developable surfaces for each $SV(f_k^{\mathbf{\Gamma}})$, and compute a triangular approximation that is within the surface deviation error threshold. A subset of the primitives $SV(f_k^{\mathbf{\Gamma}})$ that do not contribute to the final boundary, $\partial SV(\mathbf{\Gamma})$ can be pruned away using sufficient criteria described in Sec. 4.3.

2. We compute the directed unsigned distance fields for each $SV(f_k^{\mathbf{\Gamma}})$ on a uniform 3D grid, using interpolation-based rasterization hardware.

3. We use a variant of the fast marching level set method to classify all the grid points whether they are inside or outside with respect to $\partial SV(\mathbf{\Gamma})$. This gives us a signed distance field.

4. Perform the iso-surface extraction on the resulting signed distance field to reconstruct the outermost boundary, $\partial SV(\mathbf{\Gamma})$

5. Perform a topological check to see if the reconstructed approximation has more than one component. If yes, we refine the spatial grid and perform the steps 2-5 again.

The above pipeline is illustrated in Fig. 2.

4 Surface Generation

In this section, we present techniques to compute the candidate surface primitives that contribute to the boundary of SV and compute a bounded error triangulation of each primitive. We also present new techniques to cull away surface primitives that do not compute the outer boundary of SV.

4.1 Boundary Surfaces

As shown in Thm. 3.1 in Sec. 3.2, the boundary of SV is obtained by computing the SV's of individual faces, $SV(f_k^{\mathbf{\Gamma}})$, in $\mathbf{\Gamma}$, and computing their union. Moreover, Thm. 3.2 states that, besides the trivial surfaces of $f_k^{\mathbf{\Gamma}}$ at initial and final positions during sweep (i.e., $\mathbf{\Gamma(0)}$ and $\mathbf{\Gamma(1)}$ in Eq. 1), there are only two types of surfaces that belong to $f_k^{\mathbf{\Gamma}}$: ruled and developable surfaces (also see Fig. 3).

4.1.1 Ruled Surface Primitives

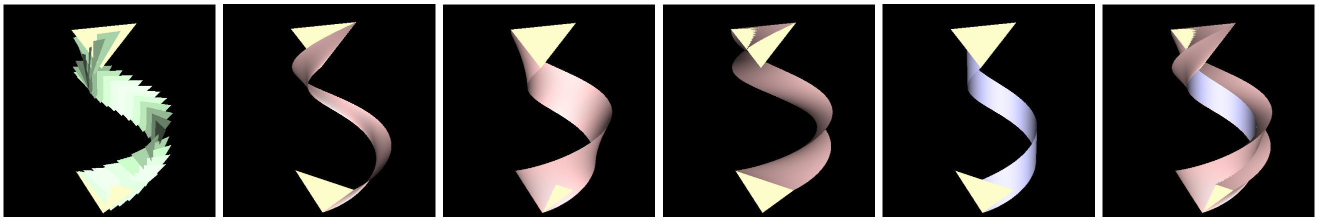
A ruled surface is generated by sweeping a ruling line along a directrix curve. The surface $\boldsymbol{x}(u,v)$ has the following form:

$$\boldsymbol{x}(u,v) = \boldsymbol{b}(u) + v\boldsymbol{\delta}(u) \tag{3}$$

Here, $\boldsymbol{b}(u)$ is a directrix and $\boldsymbol{\delta}(u)$ is the direction of a ruling line. When we sweep $f_k^{\mathbf{\Gamma}}$ along the trajectory $\boldsymbol{\tau}(t)$, each edge e in $f_k^{\mathbf{\Gamma}}$ generates a ruled surface $\boldsymbol{x}(u,v)$. We denote the endpoints of an edge $\boldsymbol{e}$ by $\boldsymbol{p_0}$ and $\boldsymbol{p_1}$. By substituting $\boldsymbol{p_0}$ and $\boldsymbol{p_1}$ for $\mathbf{\Gamma}$ in Eq. 1, we generate two curves, $\boldsymbol{b_0}(u)$ and $\boldsymbol{b_1}(u)$. Then, in Eq. 3, $\boldsymbol{b}(u)$ becomes $\boldsymbol{b_0}(u)$, and $\boldsymbol{\delta}(u)$ becomes $\boldsymbol{b_1}(u) - \boldsymbol{b_0}(u)$.

4.1.2 Developable Surface Primitives

When a plane moves continuously along a trajectory $\boldsymbol{\tau}(t)$, its envelope generates a developable surface. Intuitively, a developable surface is a surface which can be made of a piece of paper [Pottmann and Wallner 2001]. Thus, a developable surface is locally isometric to a plane, and its Gaussian curvature at regular points is zero. Furthermore, a developable surface is a subset of a ruled surface.



(a) Trajectory (b) Ruled 1 (c) Ruled 2 (d) Ruled 3 (e) Developable (f) SV

Figure 3: Boundary Surfaces of the SV of a Triangle. *(a) shows a trajectory for the helical sweep of a yellow triangle. (b), (c), and (d) show ruled surfaces generated by the sweep, and (e) shows a developable surfaces by the sweep. (f) shows the final boundary surface of the sweep.*

Let us parametrically represent a moving plane $\boldsymbol{p}(u,v,t)$ as:

$$\boldsymbol{p}(u,v,t) = \boldsymbol{q}(t) + u\boldsymbol{r_1}(t) + v\boldsymbol{r_2}(t) \tag{4}$$

where $\boldsymbol{q}(t)$ is the origin of $\boldsymbol{p}(u,v,t)$, and $\boldsymbol{r_1}(t)$ and $\boldsymbol{r_2}(t)$ are two linearly independent vectors spanning $\boldsymbol{p}(u,v,t)$. Then, using the envelope theory, by solving $det(J(\boldsymbol{p}(u,v,t))) = 0$ for u and substituting the result for $\boldsymbol{p}(u,v,t) = 0$, we get the developable surface $\boldsymbol{d}(t,v)$ as [Weld and Leu 1990]:

$$\begin{aligned} \boldsymbol{d}(t,v) &= \boldsymbol{b}(t) + v\boldsymbol{\delta}(t), \quad \text{where} \qquad (5)\\ \boldsymbol{b}(t) &= \boldsymbol{q}(t) - \boldsymbol{r_1}(t)\frac{\boldsymbol{q'}(t)\cdot\boldsymbol{r_1}(t)\times\boldsymbol{r_2}(t)}{\boldsymbol{r'_1}(t)\cdot\boldsymbol{r_1}(t)\times\boldsymbol{r_2}(t)} \\ \boldsymbol{\delta}(t) &= \boldsymbol{r_2}(t) - \boldsymbol{r_1}(t)\frac{\boldsymbol{r'_2}(t)\cdot\boldsymbol{r_1}(t)\times\boldsymbol{r_2}(t)}{\boldsymbol{r'_1}(t)\cdot\boldsymbol{r_1}(t)\times\boldsymbol{r_2}(t)} \end{aligned}$$

This derivation is valid only if $\boldsymbol{r'_1}(t)\cdot\boldsymbol{r_1}(t)\times\boldsymbol{r_2}(t) \neq 0$. Otherwise, we can derive a similar equation in terms of u and t by getting rid of v in Eq. 4.

In the SV computation, sweeping $f_k^{\boldsymbol{\Gamma}}$ also generates a developable surface. Let us assume that $\boldsymbol{\Gamma}$ is triangulated, and denote any two edges of $f_k^{\boldsymbol{\Gamma}}$ by $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$. Then, the direction vectors of $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ become $\boldsymbol{r_1}(t)$ and $\boldsymbol{r_2}(t)$ in Eq. 5. However, since Eq. 5 is derived from a plane, not from a triangle, the developable surface obtained from Eq. 5 needs to be clipped against the parametric domain of $\{u = 0, 0 \leq v \leq 1\}$, $\{0 \leq u \leq 1, v = 0\}$, and $\{u \geq 0, v \geq 0, u+v = 1\}$ for all t.

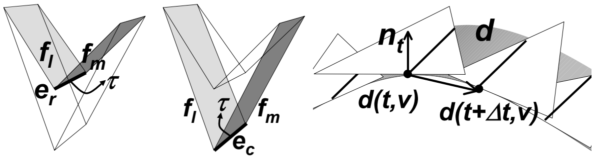
4.2 Bounded Error Triangulation

Once we have generated parametric representations for ruled and developable surface primitives, the next step is to compute a triangular approximation within a user-provided error deviation ϵ. There are many known algorithms for triangulating a rational parametric surface using either uniform [Kumar and Manocha 1995] or adaptive tessellation [Velho et al. 1999; Chung and Field 2000]. Since developable surfaces have zero Gaussian curvature, the uniform tessellation serves the purpose well; however, depending on the sweep trajectory $\boldsymbol{\tau}(t)$, the ruled surface can have regions of high curvature. In this case, the uniform tessellation tend to oversample the surface, so that the adaptive tessellation is more suitable. Notice that, depending on the chosen type of the trajectory $\boldsymbol{\tau}$, the ruled and developable surfaces can be well-known rational parametric surfaces or general parametric surfaces including trigonometric terms. However, since we can always perform a *flat-ness* test for smooth surface patches on the ruled and developable surfaces, we use a simple recursive algorithm like [Chung and Field 2000] to handle the general parametric surfaces as long as they are smooth surfaces.

On the other hand, taking advantage of the nature of line geometry in ruled surfaces, one can also devise a *variational interpolatory subdivision* scheme for ruled surfaces [Pottmann and Wallner 2001]. Here, one recursively subdivides a ruled surface by minimizing an discrete energy functional that is represented in terms of an discrete approximation of mean curvatures at points on the surface.

4.3 Culling Surface Primitive

In principle, assuming that the input model $\boldsymbol{\Gamma}$ is triangulated, each $SV(f_k^{\boldsymbol{\Gamma}})$ generates three ruled surface primitives, one developable surface, and $f_k^{\boldsymbol{\Gamma}}$'s at the initial and final positions of $\boldsymbol{\tau}(t)$. Therefore, the triangle counts of the ruled and developable surfaces significantly affect the performance of the pipeline presented in Fig. 2. Consequently, we want to identify portions of surface primitives that do not contribute to $\partial SV(\boldsymbol{\Gamma})$, and prune them away accordingly. We use a variation of a technique presented in [Abrams and Allen 2000] to cull away redundant ruled surface primitives, and also provide a novel method for developable surface primitives.



(a) (b) (c)

Figure 4: Surface Culling. *(a) A reflex edge e_r is not needed to generate a ruled surface along the trajectory $\boldsymbol{\tau}$, because the surface will be subsequently subsumed by the SV of its adjacent faces, $SV(f_l)$ or $SV(f_m)$. (b) A convex edge e_c does not need to produce a ruled surface when it is swept inside of its adjacent faces (f_l and f_m) along $\boldsymbol{\tau}$, because it will be subsumed by $SV(f_l)$ or $SV(f_m)$. (c) A developable surface $\boldsymbol{d}$ does not need to be created when it exists inside its generator triangle. This is checked by the angle between the normal n_t and the difference vector $\boldsymbol{d}(t+\Delta t,v) - \boldsymbol{d}(t,v)$* between successive time-steps.

In order to prune ruled surface primitives, we perform the following operation. First of all, a *reflex edge* e_r in $\boldsymbol{\Gamma}$ is not used to generate a ruled surface at all, since the surface will be always subsumed by the SV of the adjacent faces of e_r (also see Fig. 4-(a)). The same reasoning is applied to a *coplanar edge*, whose adjacent faces are coplanar. Furthermore, if a *convex edge* e_c instantaneously moves inward $f_l^{\mathbf{\Gamma}}$ and $f_m^{\mathbf{\Gamma}}$ at time t, where $f_l^{\mathbf{\Gamma}}$ and $f_m^{\mathbf{\Gamma}}$ are the adjacent faces of e_c, then e_c can stop generating a ruled surface at that time, since that portion will be also subsumed by $SV(f_l^{\mathbf{\Gamma}})$ or $SV(f_m^{\mathbf{\Gamma}})$ (also see Fig. 4-(b)). This test can be easily worked out by checking the velocity vectors $\boldsymbol{\tau}'(t)$ at the endpoints of e_c against the face normals of $f_l^{\mathbf{\Gamma}}$ and $f_m^{\mathbf{\Gamma}}$.

We also present a novel culling scheme for developable surface primitives. The main idea is that we generate a developable surface $\boldsymbol{d}_{f_k^{\mathbf{\Gamma}}}(t, v)$ only if its boundary can be exposed outside of its generating face $f_k^{\mathbf{\Gamma}}$. Since a developable surface $\boldsymbol{d}_{f_k^{\mathbf{\Gamma}}}(t, v)$ is locally convex [de Carmo 1976] and $f_k^{\mathbf{\Gamma}}$ is always tangent to $\boldsymbol{d}_{f_k^{\mathbf{\Gamma}}}(t, v)$, $\boldsymbol{d}_{f_k^{\mathbf{\Gamma}}}(t, v)$ locally lies inside or outside of $f_k^{\mathbf{\Gamma}}$ depending on the face normal $\boldsymbol{n}_t$ of $f_k^{\mathbf{\Gamma}}$. More specifically, since we perform uniform tessellation of a developable surface using some fixed time step Δt, we approximate the locality with Δt. Then, we compute two points $\boldsymbol{d}_{f_k^{\mathbf{\Gamma}}}(t, v)$ and $\boldsymbol{d}_{f_k^{\mathbf{\Gamma}}}(t + \Delta t, v)$ from Eq. 5, and check the angle between the difference vector $\boldsymbol{d}_{f_k^{\mathbf{\Gamma}}}(t+\Delta t, v) - \boldsymbol{d}_{f_k^{\mathbf{\Gamma}}}(t, v)$ and the plane normal $\boldsymbol{n}_t$ of $f_k^{\mathbf{\Gamma}}$. If it is less than 90 degrees, $f_k^{\mathbf{\Gamma}}$ is used during time t, otherwise it is pruned away (also see Fig. 4-(c)).

5 Sampling and Reconstruction

Once we have computed all the surface primitives of SV, we approximate the outer boundary of SV by sampling the surfaces and reconstructing the outer boundary of their arrangement. In this section, we describe the sampling and reconstruction pipeline (see Fig. 2). We compute an unsigned distance field with respect to the surface primitives on a discrete spatial grid. A signed distance field is obtained by propagating a front around the boundary of the swept volume using a fast marching method. An iso-surface extraction from this signed distance field provides us with an initial approximation to the outer boundary. We perform a topological connectedness test on this approximation. If the test fails, we refine the spatial grid, recompute the distance field, and repeat the pipeline.

5.1 Distance Field Representation

Given all the surface primitives of SV, we first discretize the 3D space occupied by the primitives. As a discrete representation of the 3D space, we choose signed distance fields with respect to the surface primitives, and attempt to compute them efficiently using graphics hardware. This discrete representation is used later in iso-surface extraction.

We sample the distance values at the discrete points of a 3D spatial grid, and use an enhanced representation of the discrete distance field. Here, the distance value at each grid point means the closest distance to one of the surface primitives. However, in our scheme, instead of simply using a scalar distance value for each grid point, we store directed distances along six principal directions corresponding to $x-$, $x+$, $y-$, $y+$, $z-$ and $z+$ axes. Our goal is to evaluate the directed distance function at the grid points of a 3D uniform grid. We would like to use an approach that maps well to SIMD-like capabilities of rasterization hardware. Current graphics processors have the capability to evaluate the distance function in parallel for each pixel on the plane. Graphics hardware-based fast techniques have been used for distance field evaluation [Hoff et al. 1999].

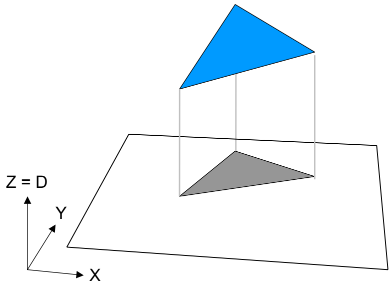


Figure 5: Directed distance field: *This figure shows how a slice of the directed distance field of a primitive (blue triangle) is computed. The primitive is rendered under orthographic projection with the slice (black rectangle) set as the image plane. The Z-buffer holds the directed distance values. The grey triangle is the projection of the primitive onto the slice.*

We employ a modified approach, and also use graphics hardware to generate the directed distance fields as follows:

1. Our algorithm computes the directed distance along a given direction by sweeping a plane along that direction. This plane corresponds to a slice of the directed distance field and is perpendicular to the direction of sweep.

2. Our algorithm computes the directed distance field one slice at a time. So the problem is reduced to defining the directed distance function of a primitive over a planar 2D slice. The main idea of our approach is that in order to obtain an approximation to the primitive's directed distance function, we simply render the primitive under orthographic projection with the slice as the image plane (see Fig. 5).

3. At each step, the slice is moved by a distance equal to the size of the grid cell. The planes corresponding to two consecutive slices are used to define a slab.

4. For each slab, we precompute the set of surface primitives that it intersects with.

5. We use orthographic projection to sample and rasterize the surfaces. The above slab is set as the near and far clipping planes. We render the surface primitives associated with the slab. Each pixel in the frame buffer corresponds to a point in the current slice and the depth buffer holds the value of the distance at that point.

6. We readback the depth buffer and store the directed distance values. Distances with absolute values larger than grid edge length are irrelevant since they are not used during isosurface extraction.

Our algorithm computes only an unsigned directed distance field. However, the isosurface extraction algorithm requires a signed distance field; i.e., a distinction needs to be made between inside and outside.

5.2 Fast Marching Front Propagation

We perform an inside/outside classification at each grid point to obtain a signed distance field. Conventionally, points that lie outside the boundary of the SV have a positive sign while those inside have a negative sign. Our surface primitives are in general not closed. As a result, we cannot define an inside/outside classification with respect to the individual surface primitives. We need to define a classification with respect to the boundary of the SV. However, this classification problem is non-trivial because we do not know the boundary of the SV.

In order to solve the inside and outside classification problem, we present a variant of the fast marching level set method [Sethian 1996] to propagate a front around the boundary of the swept volume. Level-set methods are numerical techniques for computing the position of propagating fronts. Topological changes are naturally captured in this setting. We perform the front propagation on the discrete spatial grid (see Fig. 6). We use the unsigned directed distance field generated in Sec. 5.1 for front propagation. The front consists of a set of grid points. We can initialize the front to be a set of grid points corresponding to any surface bounding the SV. One choice for the initial front is the set of grid points that lie along the boundary of the spatial grid. Our front propagation method ensures that the front visits exactly those grid points that lie outside the swept volume.

We tag grid points as *Known*, *Trial*, or *Far* depending on whether the grid point has already been visited, is currently being visited, or is yet to be visited by the front, respectively. Each grid point also has a flag whose value can be *Inside* or *Outside*. Initially all grid points are assigned a flag, *Inside*. All grid points except the initial front are tagged as *Far*. Grid points on the initial front are tagged as *Trial*. During one step of front propagation, we perform a number of operations. These include:

1. We arbitrarily pick a *Trial* grid point belonging to the front and remove it from the front. Let this point be denoted as P. We set its tag to be *Known*.

2. Consider a neighboring grid point Q of P. If point Q is tagged as *Known*, we do not update it. With respect to P, point Q lies along one of the six principal directions. We check if the directed distance of P along that direction is larger than the length of edge connecting P and Q. If that is the case, we are guaranteed that point Q lies outside the boundary of the SV. Therefore we propagate the front to point Q by adding Q to the front. In addition, the flag for point Q is set to *Outside*.

The pseudo-code is shown in Alg. 5.1. The front propagation continues in this manner until the front has visited all grid points outside the SV. At this time, front propagation terminates. In this manner, we obtain an inside/outside classification for each grid point. We combine this inside/outside classification with the unsigned distance field computed in Sec. 5.1 to obtain a signed distance field. All six directed distances at a grid point always have the same sign.

5.3 Isosurface Extraction

We estimate the outer boundary of open surfaces by performing an isosurface extraction from the signed distance field generated using the approach described in Sec. 5.1 and Sec. 5.2. We use the Extended Marching Cubes (EMC) algorithm [Kobbelt et al. 2001] to perform the isosurface extraction. This algorithm can detect sharp features and sample them in order to reduce the aliasing artifacts. The output of the isosurface extraction is a polygonal mesh. This is our initial approximation to the outer boundary.

```
while front is nonempty
  Extract a trial point $P$ from the front
  $P$.tag = KNOWN
  for each neighbor $Q$ of $P$,
    if $Q$.tag != Known then
      $d$ = Direction from $P$ to $Q$
      if Directed_Distance($P$,$d$) > Edge_Length($P$,$Q$) then
        $Q$.flag = Outside
        if $Q$.tag == Far then
          Add $Q$ to the Front
          $Q$.tag = Trial
        endif
      endif
    endif
  endfor
endwhile
```

ALGORITHM 5.1: Fast Marching Method

In order to perform isosurface extraction, we need to know which edges of a cube of the spatial grid are intersected by the isosurface. An edge of a cube is intersecting if the two endpoints of the edge have different inside/outside classification. For each intersecting edge of a cube, the directed distances of the endpoints of the edge give us the position of the intersection point (see Fig. 7). The advantage of using directed distance is that it provides us with exact surface samples. The standard Marching Cubes algorithm can produce aliasing artifacts in the vicinity of sharp features. The Extended Marching Cubes algorithm uses a tangent element approximation to reduce aliasing artifacts and provide better reconstruction in the presence of sharp features. Thus we have a better approximation to the exact, outer boundary.

We only use the directed distance of the grid point which is *Outside*. The directed distance of the grid point which is *Inside* may result in incorrect intersection points (see Fig. 7).

5.4 Topological Refinement

The underlying topology induced by our SV approximation algorithm can be different from the topology of the exact SV. This mainly results from the sampling and reconstruction steps in our computational pipeline. However, our algorithm attempts to maintain some of the topological properties of SV, i.e. closed and connected boundary. According to our sweep equation in Eq. 1, the SV that we generate for a polyhedron can be a non-manifold. However, its boundary always provides a closed, water-tight surface, since the generator is a closed set. Moreover, the structure always generates one connected component.

To ensure a single connected component in our SV algorithm, we perform a topological check by traversing the generated polygonal mesh to detect the occurrence of such a case (see Fig. 8) . We arbitrarily pick a vertex from the mesh. We mark this vertex and recurse on each of the unmarked neighboring vertices. At the end of this traversal, if any unmarked vertices remain, it implies that the mesh has more than one component. In that case, we refine the spatial grid, recompute the distance field at a higher resolution, and perform the reconstruction again. We use EMC algorithm for the iso-surface reconstruction. This algorithm always generates a closed polygonal mesh structure. Therefore, our SV algorithm can guarantee closed, connected surface structures.

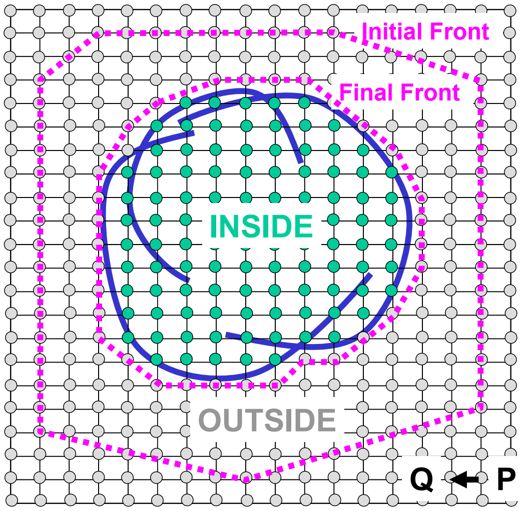


Figure 6: Fast Marching Method. *A fast marching level set method is used to propagate a front (pink dotted curve) around the surface primitives of SV (solid blue curves). All the grid points visited by the front (grey circles) lie outside the outer boundary while the remaining grid points (green circles) lie inside the outer boundary. During front propagation, a grid point P can update its neighboring grid point Q if the directed distance from P to Q is greater than the length of the edge connecting P and Q*

6 Analysis of Swept Volume Algorithm

In this section, we analyze the performance of our SV algorithm, and also discuss the sources of errors in the algorithm.

6.1 Performance Analysis

Our SV algorithm has the following computational complexity:

Surface Generation The computational cost of the surface generation is mainly determined by the surface tessellation process, and its complexity is sensitive to the output; i.e. $O(M)$, where M is the number of triangles generated by the tessellation. Let us denote N_e as the total number of convex edges of the input polyhedra $\mathbf{\Gamma}$, and N_f as the total number of faces of $\mathbf{\Gamma}$. Assuming that $\mathbf{\Gamma}$ is triangulated, in the worst case, $3N_e$ ruled surfaces and N_f developable surfaces are generated. Let us further denote $M_e^i(\epsilon, \boldsymbol{\tau})$ as the number of triangles generated by a ruled surface i, which depends on the given surface deviation error ϵ and the trajectory $\boldsymbol{\tau}$, and denote $M_f^j(\epsilon, \boldsymbol{\tau})$ as the number of triangles by a developable surface j. Let $M_e^+ = \max_{i=1}^{3N_e}(M_e^i(\epsilon, \boldsymbol{\tau}))$ and $M_f^+ = \max_{j=1}^{N_f}(M_f^j(\epsilon, \boldsymbol{\tau}))$. Then, $M = 3N_e M_e^+ + N_f M_f^+$. Typically, in our experiments, M_e^+ and M_f^+ correspond to a few hundred triangles.

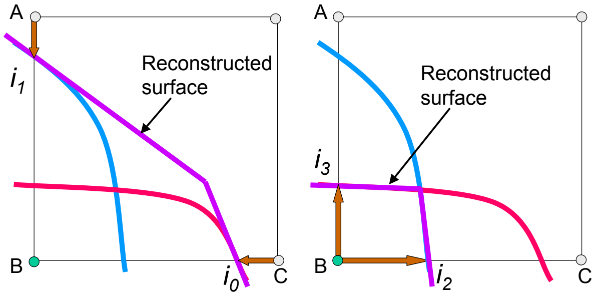


Figure 7: Iso-Surface Extraction: *Figure on the left shows the reconstructed surface (purple) for two surface primitives (blue and red). We use the directed distance (show in brown) to compute intersection points (i0 and i1). The grey and green circles respectively indicate grid points that lie Outside and Inside the outer boundary. Figure on the right shows that the directed distance of a Inside grid point (Point B) may result in incorrect intersection points (i2 and i3). We only use the directed distance of Outside grid points for reconstruction.*

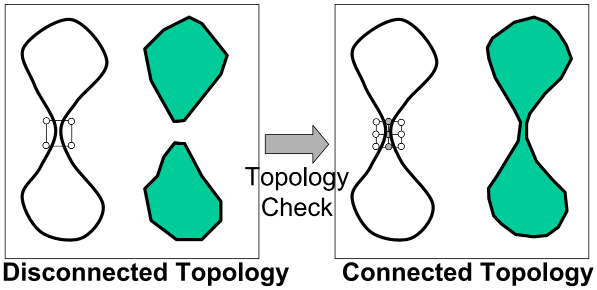


Figure 8: Topological Refinement. *We ensure that our SV approximation is a single connected component by performing topological refinement. We perform a topological check to see if our approximation has more than one component. In that case, we refine the spatial grid and perform the reconstruction again.*

Distance Field Generation Let D be the maximum dimension of the bounding box enclosing the surface primitives of SV. Then, let $K = D/\epsilon$, where ϵ is the given surface deviation error. We use a $K \times K \times K$ uniform spatial grid for generating the distance field, front propagation and isosurface extraction to restrict the reconstruction error within ϵ (see Sec. 6.2 for more detail). As stated earlier, we compute the distance field using graphics hardware. We can measure the time complexity of the distance field generation in terms of number of primitives sent to the graphics hardware. A primitive p_i gets rendered n_{p_i} times where N is the number of primitives and n_{p_i} is the number of slabs it occupies. Also the size of the spatial grid contributes to the time complexity. So the total time complexity is $O(K^3) + O(\sum_{i=1}^{N} n_{p_i})$. Typically, n_{p_i} is a small constant for most primitives. So the time to generate directed distance fields is typically linear in number of primitives.

Fast Marching Front Propagation Front propagation takes time proportional to the size of the spatial grid; i.e., $O(K^3)$

Isosurface Extraction Isosurface extraction takes time proportional to the size of the spatial grid; i.e., $O(K^3)$

Total Complexity The total computational complexity of our SV algorithm is $O(M + K^3 + \sum_{i=1}^{N} n_{p_i})$.

6.2 Error Analysis

Our SV algorithm is an approximation scheme. There are three main sources of the errors that govern the accuracy of the result of our algorithm; tessellation errors by approximating surface primitives, sampling errors from generating 3D grids of distance fields, and iso-surface reconstruction errors from the EMC.

Tessellation Errors We use adaptive tessellation to triangulate ruled surfaces, and uniform tessellation to triangulate developable surfaces. These methods can triangulate the surface primitives within an error threshold ϵ, which is essentially the Hausdorff distance between the original surfaces and approximated ones.

Sampling Errors The accuracy of the distance field is dependent on the implementation. We compute it using graphics hardware and its accuracy is determined by the number of bits of precision in the Z-buffer, typically 24 or 32 bits in current hardware.

Reconstruction Errors If ϵ is the size of the grid cell, we are guaranteed that each point on our reconstructed outer boundary lies within a distance ϵ of some point on the exact envelope. The approximation theory guarantees that a piecewise linear interpolant to a smooth surface converges with order $O(\epsilon^2)$ where ϵ measures the sampling density. In our case, ϵ is the size of the grid cell. In the presence of sharp features, however, the convergence rate drops to $O(\epsilon)$. However, the Extended Marching Cubes algorithm improves the local convergence rate by performing a tangent element approximation. This convergence rate is valid only in cells that have at most one sharp feature.

7 Implementation and Performance

In this section, we describe the implementation of our SV algorithm and highlight its performance on different benchmarks.

7.1 Implementation

To implement our SV algorithm, we used C++ programming language with the GNU g++ compiler under Linux operating system. For the choice of GUI implementation, GLUT, OpenGL, Inventor and Qt were used.

We used a public computational geometry library, CGAL, to perform an efficient traversal on the two-manifold polyhedral surfaces. Moreover, CGAL offers quite flexible data structures based on the usage of templates and STL programming, and also provide accurate evaluation of geometric predicates such as orientation test, cross product, dot product, etc [Fabri et al. 1996]. We took advantage of these benefits to implement the generation of surface primitives of SV. In particular, the *Polyhedron_3* class of CGAL was extensively used.

In order to compute the distances fields quickly and efficiently, we used nVidia's GeForce 4 GPU, which has 24 bit precision of accuracy in Z-buffer. With the availability of new GPU's such as ATI's Radeon9700, we can further improve this accuracy by using their floating point computational capability in the graphics pipeline.

7.2 Performance

We benchmarked our SV algorithm by using different models of varying complexities and with different sweeping trajectories. The complexity of our benchmarking models varies from 1,524 to 10,352 triangles. The model complexities are summarized from the second to the fifth column in Table 1. Furthermore, they consist of sharp edges and surface triangles with high aspect ratio. The sweeping paths that we chose are helical sweep (X-Wing and Swing-Clamps), translations using cubic rational functions (Input Clutch), and sinusoidal translations and rotations (the rest of the models). Therefore, most of our benchmarks perform sweep along very general trajectories. For the grid resolution in our benchmarks, we use the grid resolution $K = 128$ for all the models.

We performed timing analysis on a PC with Intel Xeon 2.4 GHz processor, 2GB of memory and nVidia GeForce 4 graphics card. The time spent during each stage in our SV computational pipeline is shown in different columns of Table 1. As the table shows, most of the time, typically more than 80% of the total computational time, is spent in the distance field generation stage of the pipeline. We observed that the distance field computation time was mainly spent on the readbacks between the framebuffer and main memory. Thus, as we increase the grid size K, the total computational time will increase linearly, since we perform the readbacks $O(K)$ times. We measured performance of our SV computation pipeline on the hammer benchmark at a grid resolution of $K = 256$. The distance field computation, front propagation and isosurface reconstruction took 41.4 s, 12.6 s and 8.9 s respectively.

In Fig. 9, we illustrate the results of the SV of the benchmarking models computed by our SV algorithms. In the figure, each row shows the generator polyhedral model $\mathbf{\Gamma}$, sweeping path $\boldsymbol{\tau}$, and the resulting SV approximation $\partial SV(\mathbf{\Gamma})$, respectively. All the rendered images in Fig. 9 are flat-shaded.

8 Comparison with Other Approaches

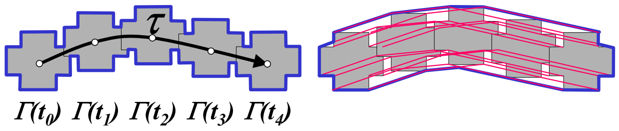
In this section, we compare the performance of our algorithm with earlier approximation schemes to estimate the boundary SV.

The algorithms presented in [Abrams and Allen 1995; Raab 1999] use a similar surface primitive generation and tessellation technique to enumerate the surface primitives of SV. However, they do not exploit the fact that developable surfaces can be precisely represented as a parametric surface using the envelope theory as provided in Eq. 5. As a result, we are able to derive better error bounds with our approximation scheme. More importantly, these algorithms perform exact computation of arrangements of polyhedral surfaces. This computation is prone to accuracy and degenerate cases [Abrams and Allen 1995] and the latter has been addressed in [Raab 1999] based on perturbation methods. However, their applications to complex models with long sweeping trajectories has been limited. Moreover, exact arrangement computation based on perturbation methods appears to take considerable amount of time. Compared to these approaches, our algorithm is less susceptible to robustness problems, provides a good error bound, and is readily extendible to complex models. However, exact computation of arrangements can produce a better approximation of sharp features on the boundary of the SV.

The algorithm described in [Schroeder et al. 1994] samples the sweeping trajectory and reduces the SV computation to computing the union of polyhedra corresponding to the discrete instances along the trajectory. It computes the union by generating distance-field based samples followed by an iso-surface reconstruction algorithm. Their formulation based on sampling the trajectory can at times lead to a coarse approximation of the SV. In particular, they need to use very high sampling rates along the sweeping path, otherwise iso-surface reconstruction may generate spikes or

Model	Combinatorial Complexity				Computational Performance (seconds)				
	$\mathbf{\Gamma}$	# of Surf	# of Surf Tri	$\partial SV(\mathbf{\Gamma})$	Surf Gen	Dist Field	Front Prop	Isosurface	Tot
X-wing	2496	3931	770K	307K	3.208	36.15	1.69	3.12	44.1
Air Cylinder	2280	1152	234K	249K	1.966	16.0	1.65	1.55	21.16
Swing Clamps	1524	1049	212K	126K	1.492	15.7	1.73	1.33	20.2
Hammer	1692	1390	281K	198K	1.822	16.1	1.59	1.97	21.4
Input Clutch	2116	1175	239K	129K	1.789	16.2	1.61	1.39	20.9
Pipe	10352	15554	803K	247K	4.038	59.2	1.61	2.48	67.2
Pivoting Arms	2158	1718	347K	162K	2.138	21.4	1.60	1.64	26.7

Table 1: Model Complexities of SV Benchmarks. *The first column shows the model names of the benchmarks. From the second to the fifth column, each column respectively shows the triangle count of the generator* $\mathbf{\Gamma}$*, the number of ruled and developable surfaces, the total number of triangles in the tessellated ruled and developable surfaces, and the triangle count of the boundary of* $\partial SV(\mathbf{\Gamma})$ *computed by our algorithm. From the sixth to the tenth column, each column respectively illustrates the timing for the surface primitive generation, distance field generation, inside/outside classification using fast marching propagation, iso-surface extraction using the EMC, and the total computation. We have chosen a grid resolution of* $K = 128$ *for all the benchmarks.*


τ
$\Gamma(t_0)$ $\Gamma(t_1)$ $\Gamma(t_2)$ $\Gamma(t_3)$ $\Gamma(t_4)$

(a) SV approximation computed by SLL94

(b) Our approximation

Figure 10: Comparison With Other Approach in 2D. *In (a), the SV algorithm approximates the blue line that is obtained by the discrete instances of the generator* $\mathbf{\Gamma}(t)$ *along the trajectory* $\boldsymbol{\tau}$*. Therefore, there can be many spiky features on the approximated SV surface. In (b), our SV algorithm approximates the blue line that is an outer boundary of the red lines, representing ruled and developable surfaces in 3D. The result is a more smooth surface.*

other high frequency features on the final approximation. For example, as shown Fig. 10-(a), the algorithm presented in [Schroeder et al. 1994] attempts to approximate an inherently spiky surface (thick blue line) as a result of discrete sampling on the trajectory, whereas our algorithm approximates a rather smooth surface which is an outer boundary (thick blue line) of ruled and developable surfaces (red line) as shown in Fig. 10-(b).

Overall, our approach generates a better error-bounded approximation as compared to [Schroeder et al. 1994] and results in a better convergence rate. Furthermore, the fast computation of directed distance fields using rasterization hardware considerable improves its running time.

9 Summary and Future Work

We present an efficient, fast algorithm to approximate SV of complex polyhedral models using the distance fields, fast marching propagation method, and iso-surface reconstruction. The algorithm has been benchmarked on a number of complex models with different sweep paths.

There are several areas for future work. We would like to look at adaptive subdivision schemes for better reconstruction [Varadhan et al. 2003]. The performance of our algorithm can be further improved by investigating more optimizations. These include more efficient surface generation based on incremental computations on sweeping path, possibility of using programmable graphics hardware to simulate the fast marching method, etc. We would like to further investigate the application of our SV algorithm to the areas such as collision detection, robot workspace analysis, and computer-aided geometric design. Finally, we will like to extend this algorithm to solids bounded by curved surfaces.

References

Abdel-Malek, K., and Othman, S. 1999. Multiple sweeping using the Denavit-Hartenber representation method. *Computer-Aided Design 31*, 567–583.

Abdel-Malek, K., and Yeh, H. J. 1997. Analytical boundary of the workspace for general 3-DOF mechanisms. *International Journal of Robotics Research 16*, 1–12.

Abdel-Malek, K., and Yeh, H. J. 1997. Geometric representation of the swept volume using the Jacobian rank deficiency conditions. *Computer-Aided Design 29*, 6, 457–468.

Abdel-Malek, K., and Yeh, H. J. 1997. On the determination of starting points for parametric surface intersections. *Computer-Aided Design 29*, 1, 21–35.

Abdel-Malek, K., Blackmore, D., and Joy, K. 2002. Swept volumes: Foundations, perspectives, and applications. *International Journal of Shape Modeling*. submitted.

Abdel-Malek, K., Yang, J., Brand, R., Vannier, M., and Tanbour, E. 2002. Towards understanding the workspace of human limbs. *International Journal of Ergonomics*. submitted.

Abrams, S., and Allen, P. 1995. Swept volumes and their use in viewpoint computation in robot work-cells. In *Proc. IEEE International Symposium on Assembly and Task Planning*, 188–193.

Abrams, S., and Allen, P. 2000. Computing swept volumes. *Journal of Visualization and Computer Animation 11*.

Ahn, J.-W., Kim, M.-S., and Lim, S.-B. 1993. Approximate general sweep boundary of a 2D curved objects. *Graphical Models and Image Processing 55*, 2, 98–128.

Baek, N., Shin, S., and Chwa, K. 2000. Three-dimensional topological sweep for computing rotational swept volumes of polyhedral objects. *Int'l J. of Computational Geometry and Applications 10*, 2.

Blackmore, D., and Leu, M. C. 1990. A differential equation approach to swept volumes. In *Proc. of Rensselaer's 2nd International Conference on Computer Integrated Manufacturing*, 143–149.

Blackmore, D., Leu, M., and Wang, L. 1997. Sweep-envelope differential equation algorithm and its application to NC machining verification. *Computer-Aided Design 29*, 629–637.

Boussac, S., and Crosnier, A. 1996. Swept volumes generated from deformable objects application to NC verification. In *Proceedings of International Conference on Robotics and Automation*, 1813–1818.

Chung, A. J., and Field, A. J. 2000. A simple recursive tessellator for adaptive surface triangulation. *Journal of graphics tools 5*, 3, 1–9.

Cohen-Or, D., Solomovic, A., and Levin, D. 1998. Three-dimensional distance field metamorphosis. *ACM Transactions on Graphics 17*, 2, 116–141.

Conkey, J., and Joy, K. 2000. Using isosurface methods for visualizing the envelope of a swept trivariate solid. In *Proc. of Pacific Graphics*.

de Berg, M., Guibas, L. J., and Halperin, D. 1996. Vertical decompositions for triangles in 3-space. *Discrete and Computational Geometry 15*, 36–61.

de Carmo, M. 1976. *Differential Geometry of Curves and Surfaces.* Prentice Hall, Englewood Cliffs, NJ.

Elber, G., and Kim, M.-S. 1999. Offsets, sweeps, and Minkowski sums. *Computer-Aided Design 31*, 3, 163.

Fabri, A., Giezeman, G.-J., Kettner, L., Schirra, S., and Schönherr, S. 1996. The CGAL kernel: A basis for geometric computation. In *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, Springer-Verlag, M. C. Lin and D. Manocha, Eds., vol. 1148, 191–202.

Frisken, S., Perry, R., Rockwood, A., and Jones, T. 2000. Adaptively sampled distance fields: A general representation of shapes for computer graphics. *Proc. of ACM SIGGRAPH*, 249–254.

Gibson, S. F. F. 1998. Using distance maps for accurate surface representation in sampled volumes. In *IEEE Symposium on Volume Visualization*, 23–30.

Halperin, D. 1997. Arrangements. In *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke, Eds. CRC Press LLC, Boca Raton, FL, ch. 21, 389–412.

Hoff, K., Culver, T., Keyser, J., Lin, M., and Manocha, D. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. *Proceedings of ACM SIGGRAPH*, 277–286.

Hoff, K., Zaferakis, A., Lin, M., and Manocha, D. 2001. Fast and simple geometric proximity queries using graphics hardware. *Proc. of ACM Symposium on Interactive 3D Graphics*.

Huang, Y., and Oliver, J. H. 1994. NC milling error assessment and tool path correction. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, ACM Press, A. Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, 287–294. ISBN 0-89791-667-0.

Hui, K. 1994. Solid sweeping in image space - application to NC simulation. *Visual Computer*.

Ju, T., Losasso, F., Schaefer, S., and Warren, J. 2002. Dual contouring of Hermite data. In *SIGGRAPH 2002, Computer Graphics Proceedings*.

Jüttler, B., and Wagner, M. 1996. Spatial rational B-spline motions. *ASME Journal of Mechanical Design 118*, 193–201.

Kieffer, J., and Litvin, F. 1990. Swept volume determination and interference detection for moving 3-D solids. *ASME Journal of Mechanical Design 113*, 456–463.

Kimmel, R., Kiryati, N., and Bruckstein, A., 1998. Multivalued distance maps for motion planning on surfaces with moving obstacles.

Kobbelt, L. P., Botsch, M., Schwanecke, U., and Seidel, H.-P. 2001. Feature-sensitive surface extraction from volume data. In *SIGGRAPH 2001, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH, E. Fiume, Ed., 57–66.

Kumar, S., and Manocha, D. 1995. Efficient rendering of trimmed NURBS surfaces. *Computer-Aided Design*, 509–521.

Law, C., Avila, S., and Schroeder, W. 1998. Application of path planning and visualization for industrial design and maintainability-analysis. In *Proc. of the 1998 Reliability and Maintainability Symposium*, 126–131.

Lee, J., Hong, S., and Kim, M. 2002. Polygonal boundary approximation for a 2D general sweep based on envelope and boolean operations. *Visual Computer 16*.

Lorensen, W. E., and Cline, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, vol. 21, 163–169.

Madrigal, C., and Joy, K. 1999. Boundary determination for trivariate solids. In *Proc. of the IASTED Intl Conf on Computer Graphics and Imaging*.

Martin, R., and Stephenson, P. 1990. Sweeping of three-dimensional objects. *Computer-Aided Design 22*, 4.

Pottmann, H., and Wallner, J. 2001. *Computational Line Geometry.* Springer.

Raab, S. 1999. Controlled perturbation for arrangements of polyhedral surfaces with application to swept volumes. In *Proc. 15th ACM Symposium on Computational Geometry*, 163–172.

Schroeder, W., Lorensen, W., and Linthicum, S. 1994. Implicit modeling of swept surfaces and volumes. In *IEEE Visualization Conference*.

Schwarzer, F., Saha, M., and Latombe, J.-C. 2002. Exact collision checking of robot paths. In *International Workshop on Algorithmic Foundations of Robotics*.

Sethian, J. 1996. A fast marching level set method for monotonically advancing fronts. In *Proc. Nat. Acad. Sci.*, vol. 93, 1591–1595.

Van Hook, T. 1986. Real-time shaded NC milling display. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, D. C. Evans and R. J. Athay, Eds., vol. 20, 15–20.

Varadhan, G., Krishnan, S., Kim, Y., and Manocha, D. 2003. Adaptive subdivision and reconstruction using box-distance fields. *Technical Report TR03-005, Department of Computer Science, University of North Carolina*.

Velho, L., de Figueiredo, L. H., and Gomes, J. 1999. A unified approach for hierarchical adaptive tessellation of surfaces. *ACM Transactions on Graphics 18*, 4, 329–360.

Wang, W., and Wang, K. 1986. Real-time verification of multi-axis NC programs with raster graphics. In *Proceedings of International Conference on Robotics and Automation*, 166–171.

Weld, J., and Leu, M. 1990. Geometric representation of swept volume with application to polyhedral objects. *International Journal of Robotics Research 9*, 5.

Winter, A., and Chen, M. 2002. Image-swept volumes. In *Proc. of Eurographics*.

Wood, Z., Desbrun, M., Schroeder, P., and Breen, D. 2000. Semi-regular mesh extraction from volumes. In *IEEE Visualization 2000 Proceedings*.

Xavier, P. 1997. Fast swept-volume distance for robust collision detection. In *Proceedings of International Conference on Robotics and Automation*.

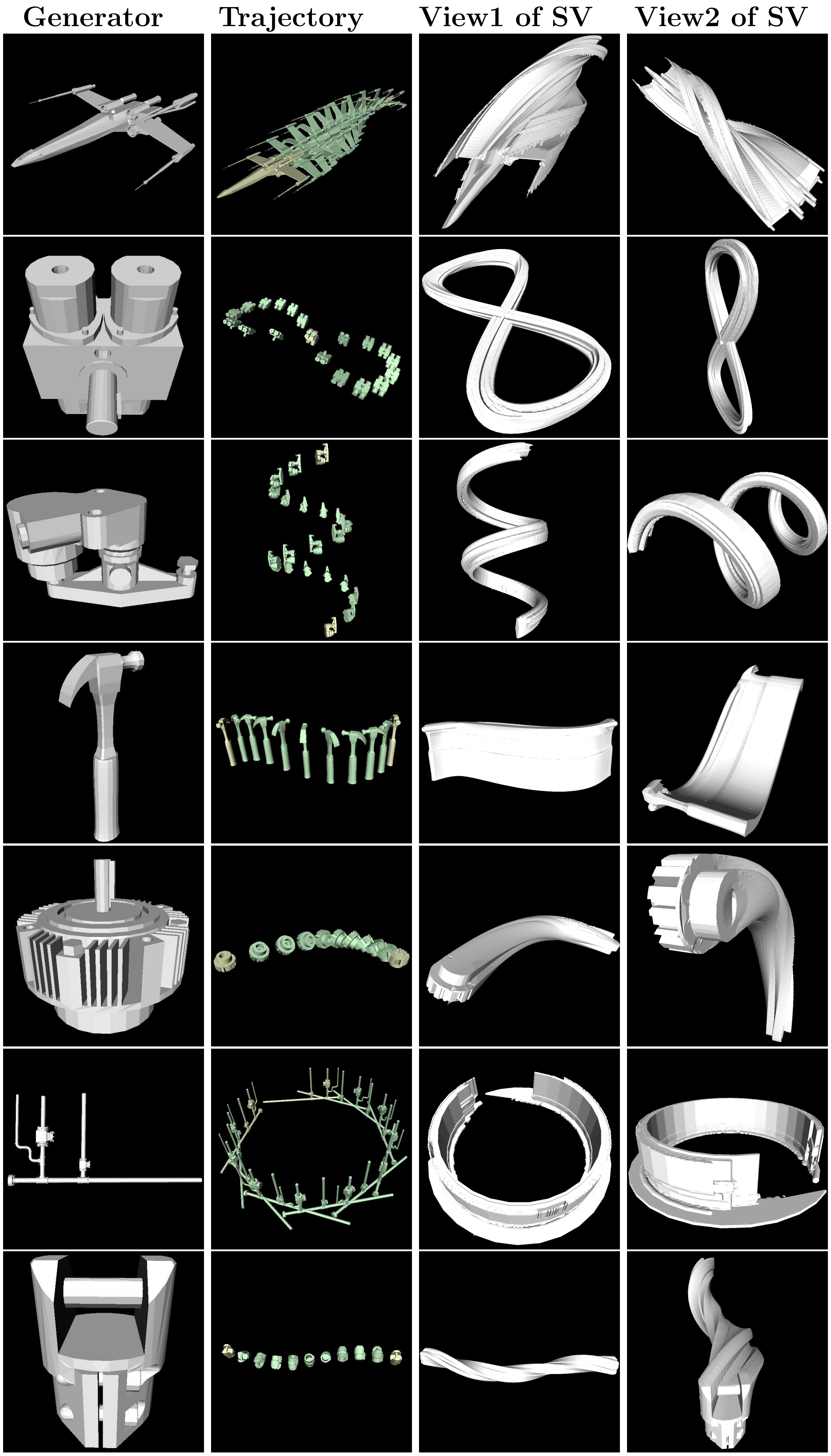
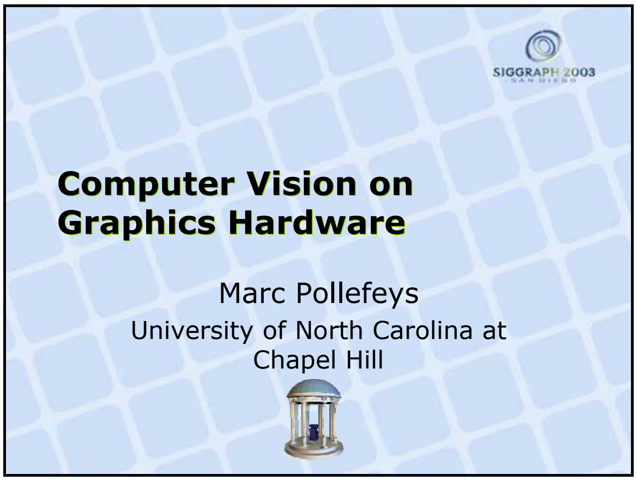


Figure 9: SV Benchmarks. *In each column, from left to right, each figure shows a generator model, sweeping trajectory, and two views of the resulting SV approximation reconstructed by our SV algorithm, respectively. In each row, each figure shows different benchmarking model, from top to bottom, X-Wing, Air Cylinder, Swing Clamps, Hammer, Input Clutch, Pipe, and Pivoting Arms, respectively.*

Computer Vision using Graphics Hardware

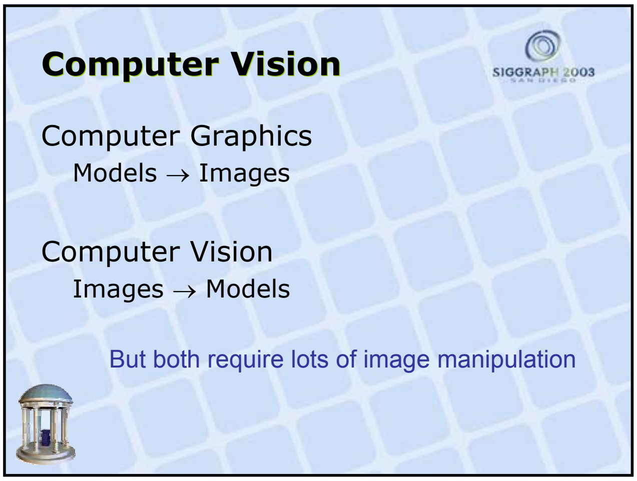
Marc Pollefeys
University of North Carolina at Chapel Hill



Computer Vision on Graphics Hardware

Marc Pollefeys

University of North Carolina at Chapel Hill



Computer Vision

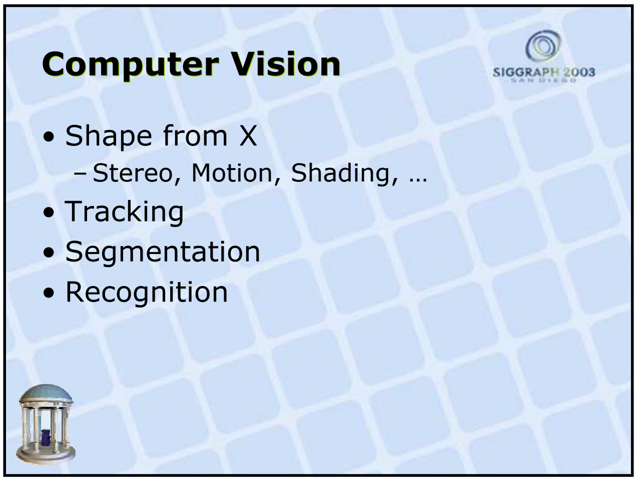
Computer Graphics

Models → Images

Computer Vision

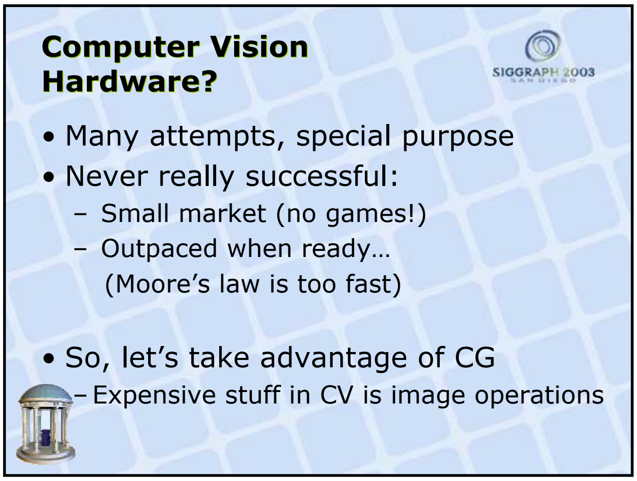
Images → Models

But both require lots of image manipulation



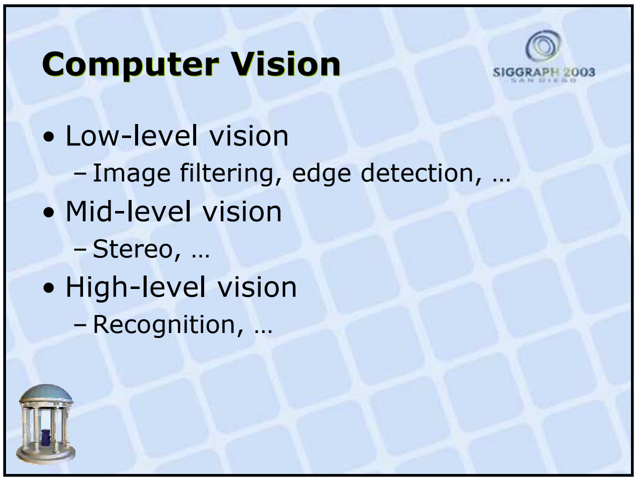
Computer Vision

- Shape from X
  - Stereo, Motion, Shading, …
- Tracking
- Segmentation
- Recognition



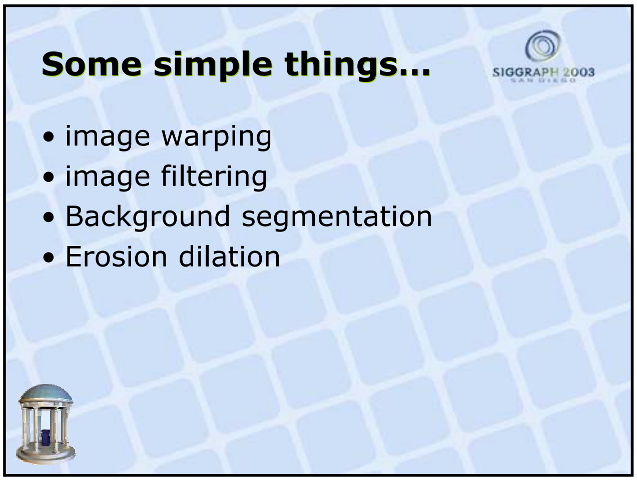
Computer Vision Hardware?

- Many attempts, special purpose
- Never really successful:
  - Small market (no games!)
  - Outpaced when ready…
    (Moore’s law is too fast)
- So, let’s take advantage of CG
  - Expensive stuff in CV is image operations



Computer Vision

- Low-level vision
  - Image filtering, edge detection, …
- Mid-level vision
  - Stereo, …
- High-level vision
  - Recognition, …



Some simple things…

- image warping
- image filtering
- Background segmentation
- Erosion dilation

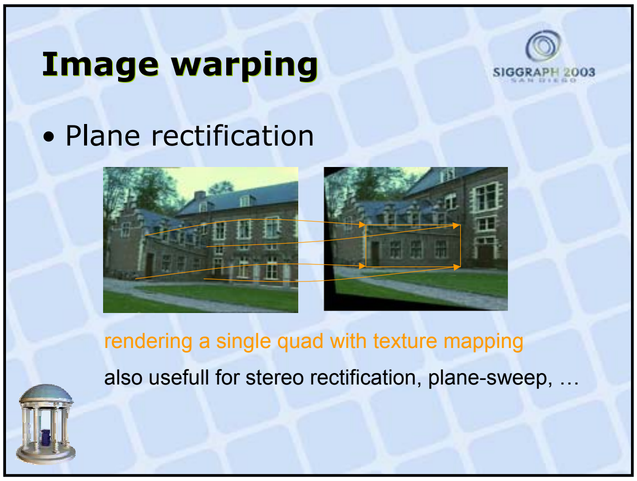


Image warping

- Plane rectification

rendering a single quad with texture mapping

also usefull for stereo rectification, plane-sweep, …

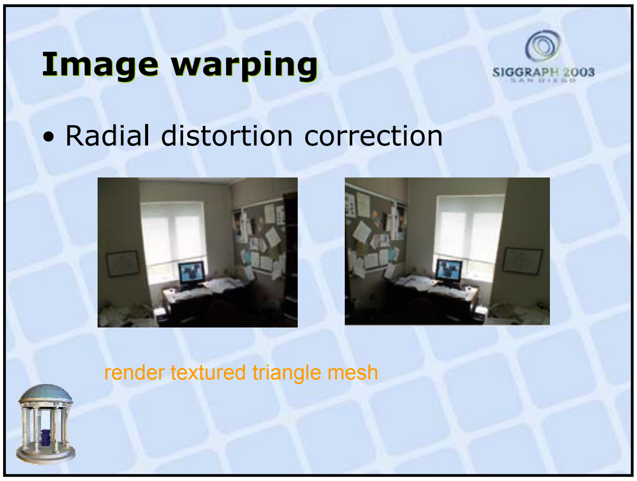


Image warping

- Radial distortion correction

render textured triangle mesh

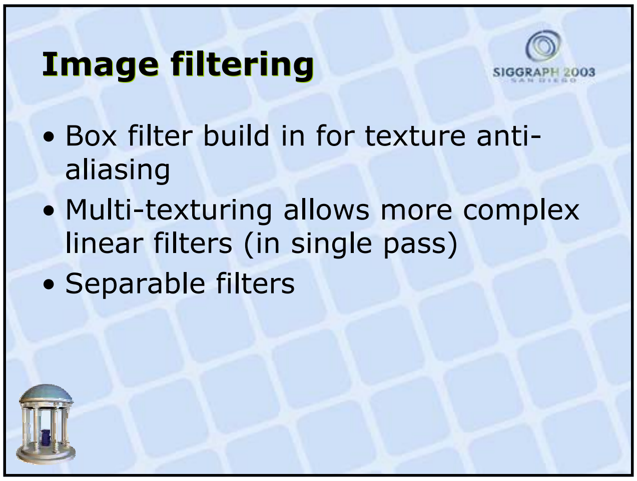
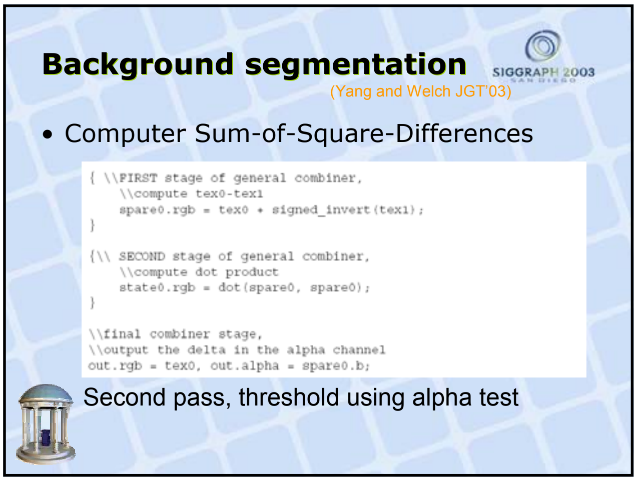


Image filtering

- Box filter build in for texture anti-aliasing
- Multi-texturing allows more complex linear filters (in single pass)
- Separable filters



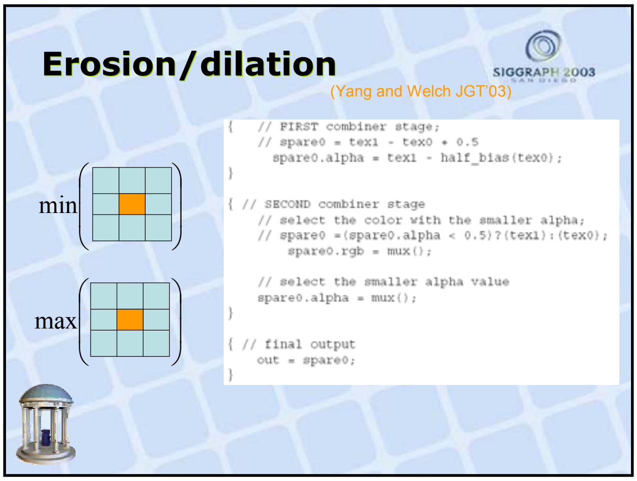
Background segmentation

(Yang and Welch JGT'03)

- Computer Sum-of-Square-Differences

```
{ \\FIRST stage of general combiner,
    \\compute tex0-tex1
    spare0.rgb = tex0 + signed_invert(tex1);
}

{\\ SECOND stage of general combiner,
    \\compute dot product
    state0.rgb = dot(spare0, spare0);
}

\\final combiner stage,
\\output the delta in the alpha channel
out.rgb = tex0, out.alpha = spare0.b;
```

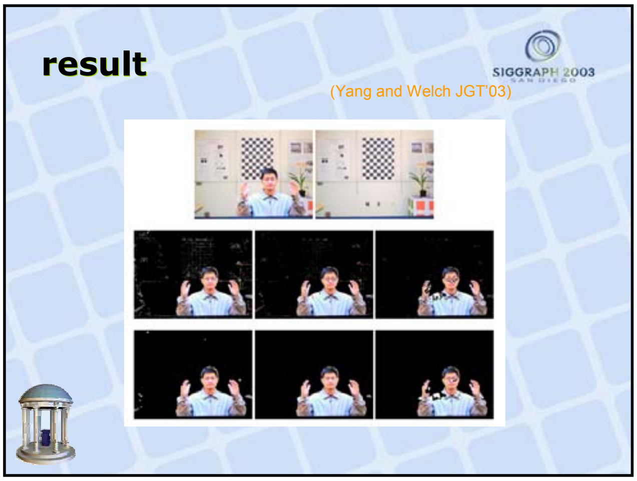
Second pass, threshold using alpha test



Erosion/dilation

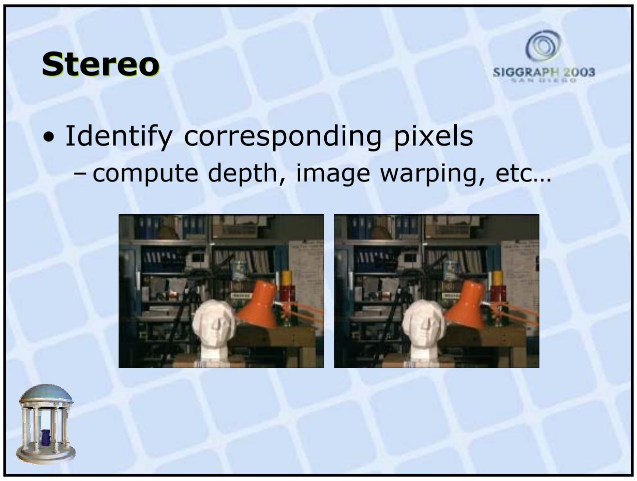
(Yang and Welch JGT'03)

```
{   // FIRST combiner stage;
    // spare0 = tex1 - tex0 + 0.5
      spare0.alpha = tex1 - half_bias(tex0);
}

{ // SECOND combiner stage
    // select the color with the smaller alpha;
    // spare0 =(spare0.alpha < 0.5)?(tex1):(tex0);
        spare0.rgb = mux();

    // select the smaller alpha value
    spare0.alpha = mux();
}

{ // final output
    out = spare0;
}
```



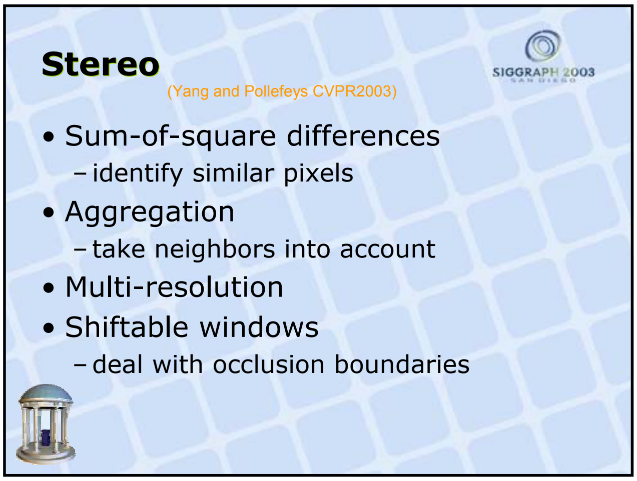
result

(Yang and Welch JGT'03)



Stereo

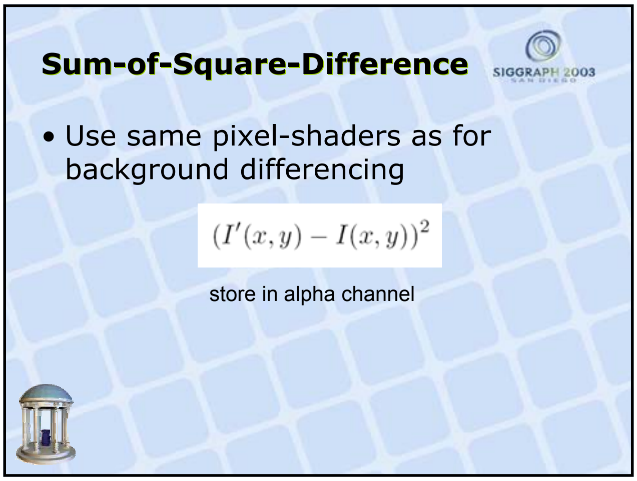
- Identify corresponding pixels
  - compute depth, image warping, etc…



Stereo

(Yang and Pollefeys CVPR2003)

- Sum-of-square differences
  - identify similar pixels
- Aggregation
  - take neighbors into account
- Multi-resolution
- Shiftable windows
  - deal with occlusion boundaries

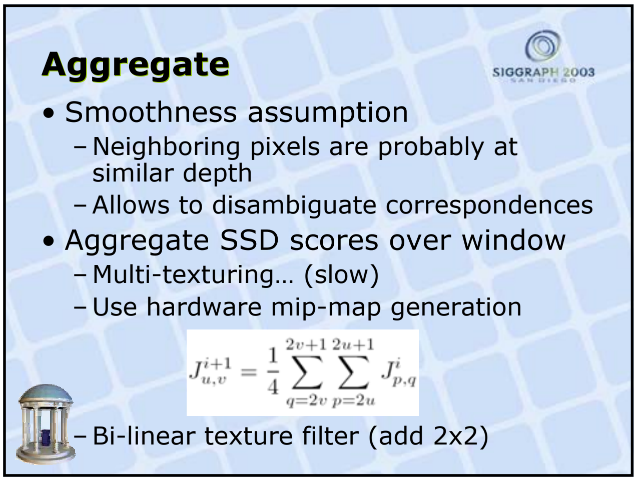


Sum-of-Square-Difference

- Use same pixel-shaders as for background differencing

$$(I'(x,y) - I(x,y))^2$$

store in alpha channel

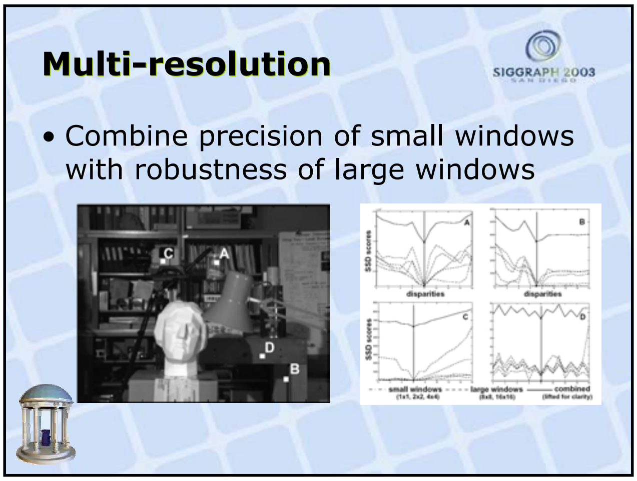


Aggregate

- Smoothness assumption
  - Neighboring pixels are probably at similar depth
  - Allows to disambiguate correspondences
- Aggregate SSD scores over window
  - Multi-texturing… (slow)
  - Use hardware mip-map generation

$$J^{i+1}_{u,v} = \frac{1}{4}\sum_{q=2v}^{2v+1}\sum_{p=2u}^{2u+1} J^{i}_{p,q}$$

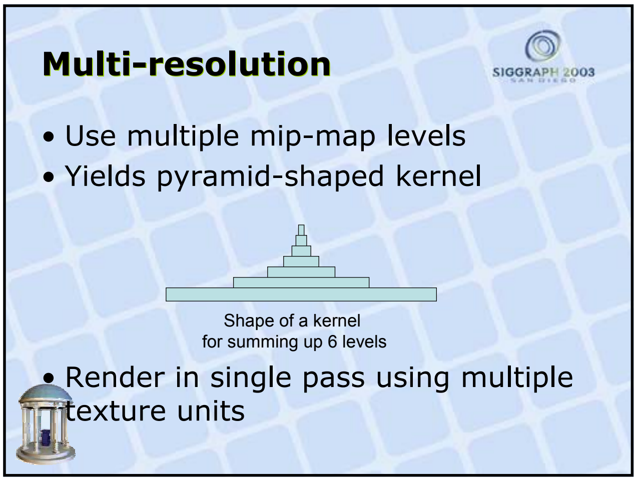
  - Bi-linear texture filter (add 2x2)



Multi-resolution

- Combine precision of small windows with robustness of large windows

SIGGRAPH 2003
SAN DIEGO
C
A
D
B
SSD scores
disparities
small windows (1x1, 2x2, 4x4)
large windows (8x8, 16x16)
combined (lifted for clarity)

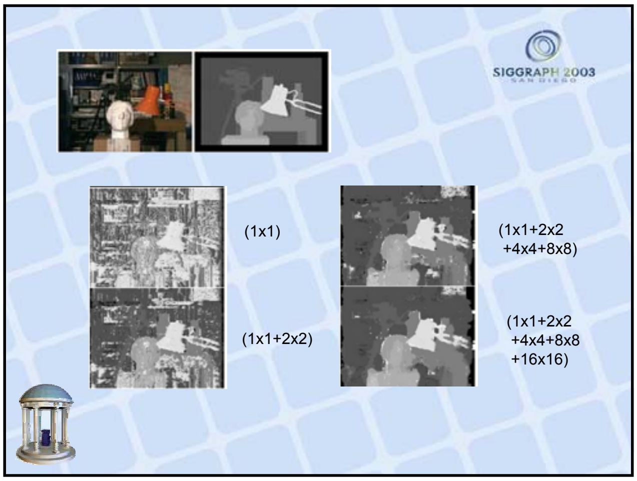


Multi-resolution

- Use multiple mip-map levels
- Yields pyramid-shaped kernel

Shape of a kernel
for summing up 6 levels

- Render in single pass using multiple texture units

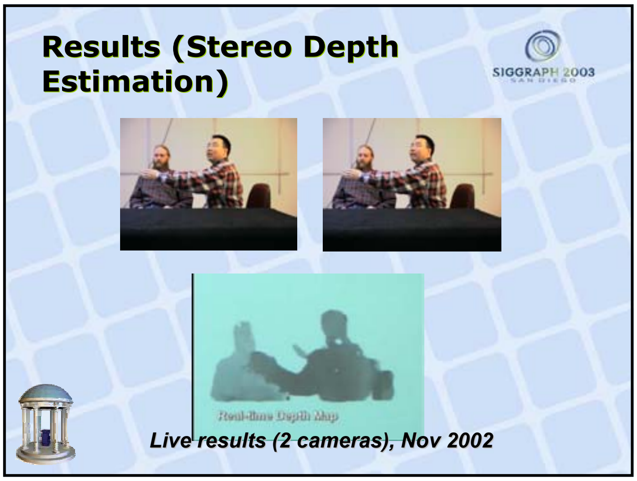


(1x1)

(1x1+2x2)

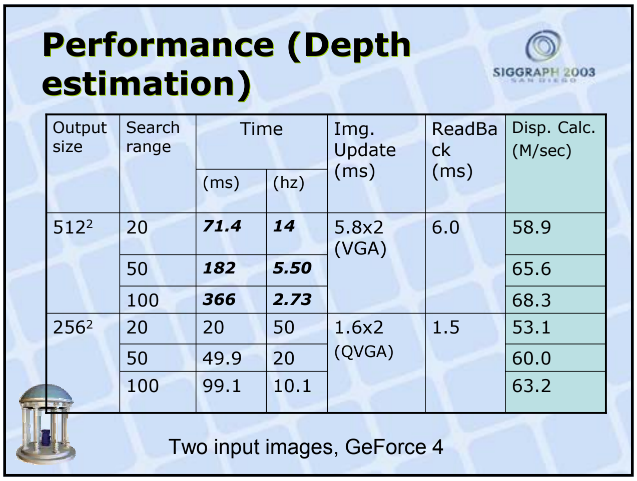
(1x1+2x2
+4x4+8x8)

(1x1+2x2
+4x4+8x8
+16x16)



Results (Stereo Depth Estimation)

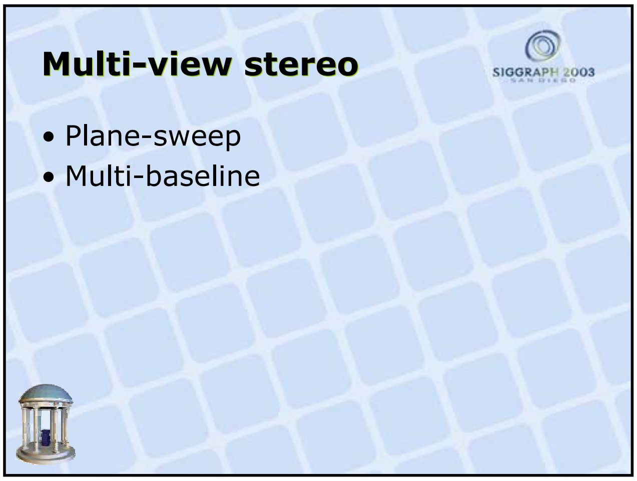
Live results (2 cameras), Nov 2002



Performance (Depth estimation)

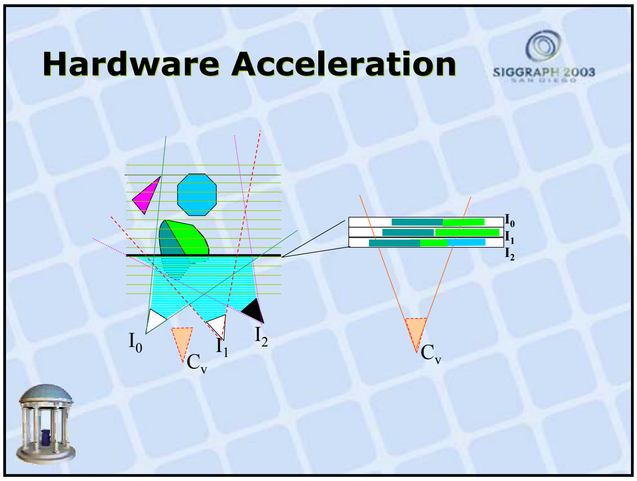
Output size	Search range	Time		Img. Update (ms)	ReadBack (ms)	Disp. Calc. (M/sec)
		(ms)	(hz)			
512^2	20	***71.4***	***14***	5.8x2 (VGA)	6.0	58.9
	50	***182***	***5.50***			65.6
	100	***366***	***2.73***			68.3
256^2	20	20	50	1.6x2 (QVGA)	1.5	53.1
	50	49.9	20			60.0
	100	99.1	10.1			63.2

Two input images, GeForce 4

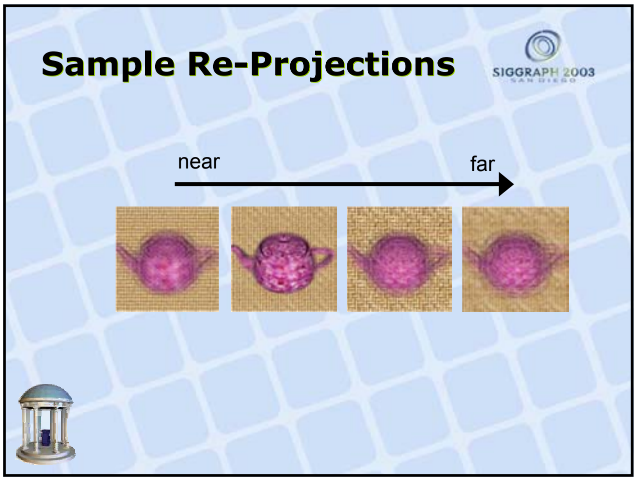


Multi-view stereo

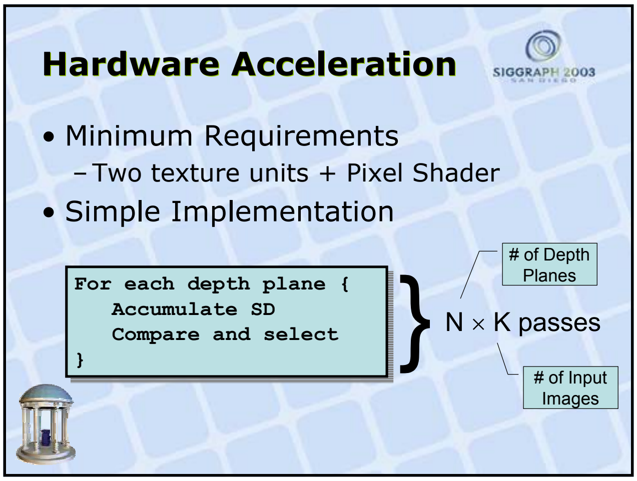
- Plane-sweep
- Multi-baseline



Hardware Acceleration



Sample Re-Projections



Hardware Acceleration

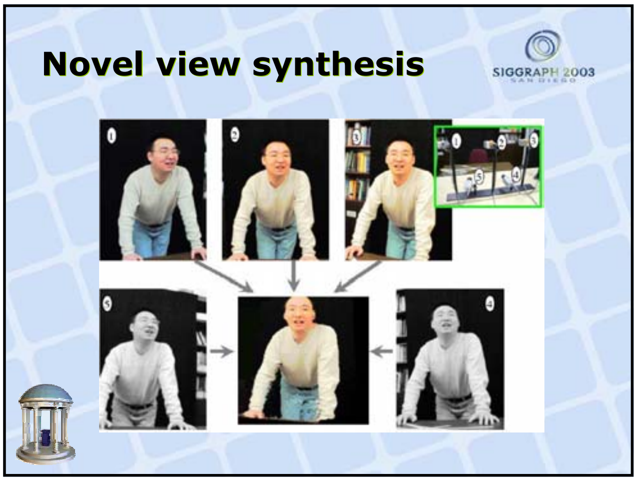
- Minimum Requirements
  - Two texture units + Pixel Shader
- Simple Implementation

```
For each depth plane {
    Accumulate SD
    Compare and select
}
```

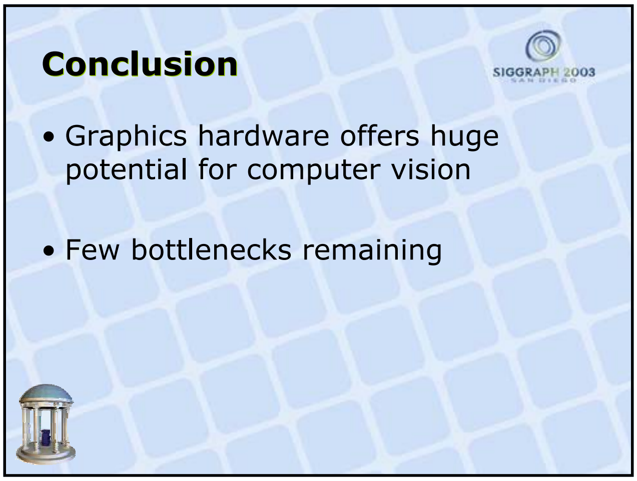
N × K passes

# of Depth Planes

# of Input Images



Novel view synthesis

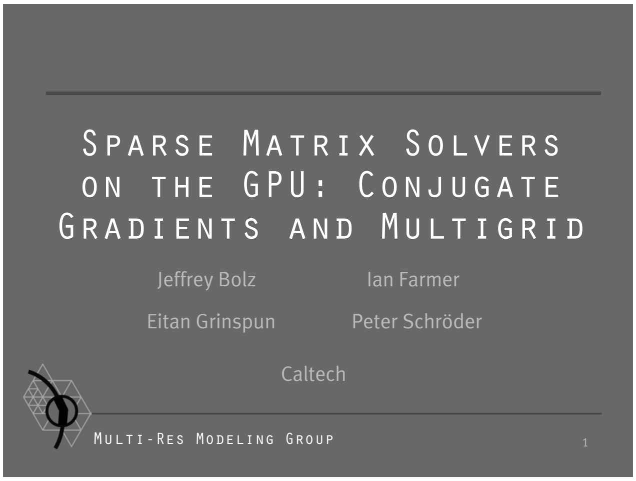


Conclusion

- Graphics hardware offers huge potential for computer vision
- Few bottlenecks remaining

Scientific Computations using Graphics Hardware

Peter Schröder
California Institute of Technology



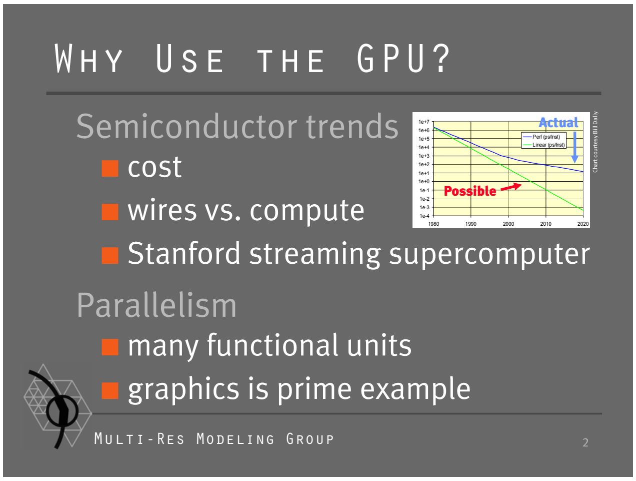
Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid

Jeffrey Bolz Ian Farmer

Eitan Grinspun Peter Schröder

Caltech

Multi-Res Modeling Group 1



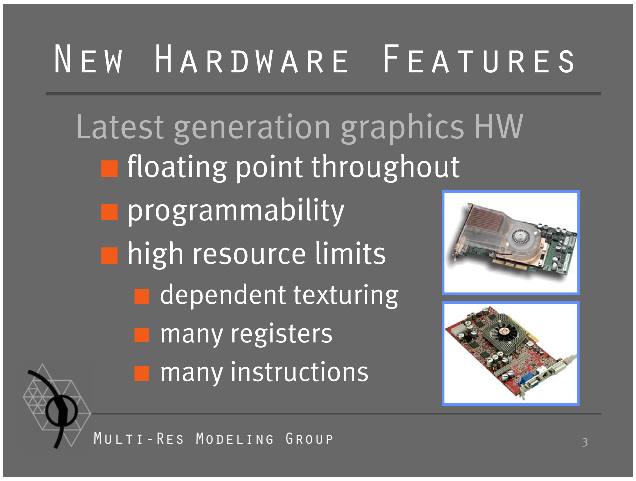
Why Use the GPU?

Semiconductor trends
- cost
- wires vs. compute
- Stanford streaming supercomputer

Parallelism
- many functional units
- graphics is prime example

Chart courtesy Bill Dally

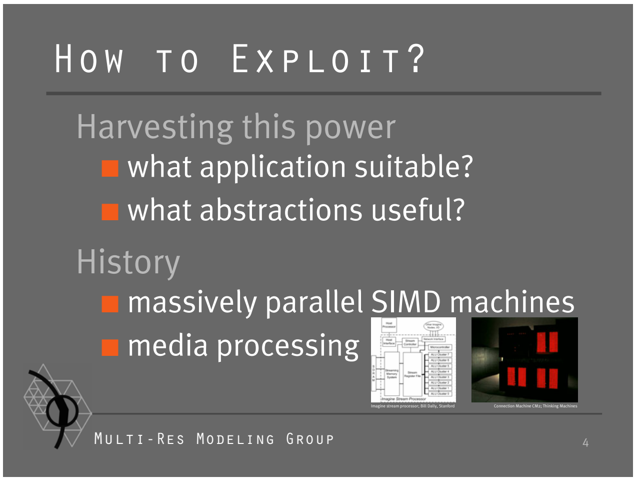
Multi-Res Modeling Group 2



New Hardware Features

Latest generation graphics HW
- floating point throughout
- programmability
- high resource limits
  - dependent texturing
  - many registers
  - many instructions

Multi-Res Modeling Group 3



How to Exploit?

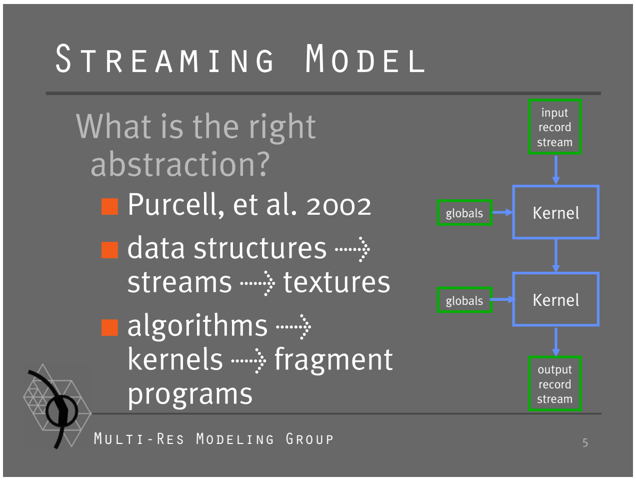
Harvesting this power
- what application suitable?
- what abstractions useful?

History
- massively parallel SIMD machines
- media processing

Imagine stream processor; Bill Dally, Stanford

Connection Machine CM2; Thinking Machines

Multi-Res Modeling Group 4



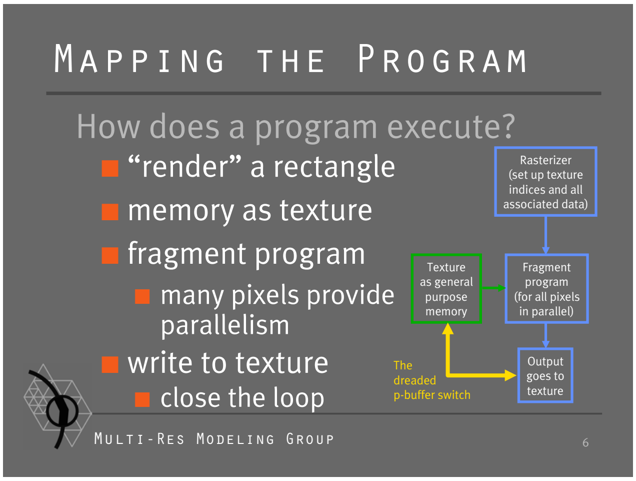
Streaming Model

What is the right abstraction?

- Purcell, et al. 2002
- data structures ⇢ streams ⇢ textures
- algorithms ⇢ kernels ⇢ fragment programs

input record stream
globals
Kernel
globals
Kernel
output record stream

Multi-Res Modeling Group 5



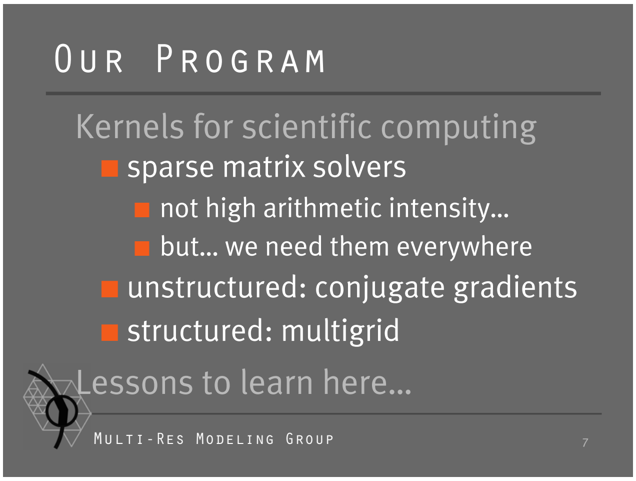
Mapping the Program

How does a program execute?

- “render” a rectangle
- memory as texture
- fragment program
  - many pixels provide parallelism
- write to texture
  - close the loop

Rasterizer (set up texture indices and all associated data)
Texture as general purpose memory
Fragment program (for all pixels in parallel)
Output goes to texture
The dreaded p-buffer switch

Multi-Res Modeling Group 6



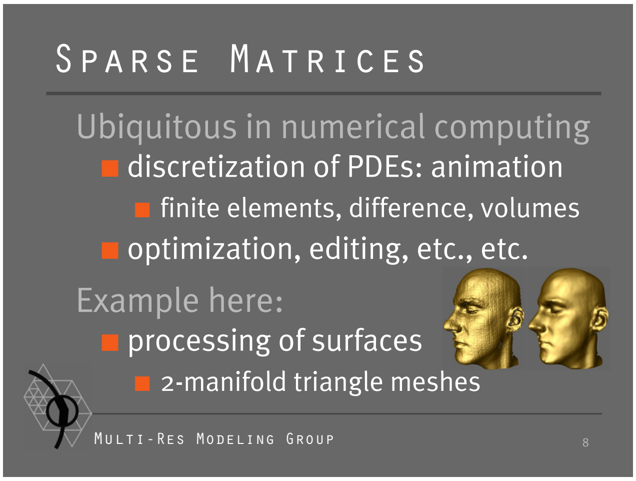
Our Program

Kernels for scientific computing

- sparse matrix solvers
  - not high arithmetic intensity...
  - but... we need them everywhere
- unstructured: conjugate gradients
- structured: multigrid

Lessons to learn here...

Multi-Res Modeling Group 7



Sparse Matrices

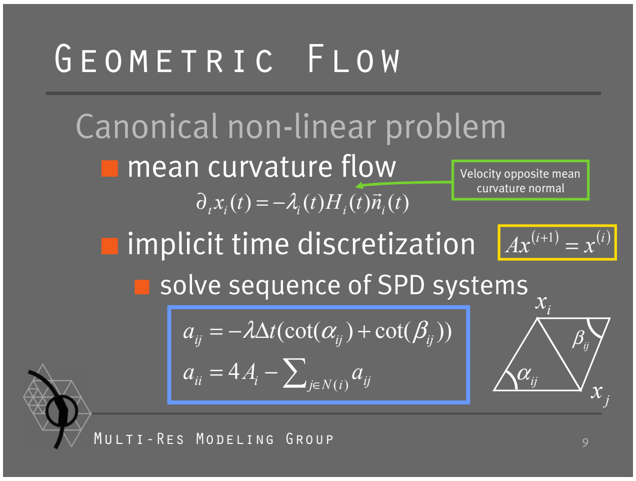
Ubiquitous in numerical computing

- discretization of PDEs: animation
  - finite elements, difference, volumes
- optimization, editing, etc., etc.

Example here:

- processing of surfaces
  - 2-manifold triangle meshes

Multi-Res Modeling Group 8



Geometric Flow

Canonical non-linear problem

- mean curvature flow

  $$\partial_t x_i(t) = -\lambda_i(t) H_i(t) \vec{n}_i(t)$$

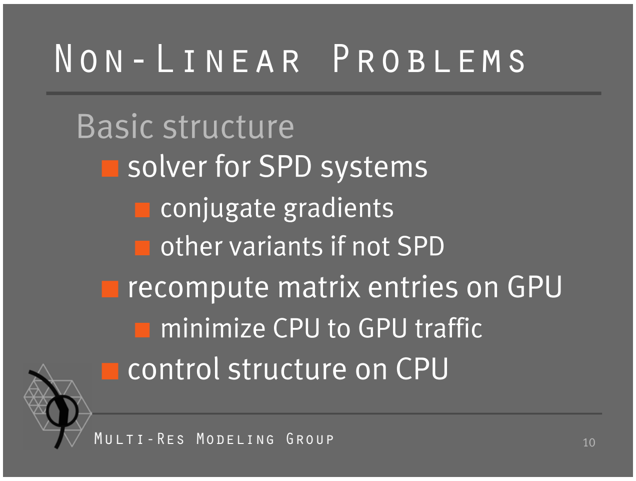
  Velocity opposite mean curvature normal

- implicit time discretization $\boxed{Ax^{(i+1)} = x^{(i)}}$
  - solve sequence of SPD systems

$$\boxed{\begin{aligned} a_{ij} &= -\lambda \Delta t(\cot(\alpha_{ij}) + \cot(\beta_{ij})) \\ a_{ii} &= 4A_i - \textstyle\sum_{j \in N(i)} a_{ij} \end{aligned}}$$

x_i
β_{ij}
α_{ij}
x_j

Multi-Res Modeling Group 9

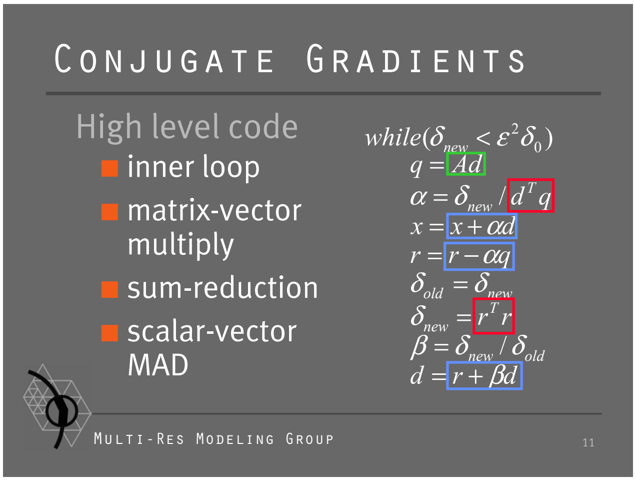


Non-Linear Problems

Basic structure

- solver for SPD systems
  - conjugate gradients
  - other variants if not SPD
- recompute matrix entries on GPU
  - minimize CPU to GPU traffic
- control structure on CPU

Multi-Res Modeling Group 10



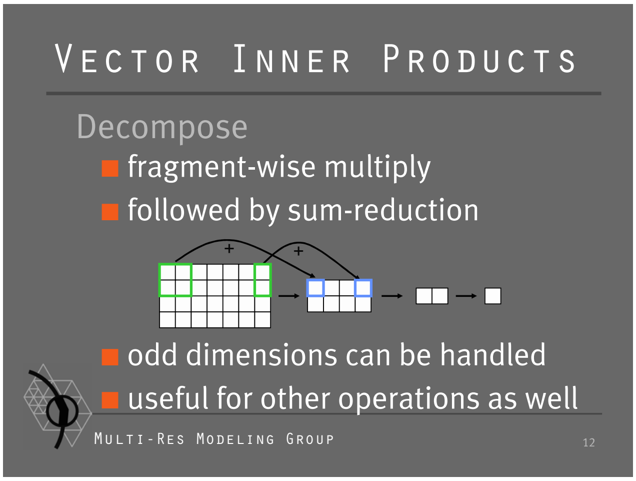
Conjugate Gradients

High level code

- inner loop
- matrix-vector multiply
- sum-reduction
- scalar-vector MAD

```
$while(\delta_{new} < \varepsilon^2 \delta_0)$
    $q = \boxed{Ad}$
    $\alpha = \delta_{new} / \boxed{d^T q}$
    $x = \boxed{x + \alpha d}$
    $r = \boxed{r - \alpha q}$
    $\delta_{old} = \delta_{new}$
    $\delta_{new} = \boxed{r^T r}$
    $\beta = \delta_{new} / \delta_{old}$
    $d = \boxed{r + \beta d}$
```

Multi-Res Modeling Group 11



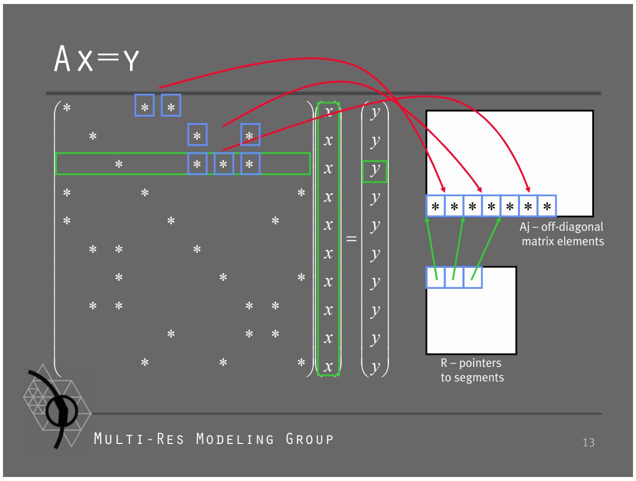
Vector Inner Products

Decompose

- fragment-wise multiply
- followed by sum-reduction

- odd dimensions can be handled
- useful for other operations as well

Multi-Res Modeling Group 12



Ax=y

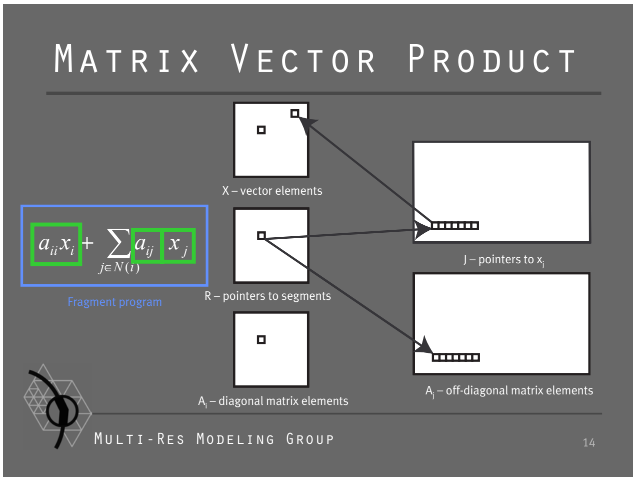
$$\begin{pmatrix} * & & & * & * & & & & & \\ & * & & & & * & & * & & \\ & & * & & & * & * & * & & \\ * & & & * & & & & & & * \\ * & & & & * & & & & * & \\ & * & * & & & * & & & & \\ & & * & & & & * & & & * \\ & * & * & & & & & * & * & \\ & & & & * & & & * & * & \\ & & & * & & & * & & & * \end{pmatrix} \begin{pmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \\ x \\ x \\ x \end{pmatrix} = \begin{pmatrix} y \\ y \\ y \\ y \\ y \\ y \\ y \\ y \\ y \\ y \end{pmatrix}$$

Aj – off-diagonal matrix elements

R – pointers to segments

Multi-Res Modeling Group

13



Matrix Vector Product

$$a_{ii}x_i + \sum_{j\in N(i)} a_{ij}\, x_j$$

Fragment program

X – vector elements

R – pointers to segments

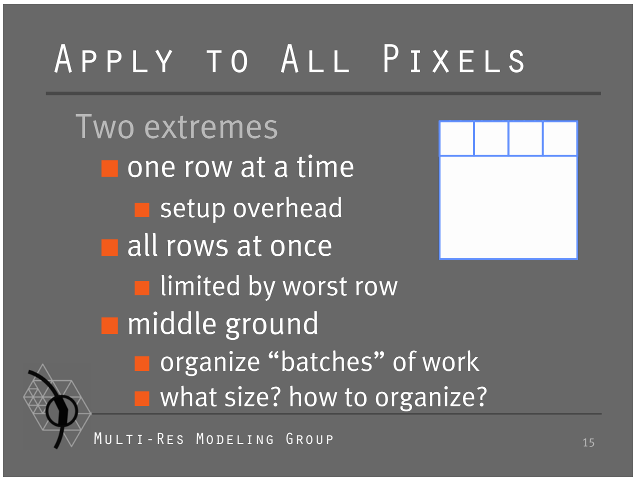
A_i – diagonal matrix elements

J – pointers to x_j

A_i – off-diagonal matrix elements

Multi-Res Modeling Group

14



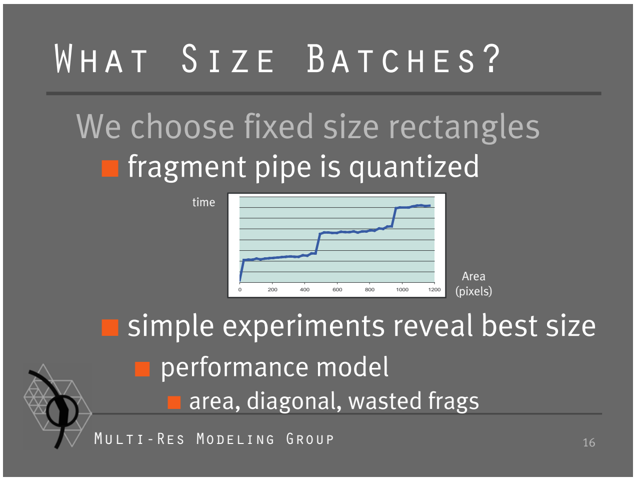
Apply to All Pixels

Two extremes

- one row at a time
  - setup overhead
- all rows at once
  - limited by worst row
- middle ground
  - organize “batches” of work
  - what size? how to organize?

Multi-Res Modeling Group

15



What Size Batches?

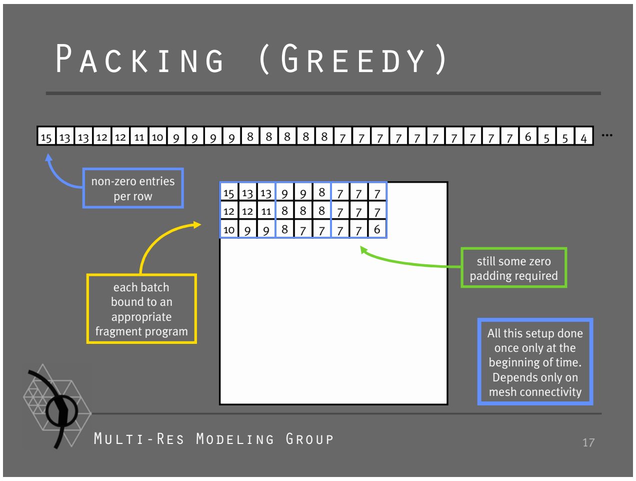
We choose fixed size rectangles

- fragment pipe is quantized

- simple experiments reveal best size
  - performance model
    - area, diagonal, wasted frags

Multi-Res Modeling Group

16



Packing (Greedy)

15	13	13	12	12	11	10	9	9	9	9	8	8	8	8	8	7	7	7	7	7	7	7	7	7	7	6	5	5	4	…

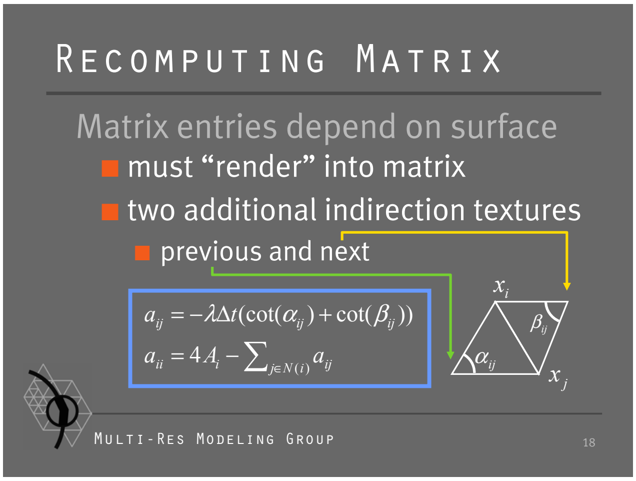
non-zero entries per row

15	13	13	9	9	8	7	7	7
12	12	11	8	8	8	7	7	7
10	9	9	8	7	7	7	7	6

each batch bound to an appropriate fragment program

still some zero padding required

All this setup done once only at the beginning of time. Depends only on mesh connectivity



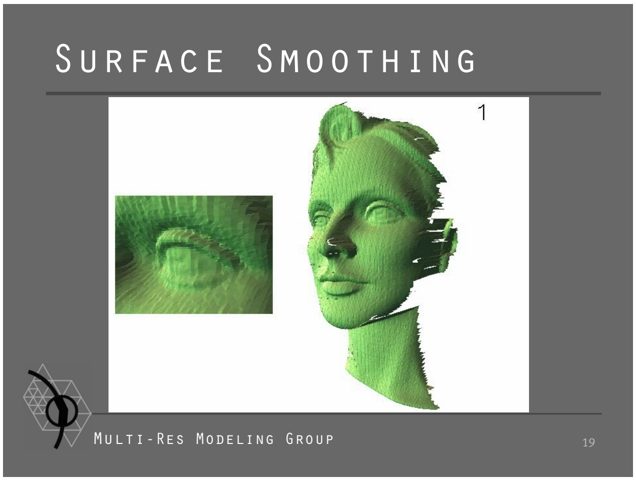
Recomputing Matrix

Matrix entries depend on surface

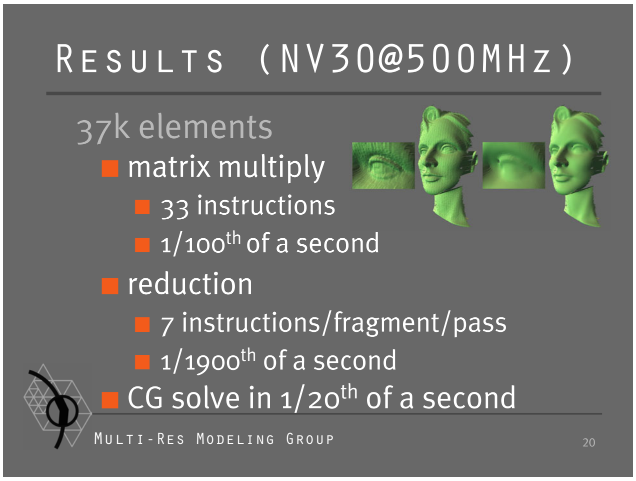
- must "render" into matrix
- two additional indirection textures
  - previous and next

$$a_{ij} = -\lambda\Delta t(\cot(\alpha_{ij}) + \cot(\beta_{ij}))$$

$$a_{ii} = 4A_i - \sum_{j\in N(i)} a_{ij}$$



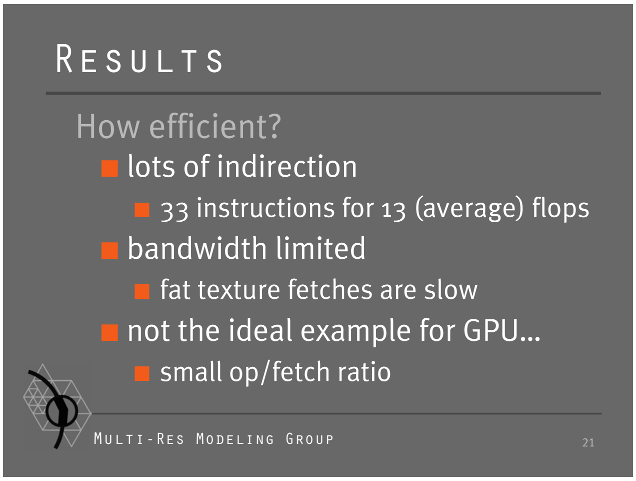
Surface Smoothing



Results (NV30@500MHz)

37k elements

- matrix multiply
  - 33 instructions
  - 1/100th of a second
- reduction
  - 7 instructions/fragment/pass
  - 1/1900th of a second
- CG solve in 1/20th of a second

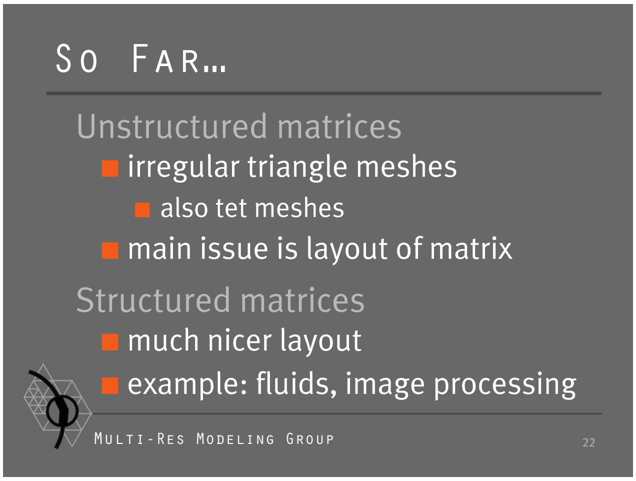


Results

How efficient?

- lots of indirection
  - 33 instructions for 13 (average) flops
- bandwidth limited
  - fat texture fetches are slow
- not the ideal example for GPU…
  - small op/fetch ratio

Multi-Res Modeling Group 21



So Far…

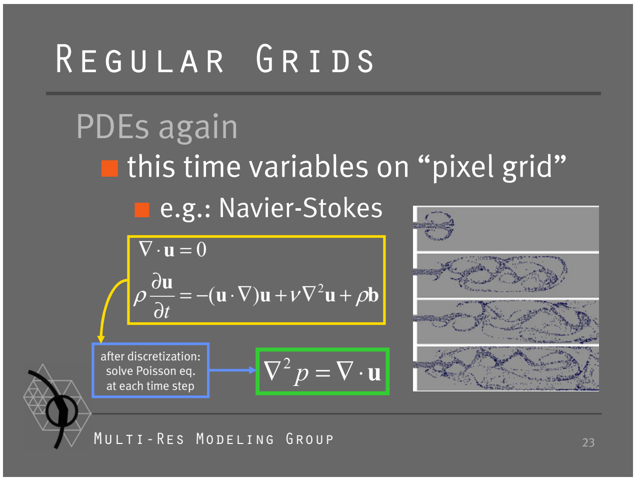
Unstructured matrices

- irregular triangle meshes
  - also tet meshes
- main issue is layout of matrix

Structured matrices

- much nicer layout
- example: fluids, image processing

Multi-Res Modeling Group 22



Regular Grids

PDEs again

- this time variables on “pixel grid”
  - e.g.: Navier-Stokes

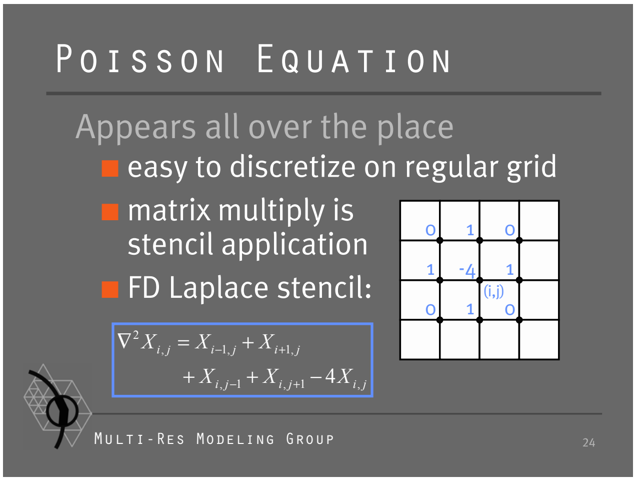
$$\nabla \cdot \mathbf{u} = 0$$

$$\rho \frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \rho \mathbf{b}$$

after discretization: solve Poisson eq. at each time step

$$\nabla^2 p = \nabla \cdot \mathbf{u}$$

Multi-Res Modeling Group 23



Poisson Equation

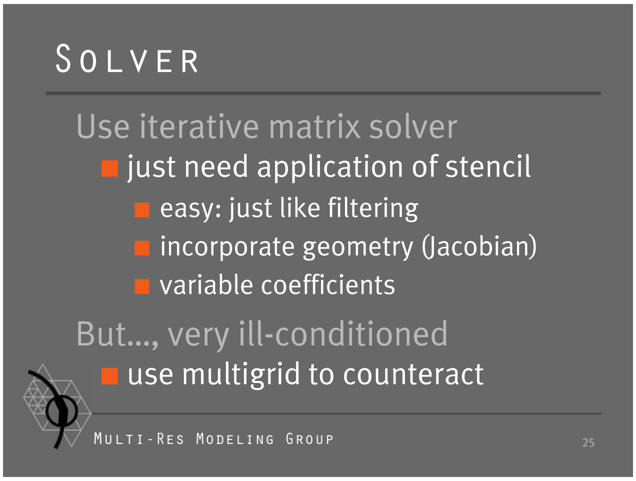
Appears all over the place

- easy to discretize on regular grid
- matrix multiply is stencil application
- FD Laplace stencil:

$$\nabla^2 X_{i,j} = X_{i-1,j} + X_{i+1,j} + X_{i,j-1} + X_{i,j+1} - 4X_{i,j}$$

0	1	0
1	-4	1
0	1	0

Multi-Res Modeling Group 24



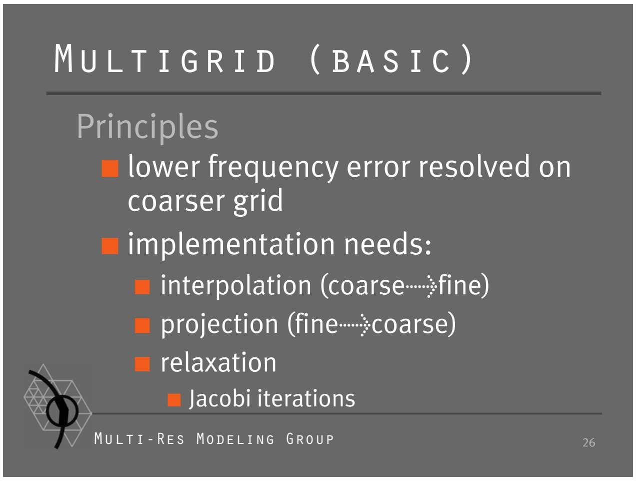
Solver

Use iterative matrix solver

- just need application of stencil
  - easy: just like filtering
  - incorporate geometry (Jacobian)
  - variable coefficients

But..., very ill-conditioned

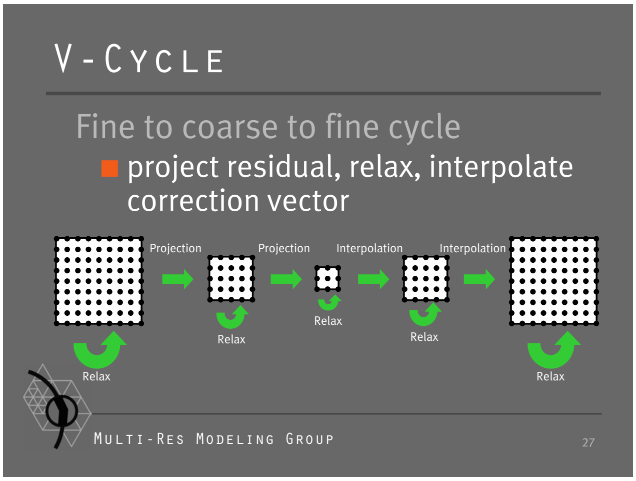
- use multigrid to counteract



Multigrid (basic)

Principles

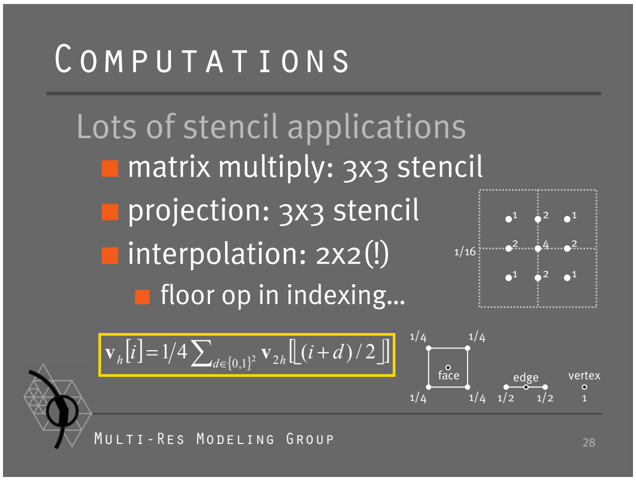
- lower frequency error resolved on coarser grid
- implementation needs:
  - interpolation (coarse⇢fine)
  - projection (fine⇢coarse)
  - relaxation
    - Jacobi iterations



V-Cycle

Fine to coarse to fine cycle

- project residual, relax, interpolate correction vector

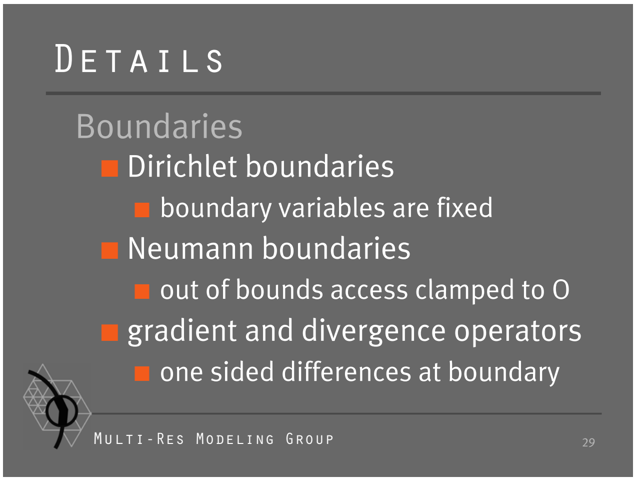


Computations

Lots of stencil applications

- matrix multiply: 3x3 stencil
- projection: 3x3 stencil
- interpolation: 2x2(!)
  - floor op in indexing...

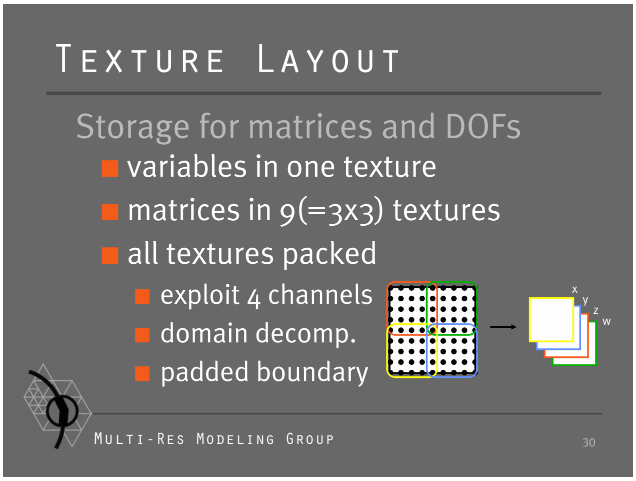
$$\boxed{\mathbf{v}_h[i] = 1/4 \sum_{d\in\{0,1\}^2} \mathbf{v}_{2h}[\![\lfloor (i+d)/2 \rfloor]\!]}$$



Details

Boundaries

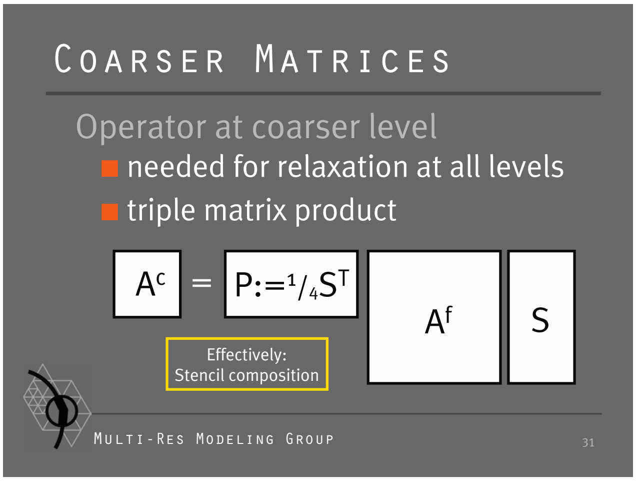
- Dirichlet boundaries
  - boundary variables are fixed
- Neumann boundaries
  - out of bounds access clamped to O
- gradient and divergence operators
  - one sided differences at boundary



Texture Layout

Storage for matrices and DOFs

- variables in one texture
- matrices in 9(=3x3) textures
- all textures packed
  - exploit 4 channels
  - domain decomp.
  - padded boundary



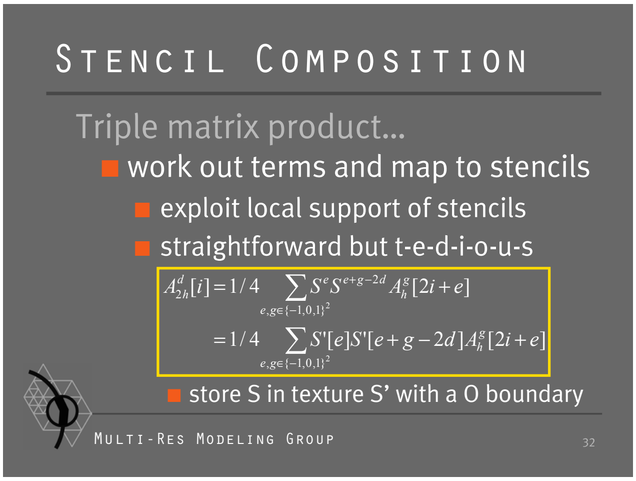
Coarser Matrices

Operator at coarser level

- needed for relaxation at all levels
- triple matrix product

A^c = $P:=\frac{1}{4}S^T$ A^f S

Effectively:
Stencil composition



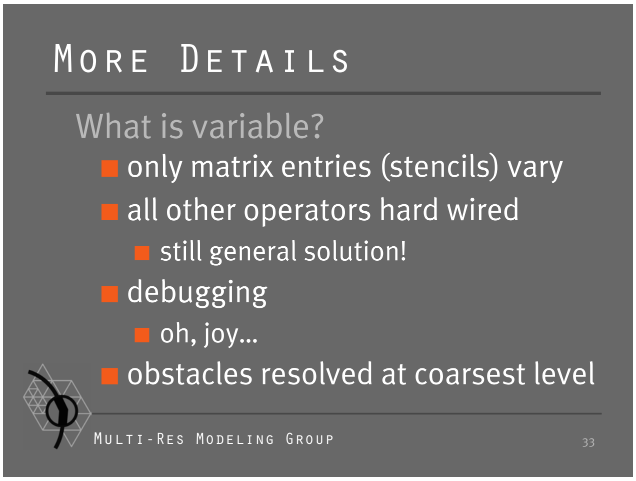
Stencil Composition

Triple matrix product...

- work out terms and map to stencils
  - exploit local support of stencils
  - straightforward but t-e-d-i-o-u-s

$$\begin{aligned} A_{2h}^{d}[i] &= 1/4 \sum_{e,g\in\{-1,0,1\}^2} S^{e} S^{e+g-2d} A_{h}^{g}[2i+e] \\ &= 1/4 \sum_{e,g\in\{-1,0,1\}^2} S'[e] S'[e+g-2d] A_{h}^{g}[2i+e] \end{aligned}$$

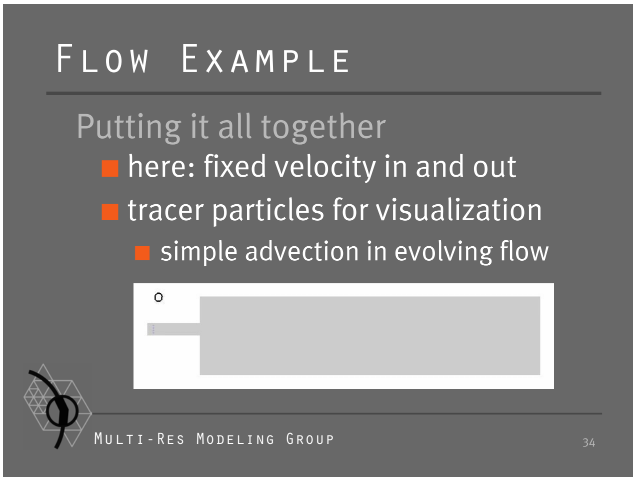
- store S in texture S' with a O boundary



More Details

What is variable?

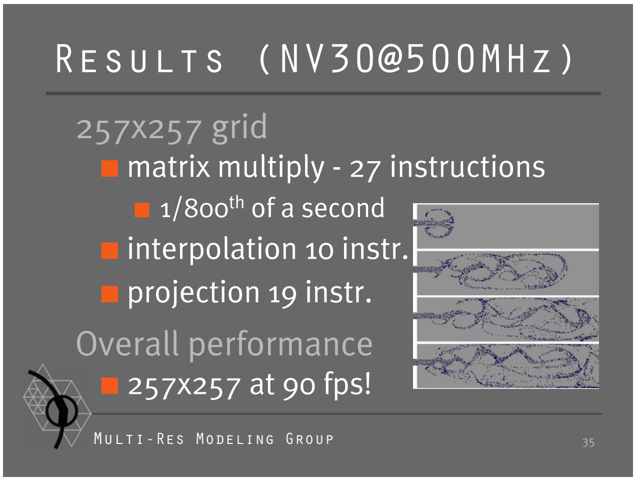
- only matrix entries (stencils) vary
- all other operators hard wired
  - still general solution!
- debugging
  - oh, joy...
- obstacles resolved at coarsest level



Flow Example

Putting it all together

- here: fixed velocity in and out
- tracer particles for visualization
  - simple advection in evolving flow



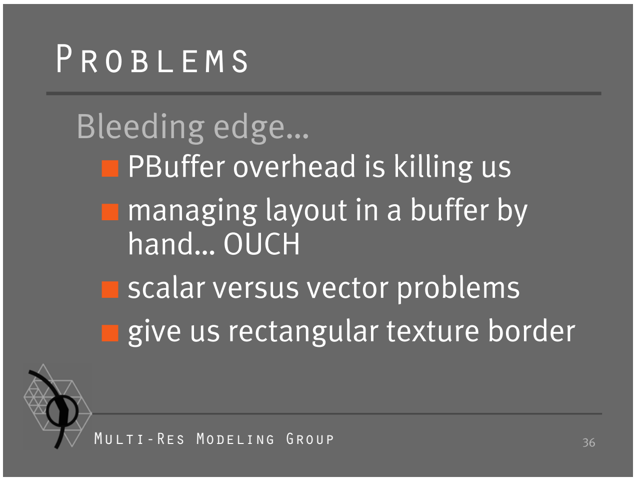
Results (NV30@500MHz)

257x257 grid

- matrix multiply - 27 instructions
  - 1/800th of a second
- interpolation 10 instr.
- projection 19 instr.

Overall performance

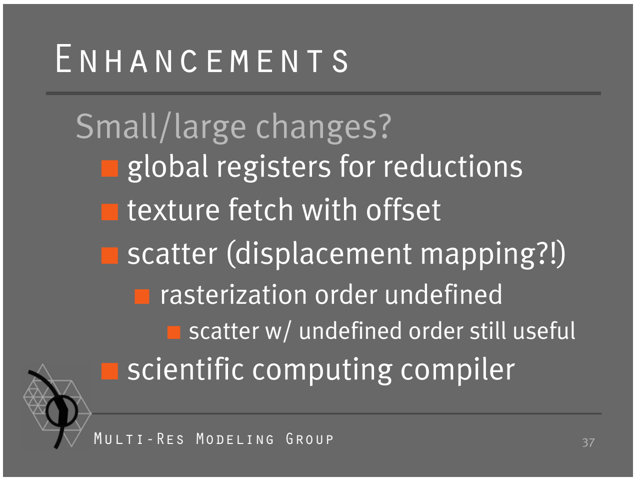
- 257x257 at 90 fps!



Problems

Bleeding edge...

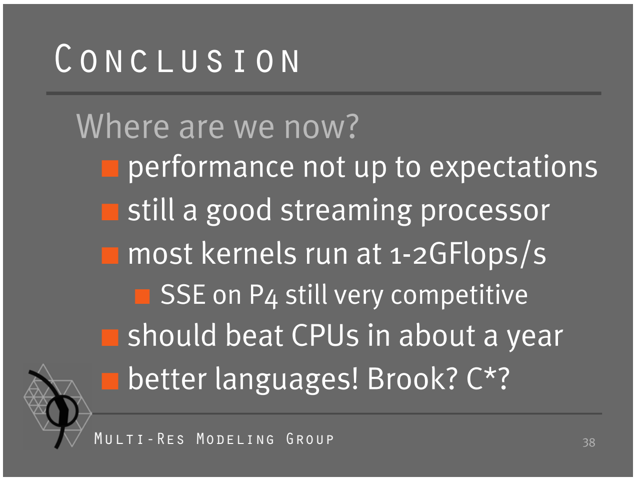
- PBuffer overhead is killing us
- managing layout in a buffer by hand... OUCH
- scalar versus vector problems
- give us rectangular texture border



Enhancements

Small/large changes?

- global registers for reductions
- texture fetch with offset
- scatter (displacement mapping?!)
  - rasterization order undefined
    - scatter w/ undefined order still useful
- scientific computing compiler



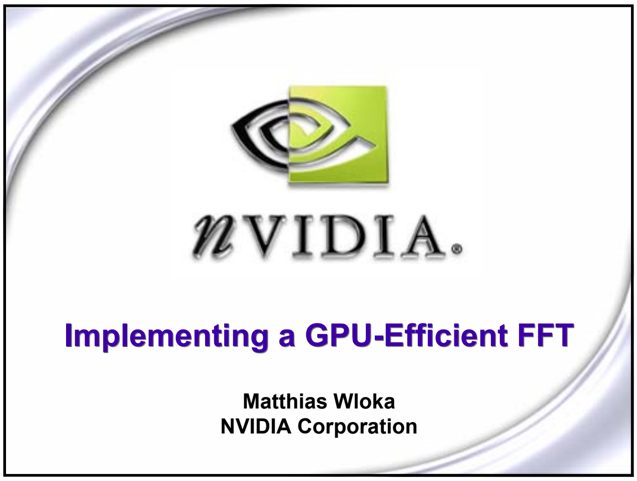
Conclusion

Where are we now?

- performance not up to expectations
- still a good streaming processor
- most kernels run at 1-2GFlops/s
  - SSE on P4 still very competitive
- should beat CPUs in about a year
- better languages! Brook? C*?

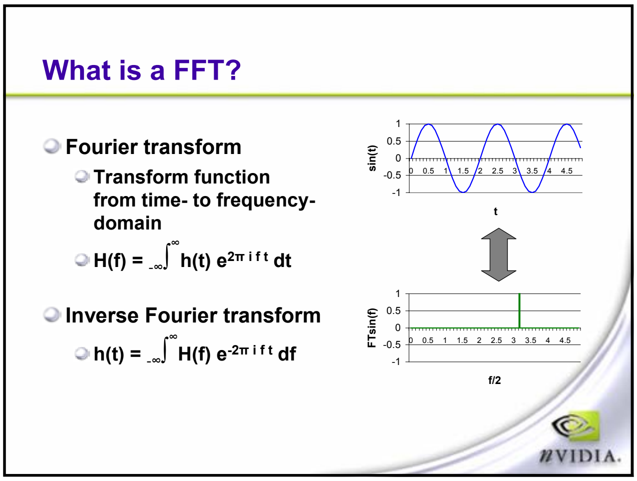
Implementing a GPU-Efficient FFT

Matthias M Wloka
NVIDIA



Implementing a GPU-Efficient FFT

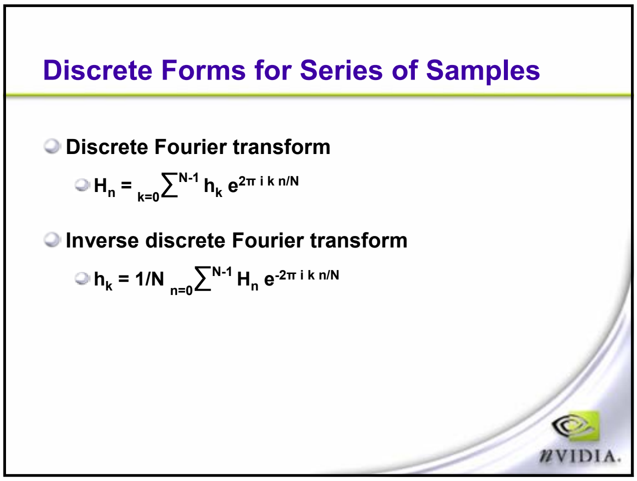
Matthias Wloka
NVIDIA Corporation



What is a FFT?

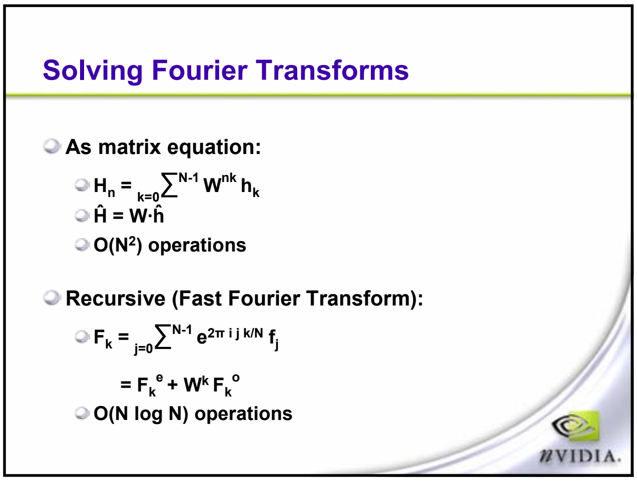
- Fourier transform
  - Transform function from time- to frequency-domain
  - $H(f) = \int_{-\infty}^{\infty} h(t)\, e^{2\pi i f t}\, dt$
- Inverse Fourier transform
  - $h(t) = \int_{-\infty}^{\infty} H(f)\, e^{-2\pi i f t}\, df$

sin(t)
1
0.5
0
-0.5
-1
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5
t
FTsin(f)
1
0.5
0
-0.5
-1
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5
f/2



Discrete Forms for Series of Samples

- Discrete Fourier transform
  - $H_n = \sum_{k=0}^{N-1} h_k\, e^{2\pi i k n/N}$
- Inverse discrete Fourier transform
  - $h_k = 1/N \sum_{n=0}^{N-1} H_n\, e^{-2\pi i k n/N}$



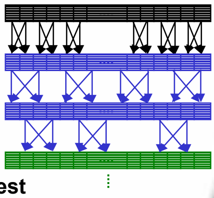
Solving Fourier Transforms

- As matrix equation:
  - $H_n = \sum_{k=0}^{N-1} W^{nk}\, h_k$
  - $\hat{H} = W \cdot \hat{h}$
  - $O(N^2)$ operations
- Recursive (Fast Fourier Transform):
  - $F_k = \sum_{j=0}^{N-1} e^{2\pi i j k/N}\, f_j$

    $= F_k^e + W^k F_k^o$
  - O(N log N) operations

Fast Fourier Transform Implementations

- [Numerical Recipes in C]
  - Loop over elements for bit-reversal
  - Loop log N times to recombine neighbors
  - Weights are computed iteratively



- Fastest Fourier Transform in the West
  - http://fftw.org
  - Optimized for current CPU architectures
  - Adapts itself to current CPU cache sizes

GPU FFT Feasibility

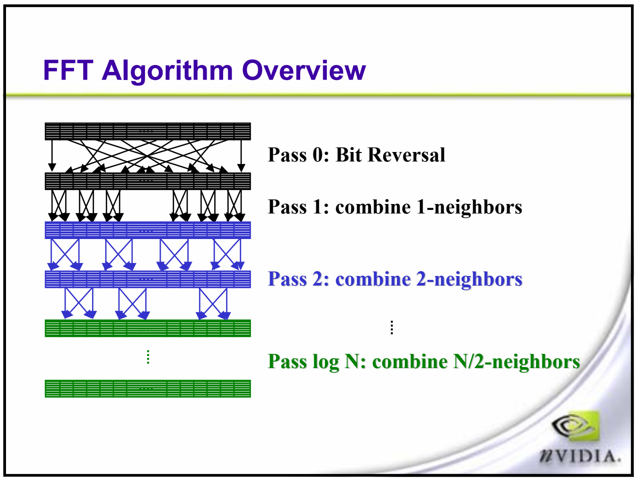
- 4k FFT requires
  - ~5 * 4k log 4k Flops = ~170 MFlops
  - 4k * 8 bytes = 32k bytes data
- Compute times for
  - 3.0 GHz CPU: 170MFlops @ ~12GFlops/s = ~14.2 ms
  - 0.5 GHz GPU: 170MFlops @ ~32GFlops/s = ~5.3 ms
- Data transfer times:
  - Download: 32k @ 2.0 GB/s = 0.016 ms
  - Upload: 32k @ 0.18 GB/s = 0.176 ms

Scenarios

- Only use 3.0 GHz CPU:
  - 1 FFT every ~14.2 ms
- Only use 0.5 GHz GPU:
  - CPU sends data, waits, gets data from GPU
  - 1 FFT every ~5.5 ms
- Use CPU and GPU simultaneously:
  - CPU sends 3 FFTs, computes 1 FFT, reads 3 FFTs
  - 4 FFTs every ~16.5 ms
  - 1 FFT every ~4.1 ms

Scenarios

- Only use 3.0 GHz CPU:
  - Baseline
- Only use 0.5 GHz GPU:
  - **~2.5X** speed-up over baseline
- Use CPU and GPU simultaneously:
  - **~3.5X** speed-up over baseline



FFT Algorithm Overview

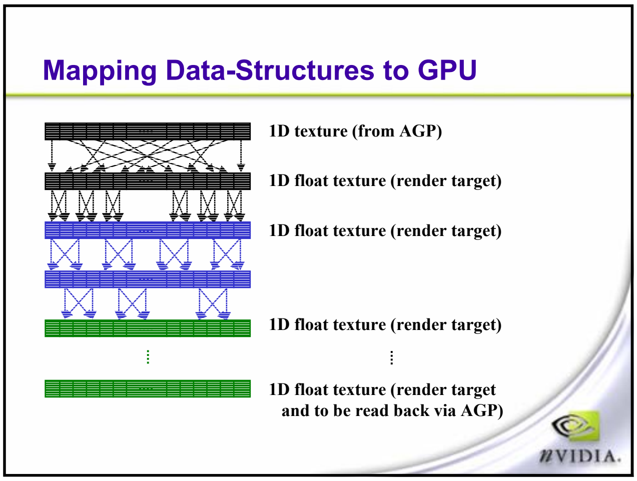
Pass 0: Bit Reversal

Pass 1: combine 1-neighbors

Pass 2: combine 2-neighbors

⋮

Pass log N: combine N/2-neighbors



Mapping Data-Structures to GPU

1D texture (from AGP)

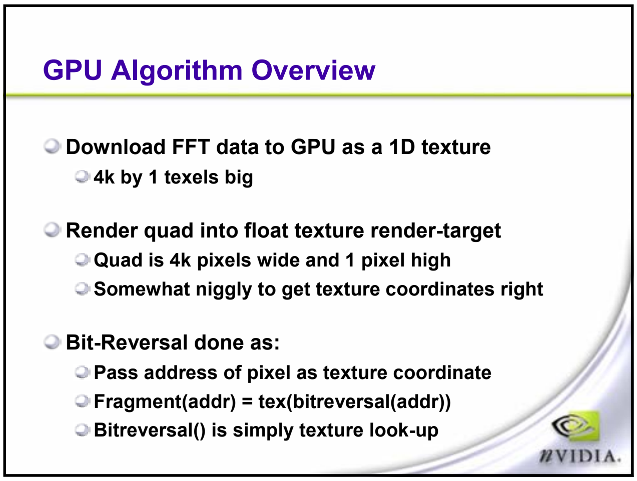
1D float texture (render target)

1D float texture (render target)

1D float texture (render target)

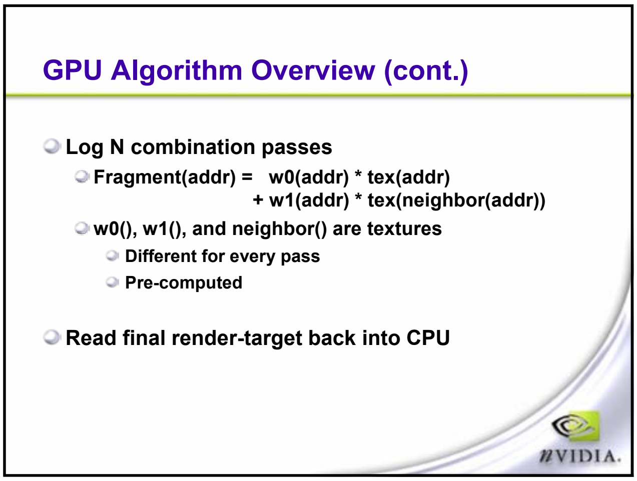
⋮

1D float texture (render target and to be read back via AGP)



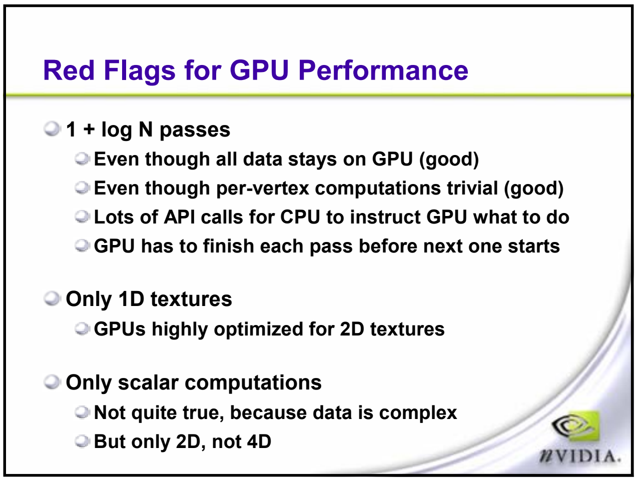
GPU Algorithm Overview

- Download FFT data to GPU as a 1D texture
  - 4k by 1 texels big
- Render quad into float texture render-target
  - Quad is 4k pixels wide and 1 pixel high
  - Somewhat niggly to get texture coordinates right
- Bit-Reversal done as:
  - Pass address of pixel as texture coordinate
  - Fragment(addr) = tex(bitreversal(addr))
  - Bitreversal() is simply texture look-up



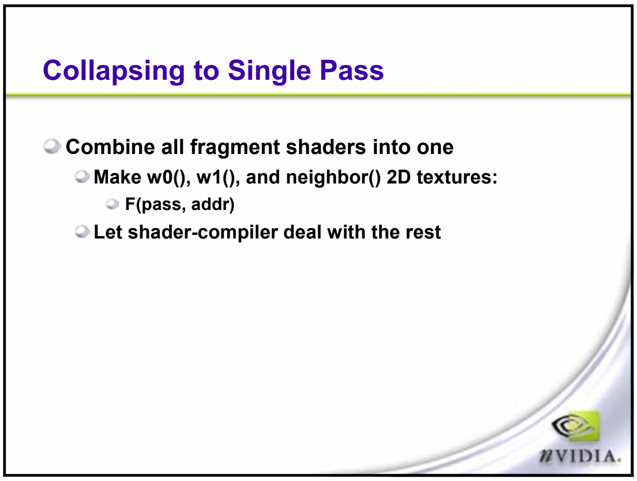
GPU Algorithm Overview (cont.)

- Log N combination passes
  - Fragment(addr) = w0(addr) * tex(addr)
    + w1(addr) * tex(neighbor(addr))
  - w0(), w1(), and neighbor() are textures
    - Different for every pass
    - Pre-computed
- Read final render-target back into CPU



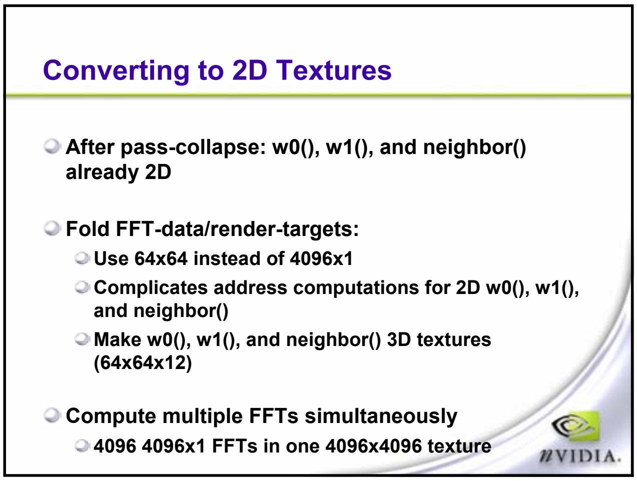
Red Flags for GPU Performance

- 1 + log N passes
  - Even though all data stays on GPU (good)
  - Even though per-vertex computations trivial (good)
  - Lots of API calls for CPU to instruct GPU what to do
  - GPU has to finish each pass before next one starts
- Only 1D textures
  - GPUs highly optimized for 2D textures
- Only scalar computations
  - Not quite true, because data is complex
  - But only 2D, not 4D



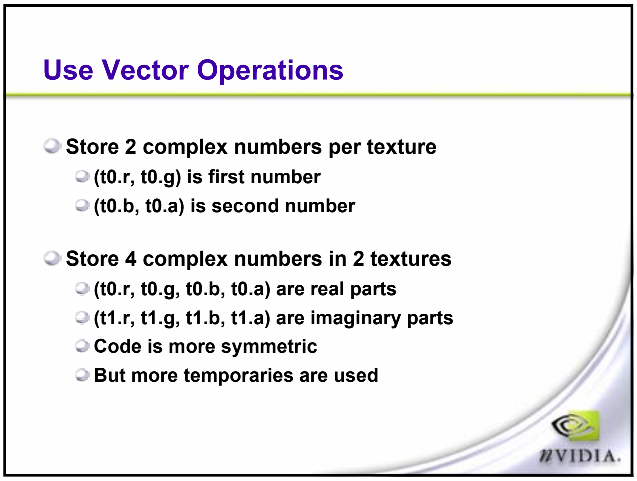
Collapsing to Single Pass

- Combine all fragment shaders into one
  - Make w0(), w1(), and neighbor() 2D textures:
    - F(pass, addr)
  - Let shader-compiler deal with the rest



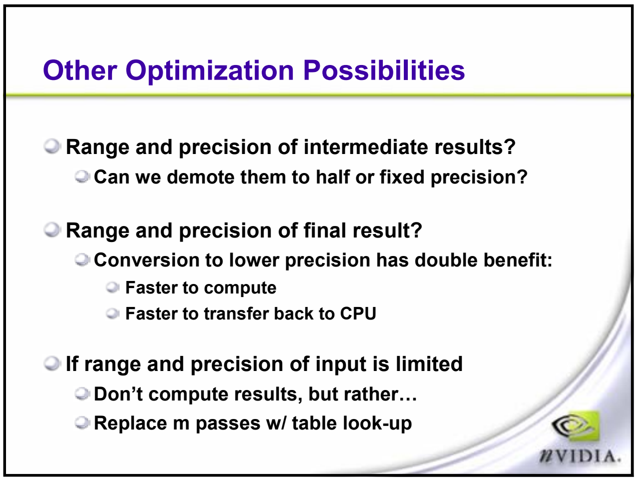
Converting to 2D Textures

- After pass-collapse: w0(), w1(), and neighbor() already 2D
- Fold FFT-data/render-targets:
  - Use 64x64 instead of 4096x1
  - Complicates address computations for 2D w0(), w1(), and neighbor()
  - Make w0(), w1(), and neighbor() 3D textures (64x64x12)
- Compute multiple FFTs simultaneously
  - 4096 4096x1 FFTs in one 4096x4096 texture



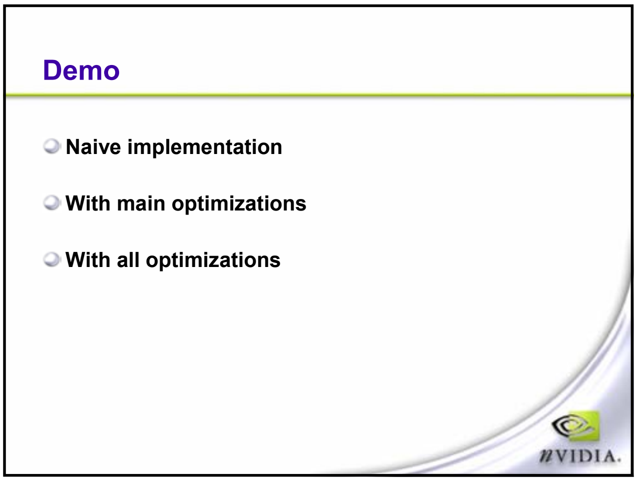
Use Vector Operations

- Store 2 complex numbers per texture
  - (t0.r, t0.g) is first number
  - (t0.b, t0.a) is second number
- Store 4 complex numbers in 2 textures
  - (t0.r, t0.g, t0.b, t0.a) are real parts
  - (t1.r, t1.g, t1.b, t1.a) are imaginary parts
  - Code is more symmetric
  - But more temporaries are used



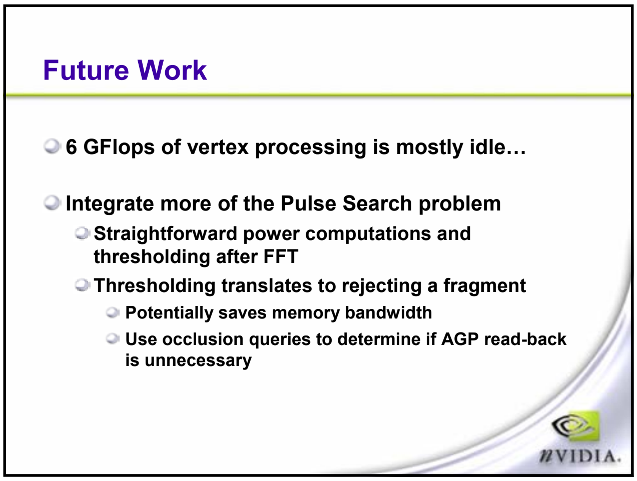
Other Optimization Possibilities

- Range and precision of intermediate results?
  - Can we demote them to half or fixed precision?
- Range and precision of final result?
  - Conversion to lower precision has double benefit:
    - Faster to compute
    - Faster to transfer back to CPU
- If range and precision of input is limited
  - Don't compute results, but rather…
  - Replace m passes w/ table look-up



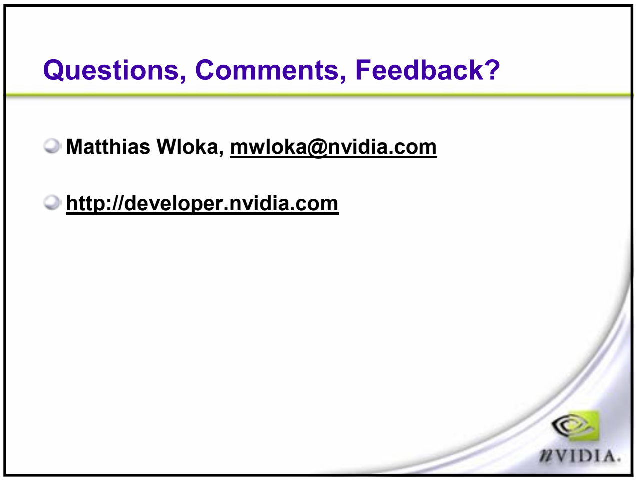
Demo

- Naive implementation
- With main optimizations
- With all optimizations



Future Work

- 6 GFlops of vertex processing is mostly idle…
- Integrate more of the Pulse Search problem
  - Straightforward power computations and thresholding after FFT
  - Thresholding translates to rejecting a fragment
    - Potentially saves memory bandwidth
    - Use occlusion queries to determine if AGP read-back is unnecessary



Questions, Comments, Feedback?

- Matthias Wloka, mwloka@nvidia.com
- http://developer.nvidia.com

Physically-Based Modeling and Interactive Navigation using Graphics Hardware

Ming Lin
University of North Carolina at Chapel Hill

Fast 3D Geometric Proximity Queries between Rigid and Deformable Models Using Graphics Hardware Acceleration

Kenneth E.Hoff III, Andrew Zaferakis, Ming Lin, Dinesh Manocha

The University of North Carolina at Chapel Hill
Department of Computer Science
{hoff,andrewz,lin,dm}@cs.unc.edu

Abstract

We present an approach for computing generalized proximity information between arbitrary polygonal models using graphics hardware acceleration. Our algorithm combines object-space localization, multi-pass rendering techniques, and accelerated distance field computation to perform complex proximity queries at interactive rates. It is applicable to any closed, possibly non-convex, polygonal object and requires no precomputation, making it suitable for both rigid and dynamically deformable geometry of relatively high complexity. The proximity queries include, not only collision detection, but also the computation of intersections, minimum separation distance, closest points, penetration depth and direction, and contact points and normals. The load is balanced between CPU and graphics subsystems through a hybrid geometry and image-based approach. Geometric object-space techniques coarsely localize potential interactions between two objects, and image-space techniques accelerated with graphics hardware provide the low-level proximity information. We have implemented our system using the OpenGL graphics library and have tested it on various hardware configurations with a wide range of object complexities and contact scenarios. In all cases, interactive frame rates are achieved. In addition, our algorithm's performance is heavily based on the graphics hardware computational power growth curve which has exceeded the expectations of Moore's Law for general CPU power growth.

1. Introduction

Many applications of computer graphics or computer simulated environments require spatial or proximity relationships between objects. In particular, dynamic simulation, haptic rendering, surgical simulation, robot motion planning, virtual prototyping, and computer games often need to perform different proximity queries at interactive rates. The set of queries include collision detection, intersection, closest point computation, minimum separation distance, penetration depth, and contact points and normals. Algorithms to perform different queries have been well studied in computer graphics, virtual environments, robotics and computational geometry. Most of the current approaches involve considerable pre-processing and therefore are not fast enough for deformable models. Furthermore, no good algorithms are known for penetration depth computation between general, non-convex models.

We present a novel approach to perform all the proximity queries between rigid and deformable models using graphics hardware acceleration. Our algorithm localizes potential interactions using object-space techniques, point-samples the region, and then uses polygon rasterization hardware to compute object intersections, closest points, and the distance field and its gradients.

The main features of our approach include a unified framework for all proximity queries, applicability to non-convex polygonal models, computational efficiency allowing interactive queries on current PCs, robustness in terms of not dealing with any special-cases or degeneracies, and portability across various CPU/graphics combinations. A user-specified error threshold for pixel point sampling density and distance approximation governs the accuracy of the overall approach. Some of the novel features of our approach include:

- Improved and efficient construction of distance meshes used to compute 3D Voronoi diagrams accelerated with graphics hardware.
- Site culling algorithms and distance mesh culling for increased performance of Voronoi computation.

- Improved graphics hardware acceleration of computing the intersection between two, possibly non-convex, polygonal objects, over an entire volume.
- Improved algorithm for computing 3D image-space intersections that handles both inter-object and self-collisions.
- Computation of the gradients of the distance field using graphics hardware.

We have implemented our algorithm on various hardware configurations, and demonstrate its performance to compute different queries between rigid and dynamically deforming polygonal objects. Our approach is well suited for computing proximity query information needed for collision responses between dynamic deformable models. The use of graphics hardware allows us to perform different queries at interactive rates on complex deformable models. Moreover, it is relatively simple to implement all these queries in a robust manner. Over the last decade, the graphics processors (GPUs) processing power has been progressing at a rate faster than the CPUs and this will result in handling even more complex scenarios at interactive rates.

2. Related Work

Algorithms for computing collisions, intersections, and minimum separation distances have been extensively researched. Many are restricted to convex objects [Cameron 97, Ehmann01, Gilbert88, Lin91, Mirtich98] or are based on hierarchical bounding-volume or spatial data structures that require considerable precomputation and are best suited for rigid geometry [Hubbard93, Quinlan94, Gottschalk96, Johnson98, Klosowski98]. Some algorithms handle dynamically deforming geometry by assuming that motion is expressed as a closed form function of time [Snyder93] or by using very specialized algorithms [Baraff92]. In our approach, we emphasize the handling of non-convex, dynamically deformable objects with no precomputation or knowledge of object motions. In addition, we obtain computational complexity that grows linearly with geometric complexity for a fixed error tolerance and contact scenario.

As compared to collision detection and separation distance computation, there is relatively little work on penetration depth computation. Penetration depth is typically defined as the minimum translational distance needed to separate two objects. We define it with respect to a point as the minimum translational distance and direction needed to separate a penetrating point from an object's interior. Dobkin et al. have presented an algorithm to compute the intersection depth of convex polytopes, though no practical implementation is known [Dobkin93]. Cameron has presented a practical algorithm that computes an approximate depth for convex polytopes [Cameron97]. No practical algorithms are known for general, non-convex polyhedra.

Our algorithm relies on the computation of discretized distance fields and graphics hardware-accelerated geometric computation. Distance fields - scalar fields that specify minimum distance to a shape for all points in the field - have been used for many applications in graphics, robotics and manufacturing [Frisken00, Fisher01]. Common algorithms for distance field computation are based on level sets [Sethian96] or adaptive techniques [Frisken00]. However, they either require static geometry, extensive preprocessing, or lack tight error bounds. Graphics hardware has been used to accelerate a number of geometric computations, such as visualization of constructive solid geometry models [Goldfeather89], cross-sections and interferences [Rossignac92], and computation of the Minkowki sum [Kaul92]. However, these only compute intersections, not distance-related queries. Algorithms also exist for motion planning using graphics hardware acceleration and distance fields [Kimmel98, Lengyel90, Pisula00]. More recently, an algorithm has been proposed to compute generalized Voronoi diagrams and distance fields using graphics hardware [Hoff99]. Its application to motion planning was presented in [Pisula00]. Also, proximity queries accelerated using graphics hardware was presented in [Hoff01], but it was restricted to 2D and its extension to 3D was not obvious.

2.1 Voronoi and Distance field Computation

In [Hoff99], they present an algorithm for computing approximate 2D and 3D generalized Voronoi diagrams for polygonal objects with a variety of distance metrics. The representation is in the form of a discretized regular grid of sample points (images) across a 2D slice. A 3D Voronoi diagram is composed of a sequence of these slices across the volume to form a regular 3D grid. At each grid point, the ID of the nearest site and its associated distance is stored. They accelerate a brute-force algorithm using graphics hardware.

Instead of relying on a distance evaluation between a point and a Voronoi site, a polygonal distance mesh is constructed so that when rendered it computes the correct distance value as the Z-coordinate. If these distance meshes are rendered for each site with Z-buffer visibility enabled, the correct comparisons and updates will also be performed. This reduces the problem to finding a polygonal mesh approximation of a 2D slice of the distance function. In 3D, the distance mesh must approximate a 2D slice of the 3D domain.

Their 3D implementation simply used a coarse regular grid with direct distance evaluations at each grid point. This often required over-meshing, inefficient direct distance evaluations at grid points, and did not take advantage of the inherent symmetry in the functions being approximated. In addition, this approximation did not provide a tight bound on the approximation and the computation times were on the order of minutes to hours for high resolution Voronoi diagrams of complex models.

We extend the 3D distance mesh ideas and formulate a very fast and efficient bounded error approximation without requiring any lookup tables or complex data structures. In addition, we present methods for greatly accelerating the distance evaluations through culling techniques.

2.2 2D Proximity Queries using Graphics HW

In [Hoff01], they presented an approach using the graphics hardware based Voronoi computation for performing more general proximity queries, such as those needed in computing collision responses in a dynamics simulation. This paper focused on the interactions between 2D, possibly non-convex, polygonal objects only, but illustrated the potential for having a unified framework for a wide range of proximity queries. Many of the queries supported are particularly difficult for object-space algorithms, such as computing intersections, penetrating points, and penetration depths and directions. They used image-space techniques for performing these queries that were accelerated using graphics hardware. The core operations were based on queries into the Voronoi diagram. They presented a pipeline that allowed load balancing between CPU and graphics subsystems by first incorporating an object-space geometric localization phase to restrict the area over which the image-space phase must be performed.

Through improvements in the Voronoi diagram computation, we have extended this work into 3D. Many additional optimizations were necessary to make this run well in practice, including: faster and efficient distance meshing with bounded error, conservative Voronoi site culling, and making the queries symmetric (query A w.r.t. B is the same as B w.r.t. A). In addition, we constructed a specialized algorithm for computing 3D intersections efficiently. Previously in [Hoff01] for 2D, they relied on pixel overwrite to find intersection points. For 3D, we used a parity based strategy similar to operations used in graphics hardware-accelerated visualization of CSG operations and shadow volumes.

3. Overview of Our Approach

Given a collection of closed 3D polygonal objects, we perform coarse geometric localization to find rectangular regions of space (axis-aligned bounding boxes) that contain either potential intersections or closest feature pairs between objects. We uniformly point-sample these regions and use polygon rasterization hardware to compute object intersections, closest points, and the distance field. The distance field and its gradient vector field provide the distance and direction to the nearest feature for each point in the localized region, which gives the contact normals, minimum separation distances, or penetration depths. Our core algorithm computes the proximity information between two 3D, possibly non-convex, polygonal objects. Higher-order curved surfaces are tessellated into polygons with bounded distance deviation error. In our hybrid approach, there are two top-level operations:

(1) Geometric object-space operations to coarsely localize potential intersection regions or closest features

(2) Image-space operations using graphics hardware to compute the proximity information in the localized regions

Most of our improvements center around Voronoi and distance field computation since it is by far the most costly operation and is the most demanding of the graphics hardware. Load balancing between CPU and graphics subsystems is achieved by varying the coarseness of the object-space localization and by using object-space culling strategies. Tighter localized regions result in fewer pixels and a smaller bound on the maximum distance needed for Voronoi computation, thus reducing the fill and geometry loads on the graphics pipeline. We can also balance the load between these two main stages of the graphics pipeline by shifting the distance error tolerance in the Voronoi computation between fill and geometry: increasing the pixel resolution decreases the distance mesh resolution and vice versa. The main parts of the proximity query pipeline are shown in the following figure:

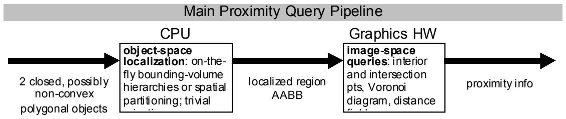

Main Proximity Query Pipeline
CPU
Graphics HW
2 closed, possibly non-convex polygonal objects
object-space localization: on-the-fly bounding-volume hierarchies or spatial partitioning; trivial
localized region AABB
image-space queries: interior and intersection pts, Voronoi diagram, distance
proximity info

Figure 1: The proximity query pipeline is composed of two main stages: geometric localization and image-space queries. The most complex queries are performed by graphics hardware. Each stage can be varied to balance the load between CPU and graphics subsystems.

4. Object-space Geometric Localization

The image-based queries operate on a uniform 3D grid of sample points in regions of space containing potential interactions. The graphics hardware pixel framebuffer is used as a 2D slice of the grid and the proximity queries become pixel operations, therefore the performance varies dramatically with the pixel resolution. To avoid excessive load, a geometric localization step is used to localize regions of potential interaction or as a trivial rejection stage. This hybrid geometry/image-based approach helps balance the load between the CPU and graphics subsystems, giving us portability between different workstations with varying performance characteristics. More sophisticated geometric techniques, to tightly localize potential intersections or closest feature pairs, dramatically reduce the graphics pipeline overhead, but increases the CPU usage and the complexity of the algorithm. We use coarse fixed-height bounding-volume hierarchies to achieve this balance between speed and complexity, and between CPU and graphics usage.

There are many general and efficient algorithms available for localizing geometry based on bounding-volume hierarchies [Gottschalk96, Hubbard93, Johnson98, Quinlan94]. However, for exact collision detection these algorithms typically perform well only on static geometry where the hierarchy can be precomputed. In order to handle dynamic deformable geometry with no precomputation, we use coarse levels for efficient trivial rejection and obtain reasonable geometric localization. In addition, we perform lazy evaluation of relevant portions of the hierarchies while performing the collision or distance query. A subtree rooted at a particular node is only computed if its children need to be visited during the query traversal. The trees are destroyed after every proximity query, and no actual tree data structures are required since the child nodes are recursively passed to the query routine. A maximum height of each object tree is used to balance the CPU and graphics load. Similar algorithms can be constructed using spatial partitioning rather than bounding-volume hierarchies. Since the resulting localized region needs to be rectangular (an axis-aligned cube) to allow simple use of the graphics hardware, we use a dynamically constructed AABB-tree. With a fixed number (depth of the tree) of linear passes over the geometry we obtain reasonable localization.

The typical proximity query is between two objects at a time. However, it is possible to perform many simultaneous queries for all objects in an N-body simulation. We could perform the proximity queries for all objects with one image-space query by using a localized region that encloses the entire scene. This may be more efficient in cases when the objects are densely packed with many complex contacts throughout the space containing the objects. For example, in a dense rigid body simulation where many objects are interacting simultaneously (e.g. an asteroid field), a single image-space query over the entire space may be more appropriate (localized region is the world bounding box). In addition, as the computational power of graphics systems continues to overtake the general CPU power, coarser and simpler localization will be favored.

The geometric localization step may often result in multiple disconnected regions on each object. In these cases, the proximity query must be repeated for each localized region. Geometric localization for intersecting and nearest features can be found by using existing bounding-volume or spatial partitioning approaches that act on object boundaries, but finding localized regions around volume intersections requires a specialized algorithm. At each step of refinement, the parent bounding box must fully contain the volume intersection. This can be accomplished by first starting with the intersection of the top-level object bounding boxes. This intersection box will surely contain the intersection volume. Now we can refine this localization by computing the bounding box of the portion of each object that lies in the current box. We then repeat the process on the intersection of these two boxes which is also guaranteed to contain the intersection volume.

5. Image-space Proximity Queries

The proximity queries are simplified using uniform point sampling inside an axis-aligned bounding box (localized region) and accelerated with graphics hardware. This image-space approach helps decouple the objects' geometric complexity from the computational complexity for a specified error tolerance. We point-sample the space containing the geometry within the localized regions with a uniform rectangular 3D grid and perform the queries on this volumetric representation using graphics hardware acceleration. The image-based queries include computing intersections between objects, computing the distance field of an object boundary, and computing the gradient of the distance field. Variations of these basic operations are used to perform the remaining queries. The basic pipeline is shown in Figure 2.

The 3D image-space queries avoid excessive data handling when processing the entire volume of the localized region. Each query must be performed over the uniform 3D grid, one 2D slice at a time. The application query information is sent to the application as it is processed slice-by-slice to avoid processing and storing the entire 3D image. In addition, many of the queries have been made symmetric to avoid a second pass as needed in the previous work.

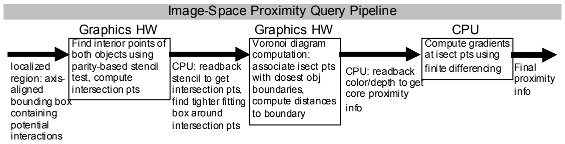


Image-Space Proximity Query Pipeline
Graphics HW
Find interior points of both objects using parity-based stencil test, compute intersection pts
Graphics HW
Voronoi diagram computation: associate isect pts with closest obj boundaries, compute distances to boundary
CPU
Compute gradients at isect pts using finite differencing
localized region: axis-aligned bounding box containing potential interactions
CPU: readback stencil to get intersection pts, find tighter fitting box around intersection pts
CPU: readback color/depth to get core proximity info
Final proximity info

Figure 2: The most computationally intensive tasks are performed by the graphics hardware. These stages are also the most difficult for geometric object-space approaches. We accelerate simple brute-force image-space solutions using graphics hardware to obtain interactive performance on complex models with no precomputation

5.1 Intersections

We compute intersection points on a 2D slice by performing a parity test, as is often used in shadow volumes and CSG rendering, using graphics hardware stencil operations [Crow77, Rossignac92]. In order to find intersections, we must first be able to identify sample points that are inside the object. The set of sample points that are inside both objects form the intersection points between the objects. We then describe another generalized strategy that can handle intersections between multiple objects simultaneously along with the more complex self-intersections.

For any closed object, we can determine if a point is inside the object by shooting a ray from the point in any direction and counting the number of times the object's surface is crossed. If the count is even, the point is outside of the object; if odd, the point is inside. We can simultaneously determine which sample points on a 2D planar slice are interior points by projecting all of the geometry on one side of the plane onto the plane and counting the number of times pixels are overwritten. This computation is performed using the graphics hardware through an odd-even parity test for rendered geometry clipped by the plane and projected onto the plane. Each time a pixel is overwritten the parity bit is flipped. Pixels whose stencil bit is set to 1 represent points on the slice that are inside the object. Initially the stencil buffer is initialized to 0.

5.1.1 Incremental Update and Bucket Sorting

For a single slice, this computation requires rendering all of the geometry on one side of the plane (clipped by the plane). However, this is inefficient for evaluating interior points on many slices swept through our 3D localization box. We improve efficiency by performing a plane sweep and updating the stencil buffer incrementally. For each slice, we only render the geometry between the current slice and the previous slice.

This incremental update improves the running time dramatically since on average the entire model is only drawn once! As opposed to the single slice approach where the entire model to one side of the slice had to be drawn for each successive slice. We can obtain even greater performance by first sorting the geometric primitives along the Z-axis by their minimum Z-values. A general sort would require O(n log n) time complexity. We obtain expected O(n) complexity by performing a bucket sort by using the slab positions as the buckets. With one pass through the geometry, we can assign each primitive to a bucket by its minimum Z-value. We maintain a list of currently active geometry for each slab. For each subsequent slab we add geometry to the list from the associated bucket. Geometry is removed from the list by checking if the old primitives' max Z-value is less than the current slab NearZ (swept past the primitives). This also dramatically improves performance with very little extra complexity or data. We avoid having to search for geometry that intersects the current slab. In addition, there is no need to add geometry to the buckets that lies outside of the XY min/max box. In practice, very little geometry has to be processed for the interior computation.

In order to find the intersection between two objects, we compute the interior of both objects inside of the localized region one slice at a time. The interior of both objects is encoded in a different bit of the stencil buffer. The set of points with both bits set are intersection points since they are interior to both objects. To actually extract these points, we must read the stencil image and search for the pixels with the appropriate value (a value of 3 from the 1st and 2nd least significant bits being set). These points must then be transformed from pixel-space into object-space.

5.1.2 Complexity and Error Analysis

Our new algorithm for intersection computation for 3D non-convex objects is simpler as compared to the 2D intersection computation algorithm presented in [Hoff01]. The major weakness of finding overwritten pixels between two non-convex polygons, was that they had to be triangulated in order to be rendered. This was the dominant part of the intersection computation since it was worst case O(n log n) rather than O(n). However, the expected running time of most triangulation algorithms is usually close to linear. In 3D, we only require the O(n) complexity where n is the number of primitives. The actual running time varies most dramatically with the ratio of the size of the localized region over the error tolerance, and is largely independent of the geometric complexity. More complex forms of contact do not result in increased running times unless the size of the localized region is increased dramatically or the error tolerance is reduced. These cases are difficult to analyze since they vary dramatically with the object configurations. More sophisticated geometric localization will reduce performance variations.

The complexity of rendering objects grows linearly with respect to the number of primitives for a fixed pixel resolution. Computing intersections geometrically between two polygon boundaries is worst case O(n^2) since all primitives could intersect each other. The complexity of our algorithm is O(n) where n is the number of primitives. The hierarchical geometric localization step is also O(n) since the maximum depth of the tree is held constant. This tree depth balances the load between the CPU and graphics subsystems.

Similarly to the 2D case, the error in the interior and intersection computation is related to the pixel error in scan-conversion. The actual interior regions will never be off by more than half of the length of the diagonal of a pixel's rectangular cell (the error tolerance). The error tolerance has a dramatic effect on the number of pixels that have to be processed. When reduced error tolerances are required, better geometric object-space localization must be employed to reduce the load on the graphics subsystem. Furthermore, we can also balance the loads between geometry and fill stages of the graphics pipeline by trading off error in the pixel resolution and the distance mesh granularity.

Incorrect intersection parity resulting from pixel sample points lying exactly on tangent points to the object surface are avoided through correct minimum-based triangle rasterization as described in [Rossignac92]: either the crossing will be counted twice or not at all.

5.1.3 Multiple Objects and Self-Collisions

We can modify the intersection routine to handle self-collisions and multiple objects with very little modification to the previous algorithm. The modification adds the complexity of having to distinguish front and back faces for polygons in each slab for a parallel projection and has the slight restriction of only handling the intersection of at most 255 simultaneous volumes (limit of 8-bit stencil buffer).

Instead of finding the interior of both objects separately and then finding their common intersection, we can simply finding the intersection directly using the geometry from both objects simultaneously using the classic parity test used in the shadow volume algorithm. Since we want to know if a point is inside two volumes simultaneously, a ray emanating from a query point must have exited at least two more volumes than it has entered.

Instead of simply flipping a bit each time a boundary is crossed (front or back facing), starting with a stencil counter initialized to zero, we increment the counter each time a volume is exited (a back face is rendered) and decrement the counter whenever a volume is entered (front face is rendered). The counter will indicate the number of objects containing the point. We are interested in the intersection points, so the counter must at least be 2. We simply modify our existing approach of rendering slabs to perform this count instead. We must classify all object faces for each slab as front or back facing with respect to a parallel projection. Since all object triangles are handled together, we can handle more than 2 objects and we can also find self-intersections of a single object. Stencil counts of 2 or greater indicates a point that is in the intersection of at least one pair of objects or an object with itself.

5.2 Distance Field Computation

We use the algorithm presented in [Hoff99] for constructing generalized Voronoi diagrams using graphics hardware for 3D polygonal objects. This approach computes an image-based representation of the Voronoi diagram in both the color and the depth buffers for one 2D slice of the 3D volume at a time. A pixel's color identifies the polygon feature (vertex or edge) that is closest to the slice pixel's sample points; its depth value corresponds to the distance to the nearest feature. The depth buffer is an image-based representation of the distance field of the object boundaries. The distance field is computed by rendering 3D bounded-error polygonal mesh approximations of a 2D planar slice of the distance function where the depth of the rendered mesh at a particular pixel location corresponds to the distance to the nearest geometric feature.

The goal in constructing a distance mesh is to find a piecewise linear approximation across a 2D planar domain of a Voronoi site's 3D scalar distance function. The distance to a site from a point (x,y,z) is defined as $D(x,y,z)$. The function we are interested in approximating is for a 2D planar slice $z=Z_{slice}$. So we wish approximate the 2D scalar function $D(x,y, Z_{slice})$, where Z_{slice} is a constant for any particular slice, such that the approximation D' and actual distance function D never differ by more than the user-specified distance error. In addition, the domain across the slice is restricted to a 2D window and the range of the function is restricted to $z\in[0,MaxDist]$. The shape of the distance mesh for a 3D point is one sheet of a hyperboloid of two sheets; for a line, an elliptical cone; and for a plane, another plane.

In [Hoff99], distance meshes were constructed using lookup tables. We construct error-bounded polygonal mesh approximations of a 2D planar slice of a primitives distance function at run-time with no precomputation at faster rates than the algorithm based on lookup tables. We solve for the mesh stepsizes needed to maintain the desired error threshold while taking advantage of symmetry. We attempt to actual obtain the desired error to make the meshes as coarse as possible for rendering efficiency. In addition, we only construct geometry that lies within the slice window.

For computing distance fields for proximity queries, we obtain higher performance than the generalized Voronoi diagram computation because of the localized regions. In the case of computing distance fields for proximity queries, the localized regions always contain the geometry that is in potential contact or that contains the closest features. The farthest away points on two objects can be is in opposite corners of the localized region box. So the maximum distance we need to construct distance meshes for is half of the diagonal length of the box. Reducing the maximum distance results in the greatest speedups in Voronoi computation since it reduces geometry and fill by reducing the overall extent of the distance meshes, and the smaller bound allows the objects to be easily culled if they are too far from the localized region thus avoiding distance mesh construction completely. In addition, the distance mesh generation routines attempt to minimize the number of primitives drawn by constructing a mesh that is as coarse as possible while staying within the specified error bound (the error bound is tight, this deviation can actually be measured for various places in the mesh approximation) and by only generating primitives that are inside the localized region bounding box. In addition, in many proximity queries we can further reduce the maximum distance needed when we only want intersection or closest points near the boundaries of the object.

5.3 Gradient of the Distance Field

We compute the gradient of the distance field at pixel locations by using central differences in all three principal axis directions. In practice, this simple approach gives reasonable results even with the distance error and lack of C^1 or higher continuity in the polygonal distance mesh approximations used to compute the distance field. Gradients are computed in software for selected points after reading back the distance values. If the entire gradient field is desired, we could accelerate the computation using multi-pass rendering or pixel shading operations.

The most difficult problem in computing the gradient is in handling discontinuities and boundaries in the distance field. There are three types of discontinuities that occur: across Voronoi boundaries, across Voronoi sites, and at the boundaries of the grid. In each case, the support of the finite differencing "kernel" has to cross a discontinuity and gives an incorrect gradient. A more robust method is shown in the fast marching methods in [Sethian96]. He solves for a distance value at an unknown point using an implicit method based on the fact that at least one adjacent distance value must be known and does not cross a discontinuity, and that for the nearest Euclidean distance metric the magnitude of the gradient must be 1 everywhere (except at the discontinuities). We use the same method by just using the one-sided difference in each direction that will result in a gradient whose magnitude is 1 (choose the adjacent value in each direction that has the maximum difference). Adjacent distance values that cross a discontinuity will not be chosen. An alternative, but slightly more complex, strategy is to compute the gradients of the continuous distance meshes directly.

By directly encoding gradients at distance mesh vertices, we can use the linear interpolation of polygon rasterization to compute gradients at all pixels. Since we would be linearly interpolating a gradient, this gives us a higher order interpolation than central differencing of adjacent distance values. This is comparable to the difference between Gouraud and Phong interpolation (the first linearly interpolates color values across a polygon, the second linearly interpolates the surface normal for per-pixel lighting calculations). In addition, the gradient is much simpler if computed only for a single site at a time during distance mesh construction. We need only provide the direction to the nearest point on the site at each distance mesh vertex. The main difficulty with this approach is in the encoding of the gradient for rapid computation by graphics hardware.

This approach as some difficulties due to limitations of graphics hardware framebuffer precision. There are a number of ways we can interpolate the gradient information. The simplest is to encode the signed normalized components into the 8-bit RGB color values at each vertex (using hardware scale and bias operations for sign). The linear interpolation would give the correct results to 8-bits of precision. This approach introduces quantization error when encoding and additional error during interpolation. Using 3D texture coordinates, high precision encoding and interpolation is obtained. However, the resulting per-pixel texture coordinates are still quantized to low precision RGB values in the framebuffer. The texture-mapping function would simply be the identity mapping. We are interpolating (s,t,r) gradient values and we want those values directly at each pixel. The graphics hardware does not allow higher precision intermediate results for multi-pass operations. However, the texture-mapping method has the advantage of only introducing significant error at the final stage; encoding and interpolation are done at floating point precision. Also, the signs will be correctly handled without any additional scaling or biasing. However, we also have no simple way of performing the identity map. We must use a 1D texture that maps [-1,1] to [0,255], but this can only be applied to one texture component at a time. This would require three passes in order to transform (s,t,r) into RGB values. A less efficient approach would involve the use of a 3D texture map. Current pixel-level programmable graphics hardware may provide a simpler and more efficient way to handle this mapping.

5.4 Other Proximity Queries

We use the basic operations of computing interior points, intersections, the distance field, and the gradient of the distance field to perform the other proximity queries mentioned in section 1.

Penetration Depth and Direction: For a point on object A that is penetrating object B, we define the penetration depth and direction for the point as the distance and direction to the nearest feature on B. This information is provided directly from the distance field and its gradient computed at the penetrating point. Penetrating points are found using the intersection operation. Intersection points are associated with each object based on the Voronoi diagram's color buffer that indicates the closest object to each point. Contact points and Normals are computed in the same way. Approximate contact points result from the objects slightly penetrating each other.

Closest Point: We find the point on object A that is closest to object B by first geometrically localizing potential closest feature regions (one bounding box on each object) using some hierarchical approach. We then compute the distance field of object B and the interior points of A in A's localized region (gives us minimum distance to B for all points in A in A's localized region). We then search these points to find the one with the smallest distance value. This point will be the point on A that is closest to B. This process has to be repeated for B with respect to A. This requires two passes, but the interior points and the distance field needs to only be computed once for each object.

Separation Distance and Direction: We find the minimum separation distance and direction between two objects A and B by first computing the closest point on A to B and vice versa. Ideally, we find the closest point on B to A from the distance value and gradient at the closest point on A to B, but the amplification of errors over the greater distance may cause problems. The distance between these two closest points is the separation distance and the line segment between them gives the separation direction.

6 Performance

We tested the system performance in both rigid and deformable body dynamic simulations on a several different hardware configurations. In the rigid body cases, we measured the performance of the system in computing proximity query information needed for computing a penalty-based collision response. In these cases, only shallow penetration is allowed since the objects bounce off of each other. For the deformable cases, we perform only the proximity queries without collision response to show the worst case of computing proximity information for many deep simultaneous contact scenarios with dynamically deforming geometry. We tried to choose three hardware configurations that would reflect variations in balance between CPU and graphics computational power:

(1) Pentium-4 1.8Ghz with GeForce3 Ti500 graphics: fast CPU and fast graphics

(2) 1 graphics pipe and 1 300Mhz MIPS R12000 processor of an SGI Reality Monster with InfiniteReality2 graphics: slower CPU and fast graphics

(3) PentiumIII-750Mhz laptop with ATI Rage Pro LT: fast CPU and slow graphics.

Because of the ability to balance the load between the CPU and graphics subsystems and between stages of the graphics pipeline, we are able to achieve interactive performance on all configurations. In most cases, we only needed very simple one-level geometric localization (intersection of top-level axis-aligned bounding boxes). Most of the balancing was between stages of the graphics pipeline (much of the geometry stage on older graphics systems was performed on the CPU: before hardware T&L). We also show the effects of the varying the distance threshold on system performance.

For performance evaluation, we implemented a rigid body simulator with collision response and a variety of deformable simulations without collision responses to allow more complex contact scenarios. The test scenarios vary from simple convex objects composed of around 2 thousand triangles with simple contact regions to non-convex objects with nearly 10,000 triangles with multiple complex overlapping and interlocking contact regions. The average query times are shown in Table 1. It is important to note that the query time is not growing because of the increase in geometric complexity, but rather because our more complex models are in more complex contact configurations.

The performance of our image-space query system depends more on the contact configuration than on the complexity of the objects. The distance error tolerance determines the point sample density across the contact volume. The density and the volume of the localized regions and the contact regions determine the number of pixels that have to be processed. If an insufficient level of geometric localization is used, the number of pixels to process may increase dramatically. The user must decide the appropriate amount of localization to properly balance the CPU/graphics load. In addition, the performance can be varied dramatically by the user-specified distance error tolerance. In Table 2, we show the effects on performance with a varying error tolerance.

Average Total Per-frame Proximity Query Times

Demo	#Tris	Isect Pts	GeForce3	IR2	Rage Pro
Cylinders	2000	513	12ms	61ms	45ms
Tori	5000	1412	71	262	257
Heart	8000	317	149	329	434
Rigid	15000	2537	313	1001	966

Table 1: Performance timings for dynamics simulations. The number of triangles, average number of intersection points, and average time to run proximity queries per frame is reported for error tolerance d (see Table2). Timing data was gather from three machines, a Pentium4 1.8GHz desktop with a 64Mb GeForce3, a SGI 300MHz R12000 with InfiniteReality2 graphics, and a Pentium-III 750Mhz laptop with ATI Rage Pro LT graphics.

Effects of Error Tolerance on Performance

Error	Isect Pts/Frame	GeForce3	IR2	Rage Pro LT
***d*/4**	89605	548ms	1701ms	2846ms
***d*/2**	11238	169	578	689
d	1413	71	262	257
2*d*	177	32	189	103
4*d*	22	15	56	40

Table 2: The effect on performance when changing the distance error tolerance *d*. The average number of intersection points per frame is also reported. We used proximity queries on the deformable tori demo. The error determines the number of pixels used in the image-based operations. Systems with low graphics performance are more directly affected by the choice of *d*; however, as the error is increased there is less dependence on graphics performance and the faster laptop CPU overtakes the InfiniteReality2 system.

Although we focused most of our efforts on handling deformable body proximity queries, our system is also applicable to rigid body queries. We use a penalty-based collision response that acts on individual point samples that approximate our object. These point samples arise from our image-space proximity queries. Particles are allowed to penetrate objects in penalty-based collision response computation. When a penetration is detected, a spring based restoring force, whose magnitude is proportional to penetration depth, is then applied to the particle until it has separated from the object. The measure of penetration is notoriously expensive to compute and limits the use of penalty-based techniques to mostly models decomposable into convex primitives. The generality and computational efficiency provided by our proximity query algorithms alleviates this problem.

7 Conclusion and Future Work

We have presented a hybrid geometry- and image-based algorithm for computing geometric proximity queries between two non-convex closed 3D polygonal objects using graphics hardware. This approach has a number of advantages over previous approaches. The unified framework allows us to compute all the queries, including penetration depth and direction and contact normals. Furthermore, it involves no precomputation and handles non-convex objects; as a result, it is also applicable to dynamic or deformable geometric primitives. In practice, we have found the algorithm to be simple to implement (as compared to similarly robust geometric algorithms), quite robust, fast (considering the complexity of the queries), and very flexible. We have developed an interactive system that shows proximity queries computed between 3D dynamic deformable objects to illustrate the effectiveness of our approach.

8 Acknowledgements

This research has been supported in part by ARO Contract DAAG55-98-1-0322, DOE ASCII Grant, NSF Grants NSG-9876914, NSF DMI-9900157 and NSF IIS-982167, ONR Contracts N00014-01-1-0067 and N00014-01-1-0496 and Intel

References

[Baraff92] D. Baraff, *Dynamic Simulation of Non-Penetrating Rigid Bodies*. Ph.D. Thesis, Dep of Comp. Sci., Cornell University, March 1992

[Cameron97] S. Cameron, *Enhancing GJK: Computing Minimum and Penetration Distance between Convex Polyhedra*. International Conference on Robotics and Automation, 3112-3117, 1997

[Crow77] F. Crow, *Shadow Algorithms for Computer Graphics*. SIGGRAPH 77.

[Dobkin93] D. Dobkin, J. Hershberger, D. Kirkpatrick, S. Suri, *Computing the Intersection Depth of Polyhedra. Algorithmica*, 9(6), 518-533, 1993

[Ehmann01] S. Ehmann and M. Lin. *Accurate and Fast Proximity Queries between Polyhedra Using Surface Decomposition*. Eurographics 2001

[Fisher01] S. Fisher and M. Lin. *Fast Penetration Depth Estimation for Elastic Bodies Using Deformed Distance Fields*. Proc. Intl. Conf. on Intelligent Robots and Systems, 2001

[Frisken00] S. Frisken, R. N. Perry, A. P. Rockwood, T. R. Jones, *Adaptively Sampled Distance Fields: A General Representation of Shapes for Computer Graphics*. SIGGRAPH00, 249-254, July 2000

[Gilbert88] E. G. Gilbert, D. W. Johnson, S.S. Keerthi. *A Fast Procedure for Computing the Distance Between Objects in Three-Dimensional Space*. IEEE J. Robotics and Automation, RA(4): 193-203, 1988

[Goldfeather89] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. *Near Real-time CSG Rendering Using Tree Normalization and Geometric*

Pruning. IEEE Computer Graphics and Applications, 9(3):20-28, May 1989

[Gottschalk96] S. Gottschalk, M. C. Lin, D. Manocha, *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*. SIGGRAPH 96, 171-180, 1996

[Hoff99] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*. SIGGRAPH 99, 277-285, 1999

[Hoff01] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha. *Fast and Simple 2D Geometry Proximity Queries Using Graphics Hardware*. ACM Symposium on Interactive 3D Graphics, 2001

[Hubbard93] P. M. Hubbard, *Interactive Collision Detection*. IEEE Symposium on Research Frontiers in Virtual Reality. 24-31, 1993

[Kaul92] A. Kaul and J. Rossignac, *Solid-interpolating Deformations: Construction and Animation of PIPs*, Computer and Graphics, vol 16, 107-116, 1992

[Kimmel98] R. Kimmel, N. Kiryati, A. Bruckstein, *Multi-Valued Distance Maps for Motion Planning on Surfaces with Moving Obstacles*. IEEE Transactions on Robotics and Automation, vol 14: 427-438, 1998

[Klosowski98] J. Klosowski, M. Held, J. Mitchell, K. Zikan, H. Sowizral. *Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs*. IEEE Trans. Vis. Comp. Graph, 4(1):21-36, 1998

[Johnson98] D. Johnson, E. Cohen, *A Framework for Efficient Minimum Distance Computation*, IEEE Conf. On Robotics and Animation, 3678-3683, 1998

[Lengyel90] J. Lengyel, M. Reichert, B.R. Donald, and D.P. Greenberg. *Real-time Robot Motion Planning Using Rasterizing Computer Graphics Hardware*. Computer Graphics (SIGGRAPH 90 Proc.), vol. 24, pgs 327-335, Aug 1990

[Lin91] M. Lin, J. Canny. *Efficient Algorithms for Incremental Distance Computation*. IEEE Transactions on Robotics and Automation, 1991

[Mirtich96] B. Mirtich, *Impulse-Based Dynamic Simulation of Rigid Body Systems*. Ph.D. Thesis, University of California, Berkeley, Dec 1996

[Mirtich98] B. Mirtich, *V-Clip: Fast and Robust Polyhedral Collision Detection*. ACM Trans. on Graph, 17(3):177-208, 1998

[Pisula00] C. Pisula, K. Hoff, M. Lin, and D. Manocha. *Randomized Path Planning for a Rigid Body Based on Hardware Accelerated Voronoi Sampling*. Proc. of Workshop on Algorithmic Foundations of Robotics, 2000

[Quinlan94] S. Quinlan, *Efficient Distance Computation between Non-Convex Objects*. International Conf. on Robotics and Automation, 3324-3329, 1994

[Rossignac92] J. Rossignac, A. Megahed, and B. Schneider. *Interactive Inspection of Solids: Cross-sections and Interferences*. SIGGRAPH 92, 26, 353-360, July 1992

[Sethian96] J. Sethian, *Level Set Methods*, Cambridge University Press, 1996

[Snyder93] J. Snyder, A. Woodbury, K. Fleischer, B. Currin, A. Barr, *Interval Methods for Multi-Point Collisions Between Time Dependent Curved Surfaces*. ACM Computer Graphics, 321-334, 1993

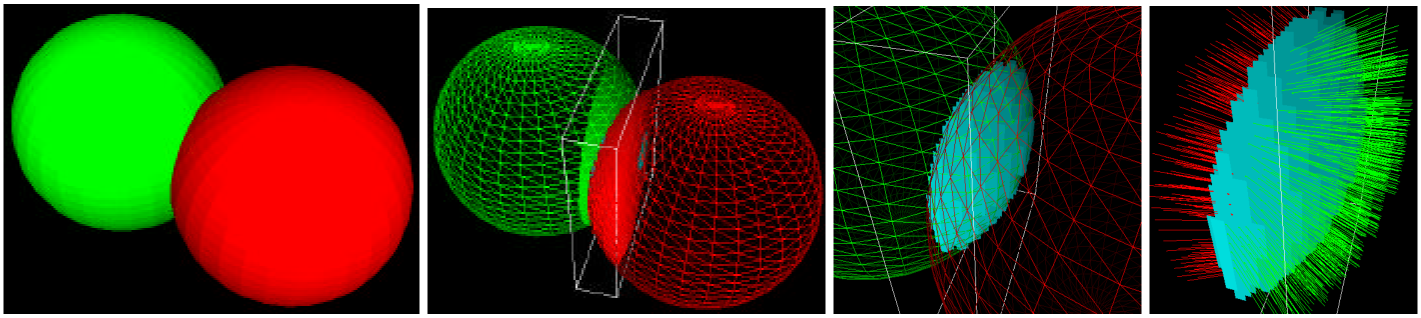


Plate 1 hybrid proximity query pipeline : Given two closed polygonal objects, a coarse object-space geometric localization step is performed to find an axis-aligned bounding box that contains a potential interaction (2). Inside the localized region, the lower-level image-space queries are performed. First the interior of each object is indentified using an incremental stencil parity test for a series of 2D slices across the volume (2). The set of point that are determined to lie in the interior of both objects form the intersection points between the objects (3). Then, the Voronoi diagram is computed inside a tighter region around the intersection points at the same resolution as the intersection resolution. The Voronoi diagram serves two purposes: associates intersection points with their closest object boundaries, and provides the distance field. The distance value at an intersection point gives the penetration depth, and the gradient gives the penetration direction.

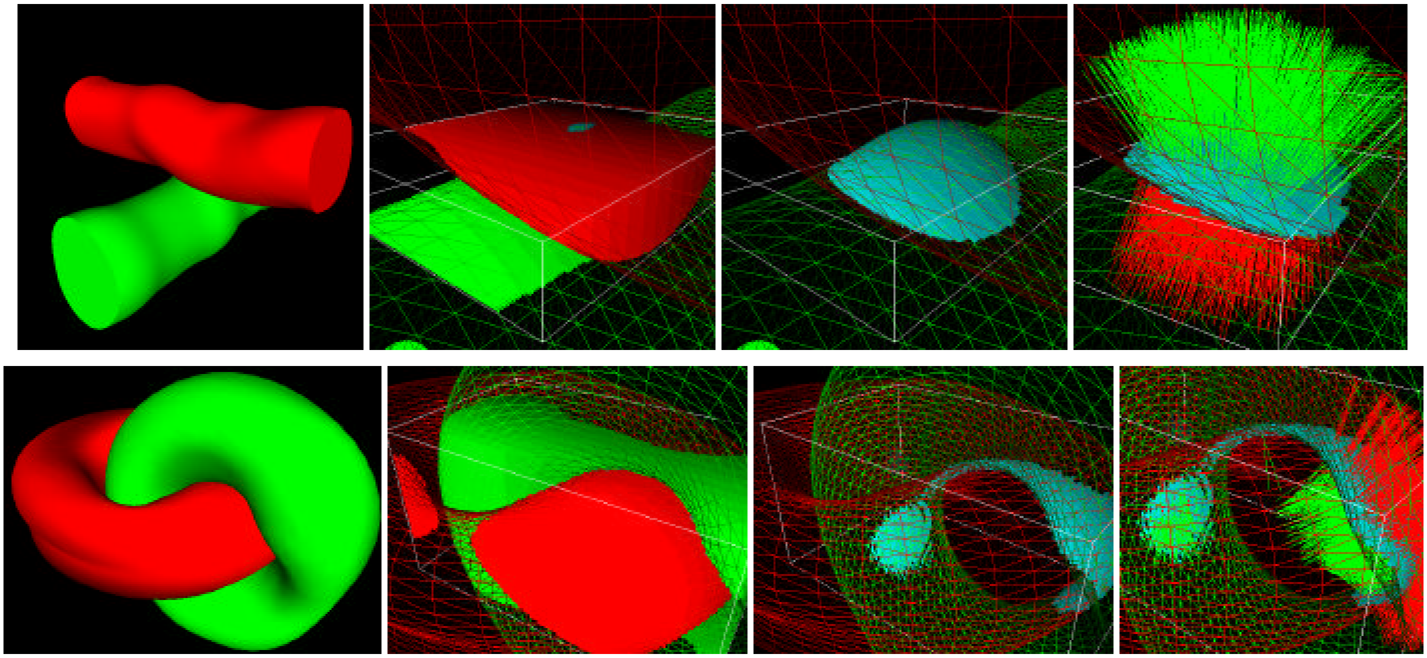


Plate 2 real-time dynamic deformable proximity queries: The same proximity query pipeline can be applied to dynamic deformable models where every vertex is assumed to change for every frame. The complex contacts between non-convex objects can result in disconnected intersection regions. Each cylinder model is composed of 2000 triangles and the average query time is 12ms for an average of 513 intersection points per query. The tori are composed of 5000 triangles and the query time is 71ms for 1412 intersection points. Each simulation performed at interactive rates on a Pentium4 1.8GHz desktop with a 64Mb GeForce3.

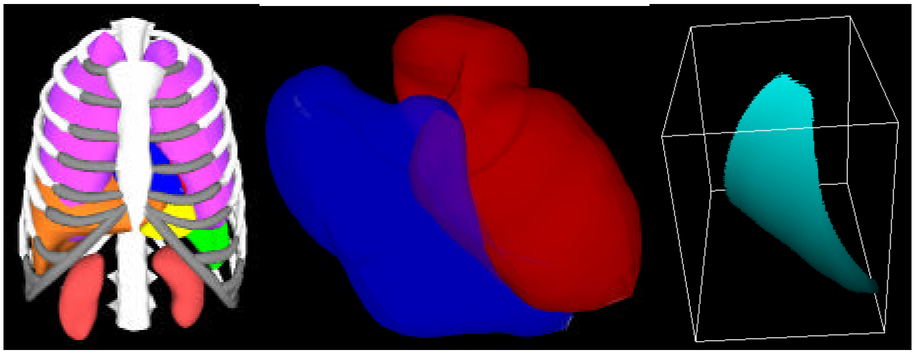


Plate 3 proximity queries on body heartbeat simulation: The proximity queries are used for path verification of the organs during a precomputed breathing simulation. Here we can see that the two ventricles are actually intersecting. The heart is composed of 8000 triangles and the average query time is 149ms for an average of 317 intersection points. This simulation performed at interactive rates on a Pentium4 1.8GHz desktop with a 64Mb GeForce3.

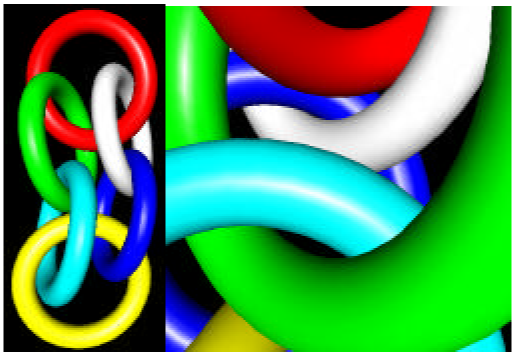


Plate 4 multiple complex contact scenario in an interactive rigid body simulation: Collision responses are computed using a penalty-based method that requires penetration depth computation. Each ring is composed of 2500 triangles, average query time is 313ms for 2537 intersection points.

Constraint-Based Motion Planning for Virtual Prototyping

Maxim Garber
Department of Computer Science
University of North Carolina at Chapel Hill
http://www.cs.unc.edu/∼garber
garber@cs.unc.edu

Ming C. Lin
Department of Computer Science
University of North Carolina at Chapel Hill
http://www.cs.unc.edu/∼lin
lin@cs.unc.edu

ABSTRACT

We present a novel framework for motion planning of rigid and articulated robots in complex, dynamic, 3D environments and demonstrate its application to virtual prototyping. Our approach transforms the motion planning problem into the simulation of a dynamical system in which the motion of each rigid robot is subject to the influence of virtual forces induced by geometric constraints. These constraints may enforce joint connectivity and angle limits for articulated robots, spatial relationships between multiple collaborative robots, or have a robot follow an estimated path to perform certain tasks in a sequence. Our algorithm works well in dynamic environments with moving obstacles and is applicable to challenging planning scenarios where multiple robots must move simultaneously to achieve a collision free path. We demonstrate its effectiveness for parts removal, automated car painting, and assembly line planning scenarios.

Categories and Subject Descriptors

I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling; I.6 [**Computing Methodologies**]: Simulation and Modeling

General Terms

Algorithms, Performance, Design, Reliability, Verification

Keywords

Virtual Environments and Prototypes, Manufacturing and Assembly Planning, Computational Support for New Manufacturing Technologies

1. INTRODUCTION

The problem of achieving design flexibility and manufacturing automation using virtual environments provides a new set of challenges for computer aided design and manufacturing. Automatically planning the motion of parts or objects, i.e. motion planning, can be a significant aid in a rapid prototyping environment. Examples of the tasks that can be assisted by motion planning include virtual assembly for design verification, parts removal for maintainability studies, path generation for automated painting, etc. Some of the commonly used computer-aided manufacturing and simulation packages like IGRIP from Delmia Inc., CimStation from Adept Technologies, PDMS from CADCENTRE, ProductVision from GE Corporate R & D, and THOR Arc Weld from AMROSE include computational support for motion planning. These packages are already used for pipe routing in plant design, arc welding of complex assemblies, spot welding of car bodies, and other uses of robots to automate the manufacturing processes. The planning tasks are typically characterized by geometric goal regions, a variety of mechanical constraints, and often a partially known environment with uncertainties and moving obstacles.

Main Results: In this paper, we present a new motion planning algorithm for virtual prototyping. Our algorithmic framework is inspired by constrained dynamics [26] in physically-based modeling. We transform the motion planning problem into a dynamical system simulation by treating each robot as a rigid body or a collection of rigid bodies moving under the influence of all types of constraint forces in the virtual prototyping environment. These may include constraints to enforce joint connectivity and angle limits for articulated robots, constraints to enforce a spatial relationship between multiple collaborative robots, constraints to avoid obstacles and self-collision, or constraints to have the robot follow an estimated path to perform certain tasks in a sequence.

Our constraint-based planning framework has the following characteristics:

- It can handle both static environments with complete geometric information or dynamic scenes with moving obstacles whose motion is not known a priori.
- It is applicable to both rigid and articulated robots of arbitrarily high degrees of freedom, as well as multiple collaborative agents.
- It allows specification of various types of geometric constraints.
- It runs in real time for modestly complex environments.

We demonstrate the effectiveness of our framework for the problem of virtual assembly and electronic prototyping with applications in assembly line planning, automated car painting, and maintainability studies.

Organization: The rest of the paper is organized as follows. In section 2, we briefly discuss the problem of motion planning and survey related work. Section 3 presents an overview of our planning framework. We describe the constraints in our framework and how they can be used to represent planning scenarios in section 4. Section 5 presents a detailed discussion of the method we have implemented to solve the geometric constraints. We demonstrate our

SM'02, June 17-21, 2002, Saarbrucken, Germany.
Copyright 2002 ACM 1-58113-506-8/02/0006 ...$5.00.

framework by applying it to several virtual prototyping problems in section 6.

2. RELATED WORK

We first introduce the terminology used in this paper, provide some background information on motion planning, and survey related literature.

2.1 Background and Terminology

We assume the robot(s) and obstacles are sets of closed and bounded geometry and whose locations at any time during the simulation are known and updated dynamically. The obstacles are free to move during the robot planning and motion execution stage, their motion acquired by sensory input is then fed back to the motion planner in real time. The robot $\mathcal{R}$ and a set of obstacles $\mathcal{O}$ move in a Euclidean space $\mathcal{W}$, called the *workspace*. The robot may be a free-flying rigid object or an articulated object. A free-flying object has no kinematic constraints that limit its motion. On the other hand, an articulated object $\mathcal{R}$ consists of several moving rigid parts $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_m$, called links, connected by joints. Each joint constrains the relative movements of the two links it connects.

The classic motion planning problem, also referred to as the Piano Mover's problem, can be stated as the following: Given a robot $\mathcal{R}$ and a workspace $\mathcal{W}$, find a path from an initial configuration $\mathcal{I}$ to a goal configuration $\mathcal{G}$, such that $\mathcal{R}$ never collides with any obstacle $\mathcal{O}_i \in \mathcal{O}$ in $\mathcal{W}$ along the path P, if such a path exists. The path P is a continuous sequence of positions and orientations of $\mathcal{R}$.

Configuration space or C-space [20] is a powerful concept for the planning problem. This formulation represents the robot as a point in an appropriate space, i.e. the robot's configuration space, and maps the obstacles in this space. This mapping transforms the motion planning problem for an object into the problem of planning the motion of a point in a higher dimensional space. For a rigid body, the configuration is specified by six coordinates, three determining the position with respect to some fixed reference point (the *origin*) on $\mathcal{R}$, and three determining the robot's orientation. An articulated object can be interpreted as a set of m moving rigid objects connected by joints. By convention, each joint affords a single degree of freedom, and a physical joint that allows more than one degree of freedom is represented by multiple joints at a single location, so that there can be more joints than links. With this convention, a configuration $\mathbf{cf}$ for an articulated body is specified by six coordinates determining the position and orientation of a given link, the *base link*, along with an additional coordinate for each joint.

2.2 Global vs. Local Planning Methods

There have been two major approaches to motion planning: global and local methods [19]. Global planning methods, including the first Roadmap Algorithm [4], PRM [15] and other geometric or "criticality-based" methods [10, 19], are guaranteed to find a complete path, if one exists, although they may take a long time computing it. Many of the global methods, with the exception to variants of PRM, have been applied with limited success for mostly lower-dimensional planning queries in static environments, due to high computational costs.

On the other hand, local methods such as artificial potential field methods [16], are usually fast, but are not guaranteed to find a path, even if one exists. Algorithms based on artificial potential field methods are widely used in many industrial applications [5, 23]. In a dynamic environment, where the motion of obstacles in the scene is not know a priori, it is difficult for global methods to compute the complete solution path in real time to avoid collision with the moving obstacles. For dynamic scenes, local, or reactive, planning techniques are often used. However, local methods have limitations as well. For example, potential field methods are known for their entrapment problems at local minima of the potential function.

Our method is an incremental construction of a roadmap, whose the curves locally satisfy all constraints imposed on the robot while staying maximally clear of nearby obstacles. At the same time, the framework can also take global geometric analysis into consideration while performing local planning, thus alleviating the local minima problem (Sec. 4.3.3).

2.3 Planning for Industrial Applications

The basic motion planning problem can be enhanced by adding a model of position uncertainty to extend it to motion planning in a partially known environment, often encountered among industrial applications. But, it remains basically unchanged if compliant motion is not allowed.

Many specialized motion planning algorithms have been developed for different applications, including part orientation and positioning [9, 11], assembly sequencing [8, 21, 13], sensor-based planning [22, 6, 7] and maintainability studies [1, 5].

2.4 Constraint Solving

Geometric constraint solving has been extensively studied in many different fields, such as CAD/CAM, molecular modeling, and theorem proving [2, 3, 17]. There are two basic strategies: *instance solvers* and *generic solvers*. Some of the common approaches include numerical algebraic techniques, graph-based algorithms, logical interference and term rewriting, symbolic algebraic solvers, and propagation methods [2]. Our approach borrows a combination of ideas from some of these techniques, and specializes them for motion planning.

The constraint solving problem for motion planning can be NP-hard for arbitrarily high degree-of-freedom manipulators. Since we are interested in real-time performance for dynamic scenes with moving obstacles and multiple robots, we consider the intended solution by inferring certain metric and topological properties of the planning problem as a dynamical system, and deduce a few heuristics that succeed with high probability under the assumptions of compatible constraints and temporal coherence. The details of our constraint solving approach will be given in Sec. 5.

3. FRAMEWORK OVERVIEW

3.1 Framework Goals

Our planning framework is targeted to enable rapid prototyping in a 3D CAD/CAM environment. This goal imposes several design requirements. The framework must be:

- **Portable:** It should be able to plan paths for both rigid and articulated robots of any topology or arbitrarily high degrees of freedom, without any change to the underlying system.
- **Dynamic:** It should be able to plan collision free motion for an object in the presence of moving obstacles, and other collaborating robots, whose motion is not known a priori.
- **General:** The system must allow the user to easily specify a wide range of relationships and behaviors between the robots and other objects in the scene.
- **Interactive:** The system must run at interactive rates and allow the user to trade execution speed for accuracy, to enable rapid design and path verification.

These design goals suggest a fusion of global and local planning techniques to capitalize on the benefits of both. We propose a constraint-based planning approach that provides a local planning framework powerful enough to handle dynamic scenes, efficient enough to run at interactive rates, and general enough to incorporate many types of geometric constraints to govern an object's motion. These constraints can represent complex relationships between collaborating entities and also link collections of rigid objects to behave as articulated robots. This framework also allows natural extension to planning of flexible robots and incorporation of dynamics, as well as non-holonomic and other types of constraints.

3.2 Simulation Framework

The basic essence of our framework is to describe each rigid object in the planning scene as a dynamical system, which is characterized by its state variables (i.e. position, orientation, linear and angular velocity). In this framework, a robot can be a rigid body, or a collection of rigid bodies, subject to the influence of various forces in the workspace, and restricted by various motion constraints. This transforms a motion planning problem into a problem of defining suitable constraints, and then simulating the rigid body dynamics of the scene with each constraint acting as a virtual force on the objects. We will return to the problem of defining constraints to solve a planning problem in Sec. 4.

Next we'll explain the simulation framework with the following notation:

- Let $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_m\}$ be a set of m rigid objects.
- For each $\mathcal{R}_i$ at time t, let a state vector $s_i^t = (pos_i^t, rot_i^t, lin_i^t, ang_i^t)$ represent the objects position, rotation, linear and angular velocity.
- Let S^t be the system state vector, obtained by concatenating the state vectors s_i^t for all i.
- Let $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ be a set of n constraints.
- For each constraint $\mathcal{C}_j$, let $F_j(S^t)$ be the force induced by constraint j given the object system state S^t.

The simulation steps from time t to time $t + h$ and updates the state of each object subject to the forces induced by the constraints.

```
BEGIN LOOP
    Compute Constraint Forces: Summing up all contributing forces, $F(S^t) = \sum_{j=1}^{n} F_j(S^t)$.
    Update System State: Compute $S^{t+h}$ from $S^t$ subject to the force $F(S^t)$ [26].
    Update Object States: For each object $\mathcal{R}_i$, update $s_i^{t+h}$ from $S^{t+h}$.
    Increment Time: $t = t + h$
END LOOP
```

In this framework the solution to the motion planning problem, for a particular object R_i emerges as the sequence of states, $\{s_i^t, s_i^{t+h}, \ldots, s_i^{t+k*h}\}$, such that the object is in its initial configuration at time t, and achieves the goal configuration at time $t+k*h$. The simulation must run for as many time steps as necessary for all objects, for which a planned path is desired, to reach their goal configurations.

4. PLANNING SCENE FORMULATION

In this section we will describe how to render a planning scenario in the form of constraints for the constraint-based planning framework. Assume that the geometry representing the robots and obstacles is given, as well as prescribed motion, simulated or scripted, for the obstacles over time. Our system then defines constraints that will restrict the motion of the robots to meet the design specifications, and also guide the robots to complete the planning tasks.

4.1 Constraint Classification

To achieve the desired results without robustness problems, we classify the constraints into two categories: soft and hard constraints.

Hard Constraints are those that absolutely must be satisfied at every time step of the simulation. Examples of the high level hard constraints include object non-penetration, articulated robot joint connectivity, and articulated robot joint angle limits.

Soft Constraints serve as guides to *encourage* or influence the objects in the scene to behave in certain ways. Some common examples of soft constraints include having an object move towards a goal configuration, avoid the nearest obstacles, and move along some predefined path. Soft constraints are the more difficult type to handle because there can be many competing ones acting on an object. We provide several methods to resolve such conflicts. First, the soft constraint penalty force scales with the degree to which the soft constraint is violated. Second, each soft constraint is given a priority, from $[0, 1]$, which scales the constraint force.

4.2 Hard Constraints

To ensure that the simulation enforces the high level hard constraints we use three atomic hard constraints:

- Non-Penetration Constraints
- Point Distance Constraints
- Point Planar Angle Constraints

4.2.1 Non-Penetration Constraints

Assuming that all rigid objects in the scene represent closed volumes, we consider a non-penetration constraint between two objects to be satisfied as long as their volumes are disjoint. In most cases, this constraint can be weakened to only require that the objects' boundaries do not penetrate each other. In most planning scenarios, non-penetration constraints should be applied between all objects in the scene, ensuring that the objects behave as if they are solid, although it is possible to have objects that are selectively solid, or completely permeable, if desired.

Unlike the non-penetration constraints, the second and third categories of hard constraints are independent of the object geometry; they instead enforce relationships between points and vectors defined in the scene. These points and vectors can be fixed in world coordinates or expressed relative to the coordinate frames of objects in the scene.

4.2.2 Point Distance Constraints

These constraints enforce a fixed separation between pairs of points. Thus at a time t, given:

- $p_1 \in \mathcal{R}^3$, and its world transformation T_1^t
- $p_2 \in \mathcal{R}^3$, and its world transformation T_2^t
- $d \in \mathcal{R}$, the constraint distance.

the constraint is satisfied when $dist(T_1^t(p_1), T_2^t(p_2)) = d$, where $dist()$ is the Euclidean distance.

4.2.3 Point Planar Angle Constraints

These constraints enforce the angle between two points, about a specified axis of rotation. To define these constraints we require, at time t:

- $p_1 \in \mathcal{R}^3$, and its world transformation T_1^t
- $p_2 \in \mathcal{R}^3$, and its world transformation T_2^t
- $o \in \mathcal{R}^3$ the origin of the joint, and its world transformation T_o^t

- $\overrightarrow{axis} \in \mathcal{R}^{\mathbf{3}}$, the axis of rotation for the joint, and its world transformation T_a^t
- $\theta_{min}, \theta_{max} \in \mathcal{R}^3$, the angle limits.

We define the angle θ as the planar angle between the vectors $T_1^t(p_1) - T_o^t(o)$ and $T_2^t(p_2) - T_o^t(o)$ in the plane normal to $T_a^t(\overrightarrow{axis})$. The constraint is satisfied as long as θ is in the interval $[\theta_{min}, \theta_{max}]$, see Fig. 1.

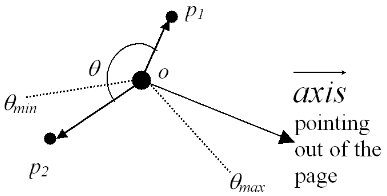


Figure 1: An angular constraint between two points.

4.2.4 *Combining Point Constraints*

In this section we will illustrate how point constraints can be combined to link rigid objects to form articulated objects. Our method uses a rigid structure, such as a tetrahedron, which we define using point distance constraints, to represent the coordinate frame of each rigid object. We chose four linearly independent points in the object's coordinate system, and set distance constraints distances between them to enforce their initial separations. When the object is transformed the rigid structure is also transformed and its distance constraints remain trivially satisfied. At the same time, as long as the constraints that define the rigid structure are satisfied we can uniquely determine the object's transformation from the world locations of the four points of the rigid structure. To constrain the relative motion of objects in the scene we define constraints between the points of their rigid structures. At each frame of the simulation the constraints between all of these points are enforced, possibly changing their locations. From these new point locations we update the world state of the associated rigid object to a state that respects the constraints.

Example: A Ball Joint Suppose that we have two rigid objects, $\mathcal{R}_1$ and $\mathcal{R}_2$, and wish to constrain them to only two relative rotational degrees of freedom, i.e. a ball joint, about some world point o between them. We would then define $p_1 = o$ in the coordinate frame of $\mathcal{R}_1$, and $p_2 = o$ in the coordinate frame of $\mathcal{R}_2$. Then we would link p_1 and p_2 to the rigid structures of $\mathcal{R}_1$ and $\mathcal{R}_2$ respectively, using three linearly independent distance constraints each. These constraints ensure that p_1 is rigidly attached to the structure representing $\mathcal{R}_1$ and p_2 is rigidly attached to the structure representing $\mathcal{R}_2$. We then define a distance constraint, with constraint distance of 0, between p_1 and p_2, so that their world positions are constrained to coincide. This ensures that when all constraints are satisfied the two objects, $\mathcal{R}_1$ and $\mathcal{R}_2$, are rigidly attached to a common rotation point at o, whose position is now expressed in the coordinate frames of the two objects, see Fig. 2.

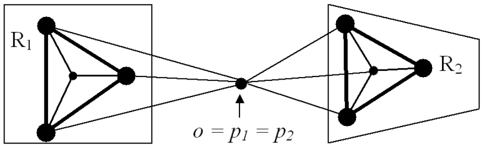


Figure 2: A ball joint, built from distance constraints linking the rigid structures of the two objects.

Example: A Revolute Joint We can extend the formulation for a ball joint to obtain a revolute joint between two rigid objects from the specification of the joint location, axis of rotation, and angle limits. To define a revolute joint between two rigid objects, $\mathcal{R}_1$ and $\mathcal{R}_2$, we use two ball joints. As long as the two ball joints have distinct centers of rotation that lie on the intended rotation axis, the constraints will limit the rigid objects to only one relative rotational degree of freedom about the axis. For additional stability we define a redundant distance constraint between the two rotation centers to maintain their separation along the rotation axis.

To limit the angle of the revolute joint, we use a point planar angle constraint, defined to limit the angle about the rotation axis between one point attached to the rigid structure of R_1, and one point attached to the rigid structure of $\mathcal{R}_2$.

There has been previous work, in the field of robot control, on specifying joints using constraints [25]. The advantage of our approach is that, using the rigid structure formulation, we reduce the problem of solving constraints between rigid objects to a much simpler problem of solving constraints between points. As we will show in Sec. *5.1.2*, this allows for a very fast and stable constraint solving method to be used.

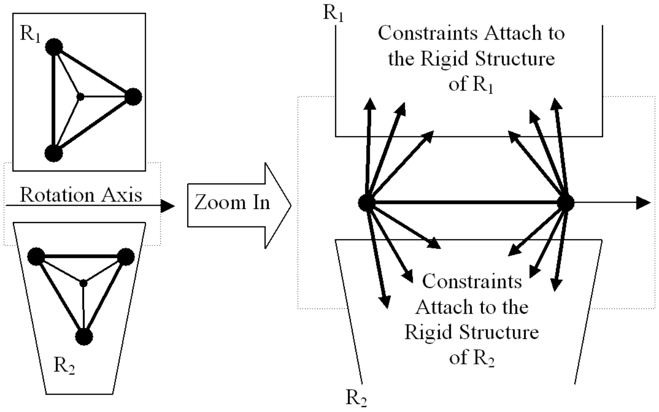

R1
Rotation Axis
Zoom In
R2
R1
Constraints Attach to the Rigid Structure of R1
Constraints Attach to the Rigid Structure of R2
R2

Figure 3: A revolute joint, built from distance constraints linking the rigid structures of the two objects.

4.3 Soft Constraints

Soft constraints in our framework are user specified constraints that generate penalty forces to *guide* the motion of objects without imposing strict motion restrictions. The types of soft constraints in our current system are:

- Goal Attraction
- Surface repulsion
- Path following

4.3.1 *Goal Attraction*

If the robot has not reached its goal configuration, this constraint generates a penalty force that attracts the robot towards its goal. This constraint could be applied to one component of an articulated robot, such as the end effector, or to multiple components of the robot.

4.3.2 *Surface Repulsion*

These constraints are used to have the robot be repelled from the surfaces of nearby obstacles. To define a surface repulsion constraint for a rigid body, $\mathcal{R}$, relative to a second rigid object O, we specify a distance threshold, δ, and a force coefficient $k \in [0, 1]$. The distance threshold is the minimum distance allowed between the objects before the $\mathcal{R}$ acts to move away from O. If $\mathcal{R}$ is following an estimated path, often the path will have an associated minimum distance tolerance for static obstacles, which can be used to initialize δ. The priority specifies the relative importance of repulsion constraints, so that if $\mathcal{R}$ is trapped between two obstacles, it will give more priority to evading one than the other. Soft constraint priorities are all by default equal to 1, unless the user specifies another value.

4.3.3 Estimated Paths

One well-known problem with using the penalty forces to achieve planning goals is that the robot can be caught in a local minima and fail to reach the goal. To address this issue we integrate global geometric analysis, generated ahead of time by a high-level task planner [21], into our planning framework. There are many well known techniques for obtaining an estimated path for a robot based on the static obstacles in the scene, such as a medial axis based planner [10, 14], a Probabilistic Roadmap Planner [15], a binary space partitioning of the workspace [23], or simply by taking input from a user [5]. A path from any of these can be integrated into the simulation using a path following constraint.

To specify a path constraint for some rigid body, $\mathcal{R}$, we require an ordered list of points $p_1, p_2, p_3, ..., p_n$, which represent the milestones along the path, a distance threshold δ, and a force coefficient $k \in [0, 1]$. In our system, it is assumed that the last point on the path is the goal location that we wish the object $\mathcal{R}$ to reach, and all other points are only landmarks guiding the object as to how this goal may be achieved. This assumption could be relaxed if we add weights to every path location indicating how important it is for $\mathcal{R}$ to reach that location. The distance threshold, δ, is used to determine if $\mathcal{R}$ is close enough to a milestone to consider that milestone reached. The default value for δ is the radius of the smallest sphere enclosing $\mathcal{R}$. As in the case of surface repulsion constraints, Sec. *4.3.2*, the default value of k is 1, which can be changed if some particular priority ranking between forces is desired. In our current system, the object's orientation along the path is left arbitrary, so that it can be determined by other soft constraints, such as surface repulsion. For a scene with an articulated robot, we can assign a part of the robot, such as the end effector, to follow the high-level path sequence.

5. CONSTRAINT IMPLEMENTATION

5.1 Solving The Constraints

We use two main constraint solving techniques in our current framework: penalty-based methods, used to represent soft constraints, and iterative relaxation, used to enforce hard constraints. The algorithm for updating the states of objects in our simulation is quite simple. We first apply the penalty forces due to the active soft constraints and update the simulation state. Next we apply an iterative relaxation technique which modifies the object states to ensure that all hard constraints are satisfied at the end of the time step.

5.1.1 Applying Penalty Forces

We require that each type of soft constraint, $\mathcal{C}_i$, has a method $Get\_Penalty\_Force(\mathcal{C}_i,\ \mathcal{R}_j,\ S^t,\ t)$ which returns the force and torque generated by the constraint $\mathcal{C}_i$, on object $\mathcal{R}_j$, at time t, with the scene in the state S^t. The total force on an object is the sum of all penalty forces acting on that object. To update the object state, for each object, we integrate this total force using the Midpoint Method [26]. We use this method because some of the penalty forces are computationally intensive to evaluate, and the midpoint method provides stable integration with only two force evaluations per time step.

5.1.2 Iterative Relaxation

Once objects in the scene are updated as a result of the penalty forces due to the soft constraints, their state may violate one or more of the hard constraints. To efficiently ensure that these constraints are satisfied, we use the well known Nonlinear Gause-Siedel iterative relaxation method [24]. For each hard constraint, $\mathcal{C}_h$, we define the residual, $Res(\mathcal{C}_h, S^t)$, to be a real number which represents the degree to which $\mathcal{C}_h$ is violated when the system is in state S^t. For each type of hard constraint $\mathcal{C}_h$, we require an instance solver $Relax(\mathcal{C}_h, S^t)$, which returns a new state S in which $Res(C_h, S) = 0$. The specific methods used to relax each type of hard constraint, will be explained in Sec. 5.2.

The iterative relaxation method, described in Algorithm. 5.1, relaxes each constraint in sequence repeatedly, until the objects converge to a state for which the sum of the residuals, over all constraints, is zero. Particular care must be taken to ensure that point distance constraints described in Sec. *4.2.2* and point planar angle constraints described in Sec. *4.2.3* are satisfied first, because only when these constraints are satisfied can we read back the resulting transformations on the rigid objects to be used in the remainder of the constraint solving iteration. We call the procedure which updates the states of the rigid objects in the system from the locations of the points that make up their associated rigid structures $Update\_Object\_States\_From\_Points(S)$ in Algorithm. 5.1.

5.1.3 Convergence of the Relaxation Method

The reason that we use the iterative relaxation method, Sec. *5.1.2*, instead of other techniques (e.g. Lagrangian formalism) for satisfying all the hard constraints, is because of its simplicity and because it allows our implementation to achieve interactive performance in most practical scenarios, even with a large number of active constraints. This rapid convergence can be attributed to two factors: temporal coherence and compatible constraints.

Our system is able to take advantage of temporal coherence because, as a physical simulation, it uses small time steps during which the objects in the scene move very little. Moreover, if we assume that all hard constraints are satisfied at the beginning of a time step, the fact that the object motion due to soft constraints is small ensures that when the iterative method begins the objects are not far from a configuration that satisfies the hard constraints. This all but ensures that the iterative method converges to the solution. It also allows convergence to take place in a relatively small number of iterations, providing stable interactive performance in many practical scenarios. Of course, it is possible for the method to never converge when, for example, there are two incompatible constraints, such that satisfying one necessarily violates the other. This situation does not occur in practice because the hard constraints are typically defined so that they are compatible with each other. Furthermore, in practical situations the hard constraints are satisfied before the planning simulation starts, allowing the system to benefit from temporal coherence from the beginning.

One of the goals of our planning framework was to allow the user to trade performance for accuracy to enable both very rapid prototyping and exact path computation. We can achieve this goal by allowing an upper bound to be imposed on the number of iterations used in relaxation, to guarantee that the simulation runs at the rate desired. If the system fails to converge within the specified number of iterations, the simulation continues as if it had converged. The result is error that manifests as small violations in the hard constraints. Despite of this problem, we have found that in practice the computed solution is a good approximation of the error free path obtained when no limit is placed on the number of constraint solving iterations. This can be considered as a useful feature for rapid prototyping of CAD designs, when a quick estimation of the planned motion is required to provide real-time user feedback in the design process.

```
Relax_Constraints

Input  The state $S^t$, at time $t$, of all rigid objects, points and vectors
       in the simulation, the set $\mathcal{C}_p$ of point hard constraints, and the
       set $\mathcal{C}_r$ of rigid object hard constraints.
Output  New state vector $S$ which satisfies all hard constraints.

Let $S \leftarrow S^t$.
While $\sum_{i=0}^{|C_p|} |Res(C_i, S)| + \sum_{j=0}^{|C_r|} |Res(C_j, S)| > 0$:
{
    While $\sum_{i=0}^{|C_p|} |Res(C_i, S)| > 0$:
    {
            for each point hard constraint $\mathcal{C}_p$:
                $S \leftarrow Relax(C_p, S)$.
    }

    $S \leftarrow Update\_Object\_States\_From\_Points(S)$

    for each hard rigid object constraint $\mathcal{C}_r$:
        $S \leftarrow Relax(C_r, S)$.
}
return The state $S$.
```

ALGORITHM 5.1: Relax Hard Constraints

5.2 Solving Hard Constraints

To satisfy each of the hard constraint types of Sec. 4.2: point distance, point planar angle and non-penetration, we have a corresponding $Relax$ solver that is used in the iterative algorithm of Sec. 5.1.2. For the constraints that act on rigid objects the solver updates the state, position and orientation, of the objects, while for point constraints the solver modifies the positions of the points.

5.2.1 *Point Distance Constraints*

Given the formulation of a point distance constraint (Sec. 4.2.2) between two points, p_1 and p_2, with corresponding world transformation, T_1^t and T_2^t, and a distance threshold d, we define the residual of the distance constraint:

$$Res(C_{dist}, S) = \Delta_d = dist(T_1^t(p_1), T_2^t(p_2)) - d,$$

where Δ_d represents the linear distance that the two points must travel to reach the required separation. To solve the distance constraint we simply move each point a straight line distance of $\Delta_d/2$ towards each other.

5.2.2 *Point Planar Angle Constraints*

Given the formulation of a point planar angle constraint between points p_1 and p_2, with angle limits θ_{min} and θ_{max}, we compute the angle θ as defined Sec. 4.2.3. The residual of the angle constraint is then defined as:

$$Res(C_{ang}, S) = \begin{cases} \theta - \theta_{max} & \text{if } \theta > \theta_{max} \\ \theta - \theta_{min} & \text{if } \theta < \theta_{min} \\ 0 & \text{otherwise} \end{cases}$$

To satisfy the angel constrain, if $\Delta_\theta = Res(C_{ang}, t) \neq 0$, the point $T_1^t(p_1)$ is rotated an angle $\Delta_\theta/2$, and $T_2^t(p_2)$ rotates an angle $-\Delta_\theta/2$ about the rotation axis, $T_a^t(\overrightarrow{axis})$.

5.2.3 *Non-Penetration Constraints*

In our current implementation, these are the only constraints for which the $Relax$ method directly modifies the transformations of rigid objects in the scene. We use an in-house proximity (collision) query package, PQP [18, 12], to detect when two objects penetrate. The residual for a non-penetration constraint is then just 0 if the object are disjoint and 1 if the objects are not. The problem of separating objects that penetrate, in a physical simulation, is one that has been addressed in many ways [26]. The approach that we use to implement the $Relax$ solver for non-penetration constraints is the impulse-based rigid body dynamic simulation [26]. The advantages of this method is that objects are guaranteed to be disjoint at the end of every time step, and that objects rebound from collisions in the most natural possible way.

5.3 Solving Soft Constraints

For each soft constraint we require a method, $Get\_Penalty\_Force$, which produces a force along the gradient vector of the constraint.

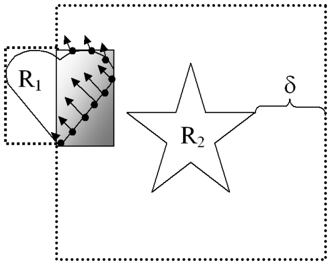


Figure 4: A portion of the distance field of R_2 which generates forces that act on R_1 pushing it away from R_2.

5.3.1 *Surface Repulsion*

To apply a surface repulsion constraint between object $\mathcal{R}_1$ and a second object $\mathcal{R}_2$, as described in Sec. 4.3.2, we first perform some computations using axis-aligned bounding boxes as approximations for the objects involved. For object $\mathcal{R}_2$ we take the axis-aligned bounding box and expand it by the distance threshold, δ (Fig. 4). We intersect this expanded bounding box with the bounding box of $\mathcal{R}_1$ to perform a quick rejection test to determine if the two objects are further than the distance threshold δ apart. If this is the case, then we can terminate the computation with no penalty force applied. If the test fails then we compute the intersection, call it I, of the two bounding boxes. We then use a hardware accelerated distance field computation [14], to generate the distance field for the surface features of object $\mathcal{R}_2$ in the region I. The fact that this computation is performed using graphics hardware enables the distance field of the object to be generated in real time without any precomputation or assumptions about the geometry. As a preprocess, a sampling of the surface of object $\mathcal{R}_1$, at some user specified resolution, is computed. The default resolution is the pixel resolution. For each sample point on $\mathcal{R}_1$ that lies in I, we check the distance from that point to the nearest point on the surface of $\mathcal{R}_2$ by referencing the distance field. For each sample point we generate a force in the direction of the gradient of the distance field, proportional to the distance between that sample point and $\mathcal{R}_2$, as seen in Fig. 4. This force should be zero for sample points beyond the distance threshold, and increase to infinity as the distance between the surfaces decreases. In our system the force for each sample point, p_i, is:

$$force(p_i, \mathcal{R}_2) = \begin{cases} \frac{1}{dist(p_i, \mathcal{R}_2)^4} - \frac{1}{\delta^4} & \text{if } dist(p_i, \mathcal{R}_2) < \delta \\ 0 & \text{otherwise} \end{cases}$$

And, the force induced by this constraint on $\mathcal{R}_1$ is the sum of all forces on all l sample points on $\mathcal{R}_1$. This is a force that moves $\mathcal{R}_1$ away from $\mathcal{R}_2$, enforcing the surface repulsion constraint.

5.3.2 Goal Attraction

As described in Sec. 4.3.1, if the robot has not reached its goal configuration, this constraint generates a penalty force that attracts the robot towards its goal. The strength of this force is directly proportional to the distance between the robot and the goal, and its direction is simply the direction between the center of mass of the robot and the location of the goal.

5.3.3 Path Following

Currently we implement a path following constraint, Sec. 4.3.3, by first finding the closest point on the path, p_i, to the center of mass of the robot $\mathcal{R}$. If this point is not within the path distance threshold, δ, from the center of mass, then we consider that $\mathcal{R}$ has not reached p_i and we apply a force at the center of mass of $\mathcal{R}$ to push it towards p_i . If $\mathcal{R}$ is within the distance threshold of p_i, we apply a force at the center of mass of R in the direction of $p_{i+1} - p_i$ to push $\mathcal{R}$ along the path.

6. APPLICATIONS TO PROTOTYING

6.1 Implementation

Our system was implemented in an object-oriented framework using C++. We use the Proximity Query Package [12, 18] for collision detection, to enforce non-penetration constraints, and our in-house library HAVOC3D [14] to generate distance fields for surface repulsion constraints.

6.2 System Demonstration

We have tested our motion planning system in the following virtual prototyping applications:

Scene 1: Maintainability Study

In assembly maintainability studies, motion planning is used to find whether it is possible to remove a particular part from an assembly, and if so, to find one possible removal path [5]. In our example, shown in Fig. 5(a), a bolt and a washer must avoid each other in the confines of tight compartment inside a pump assembly. The goal, to remove the bolt from the assembly, requires both objects to maneuver around each other without colliding.

Scene 2: Automated Car Painting

In this example seen in Fig. 5(b), an articulated robot arm, with 6 degrees of freedom, is used to trace a path along the body of a car for painting. The robot is composed of rigid components that are held together by constraints. For all of the components of the robot, the planner must compute paths that satisfy the joint constraints, do not collide with the obstacles or the car, and lead the end effector along the prescribed path.

Scene 3: Assembly Line Planning

In this example, shown in Fig. 5(c), the robot arm from scene 2 must access a part moving past it on a conveyer belt. The factory floor contains a piping structure that is moving over the conveyer belt in the opposite direction to the part's movement. The moving obstruction causes the robot to reactively modify its path to avoid collision.

The timings for these scenarios are presented in Table 1. The timings were taken on a PC with a 933MHz Pentium III processor, 256MB RAM and an nVidia GeForce3 graphics card. The motion sequences captured in MPEG are available at:

http://gamma.cs.unc.edu/cplan.

Scene	Poly	Cons	**Per Step**	**Total**
(1) Maintainability	20470	4	0.093 sec	67 sec
(2) Auto Painting	25738	43	0.038 sec	18 sec
(3) Assembly Line	16962	43	0.0085 sec	16 sec

Table 1: Benchmark timings in seconds on three example scenes. *Poly:* The number of polygons in each scene. *Cons:* The total number of active constraints in each scene. *Per Step:* The average time for the planner to compute one time step of the simulation. *Total:* The total time taken to complete the planning task.

6.3 Discussion

The planning tasks in the example scenes execute, on average, between 10 and 120 time steps per second. The primary bottleneck in our current implementation is the distance field computation used to determine the penalty forces for the surface repulsion constraints. We use a one-level bounding box culling to limit the application of this computation to areas near potential surface collisions. We also use simplified geometry for computing the distance field wherever appropriate to speed up the proximity queries. This approach works well, unless the scene, as in the case of Scene 1, has highly non-convex complex geometry that is poorly approximated by the bounding boxes. In such cases, hierarchical bounding box culling could be used to further limit the application of the distance field computation to increase runtime performance. We are currently working on this optimization, as well as accelerating the 3D distance field computation.

The constraint solver we have developed for the current system uses an iterative relaxation method that is specialized to provide interactive performance when planning the motion of rigid and articulated robots in dynamic scenes. It works well for overconstrained and consistent systems, such as those produced by our method of modeling robot joints using constraints, and in our planning framework where the dynamic simulation typically advances in small time steps allowing it to take advantage of temporal coherence to achieve performance and stability. However, it is possible for our framework to incorporate other, very efficient, constraint solvers based on the *extension* of [2, 3, 17], as we extend this work to plan the motion of flexible bodies and also include different types of constraints in our system.

7. CONCLUSION AND FUTURE WORK

We have presented a novel framework for motion planning in virtual prototyping applications. We reformulate the motion planning problem into a physical simulation where constraints on the robot's motion guide it from its starting configuration to its goal. These constraints can enforce non-penetration constraints among objects, the angle limits and connectivity of articulated robot joints, the avoidance of collision, the following of estimated paths, and many other possible relationships between the robots and objects in the scene. The flexibility of our framework offers the possibility of natural extension in the following areas:

- **Inclusion of Additional Constraints:** such as non-holonomic constraints on object motion, as well as constraints that enforce more complex interactions (e.g. maintaining line of sight) between collaborating robots.
- **Extension to Flexible Geometry:** since our planning framework assumes no fixed or rigid geometry throughout the planning simulation.
- **Incorporating Direct Human Interaction:** to allow the user to directly control the motion of parts of the robot or obsta-

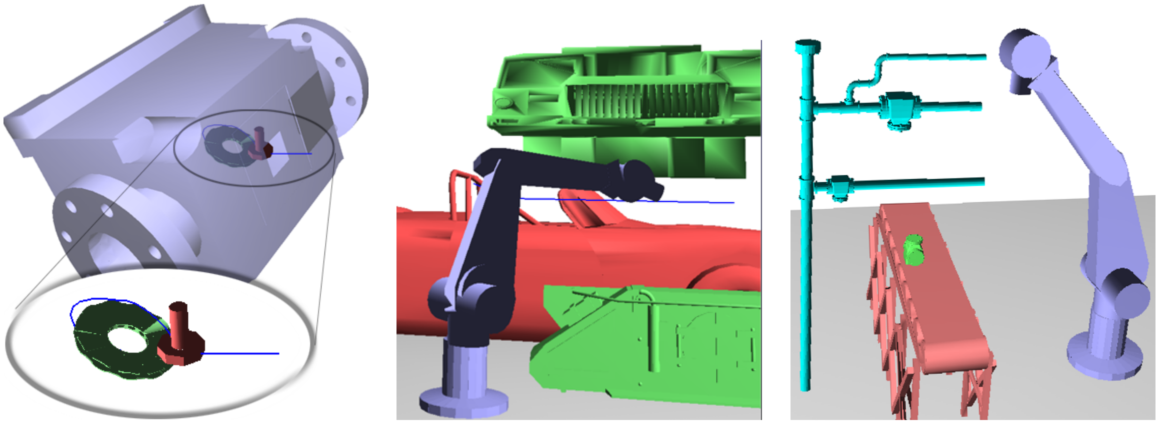


Figure 5: From left to right, *Maintainability Study Scene:* the planner must extract the bolt from the pump assembly. Both the bolt and the washer must be moved simultaneously around each other to avoid collision; *Automated Car Painting Scene:* the robot arm follows a path over the car body while avoiding obstacles; *Assembly Line Planning Scene:* the robot arm avoids the moving pipes to reach a moving part passing on the conveyer belt.

cles in the scene, thus better enabling interactive prototyping of CAD designs and faster verification of the design.

Acknowledgements

This research is supported in part by NSF DMI-9900157, NSF IIS-9821067, ONR N00014-01-1-0067 and Intel.

8. REFERENCES

[1] J. Barraquand and J.-C. Latombe. Robot motion planning: A distributed representation approach. *Int. J. Robotic Research*, 1991.

[2] W. Bouma, X. Chen, I. Fudos, C. Hoffmann, and P. Vermeer. *An Electronic Primer on Geometric Constraint Solving*. http://www.cs.purdue.edu/homes/cmh/electrobook/intro.html, 1990.

[3] B. Bruderlin and D. Roller (eds). *Geometric Constraint Solving and Applications*. Spring Verlag, 1998.

[4] J.F. Canny. *The Complexity of Robot Motion Planning*. ACM Doctoral Dissertation Award. MIT Press, 1988.

[5] H. Chang and T. Li. Assembly maintainability study with motion planning. In *Proceedings of International Conference on Robotics and Automation*, 1995.

[6] H. Choset and J. Burdick. Sensor based planning, part ii: Incremental construction of the generalized voronoi graph. *IEEE Conference on Robotics and Automation*, 1995.

[7] H. Choset and J. Burdick. Sensor based planning: The hierarchical generalized voronoi graph. *Workshop on Algorithmic Foundations of Robotics*, 1996.

[8] L. S. Homem de Mello and A. C. Sanderson. A correct and complete algorithm for the generation of mechanical assembly sequences. *IEEE Trans. on Robotics and Automation*, 7(2):228–240, 1991.

[9] M. Erdmann and M. Mason. An exploration of sensorless manipulation. *IEEE Tr. on Robotics and Automation*, 4:369–379, 1988.

[10] M. Foskey, M. Garber, M. Lin, and D. Manocha. A voronoi-based hybrid planner. *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2001.

[11] K. Goldberg. Orienting polygonal parts without sensors. *Algorithmica*, 10:201–225, 1993.

[12] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. In *Proc. of ACM Siggraph'96*, pages 171–180, 1996.

[13] D. Halperin, J. Latombe, and R. Wilson. A general framework for assembly planning: The motion space approach. *Algorithmica*, 1999.

[14] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. *Proceedings of ACM SIGGRAPH 1999*, pages 277–286, 1999.

[15] L. Kavraki and J. C. Latombe. Randomized preprocessing of configuration space for fast path planning. *IEEE Conference on Robotics and Automation*, pages 2138–2145, 1994.

[16] O. Khatib. Real-time obstable avoidance for manipulators and mobile robots. *IJRR*, 5(1):90–98, 1986.

[17] G. Kramer. *Solving Geometric Constraint Systems: A case study in kinematics*. MIT Press, 1992.

[18] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha. Distance queries with rectangular swept sphere volumes. *Proc. of IEEE Int. Conference on Robotics and Automation*, 2000.

[19] J.C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[20] T. Lozano-Pérez and M. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Comm. ACM*, 22(10):560–570, 1979.

[21] T. Lozano-Perez and R. Wilson. Assembly sequencing for arbitrary motions. *Proc. IEEE International Conference on Robotics and Automation*, 1993.

[22] V. Lumelsky and A. Stepanov. Path planning strategies for point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.

[23] K. Ahrentsen N. Jacobsen, R. Larsen, and L. Overgaard. Automatic robotwelding in complex shipstructures. *J. Applied Artificial Intelligence*, 1997.

[24] J.M. Ortega and W.C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, 1970.

[25] L Overgaard, H. Petersen, and J. Perram. A general algorithm for dynamic control of multilink robots. *Int. J. Robotics Research*, 14(3), 1995.

[26] A. Witkin and D. Baraff. *Physically Based Modeling: Principles and Practice*. ACM Press, 1997. Course Notes of ACM SIGGRAPH.

Appeared as Technical Report TR03-014, Department of Computer Science, UNC Chapel Hill

CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware

Naga K. Govindaraju Stephane Redon Ming C. Lin Dinesh Manocha
University of North Carolina at Chapel Hill
{naga,redon,lin,dm}@cs.unc.edu

Abstract:

We present a novel approach for collision detection between multiple deformable and breakable objects in a large environment. Our algorithm takes into account low bandwidth to and from the graphics cards and computes a potentially colliding set (PCS) using visibility queries. It involves no precomputation and proceeds in multiple stages: PCS computation at an object level and PCS computation at sub-object level, followed by exact collision detection. We use a linear time two-pass rendering algorithm to compute each PCS efficiently. The overall approach makes no assumption about the input primitives or the object's motion and is directly applicable to all triangulated models. It has been implemented on a PC with NVIDIA GeForce FX Ultra graphics card and applied to different environments composed of a high number of moving objects with tens of thousands of triangles. It is able to compute all the overlapping primitives up to image-space resolution in a few milliseconds.

1 Introduction

High-performance 3D graphics systems are becoming as ubiquitous as floating-point hardware. They are now a part of almost every personal computer or game console. In addition, graphics hardware is becoming more programmable and is increasingly used as a co-processor for many diverse applications. These include ray tracing, intersection computations, simulation of dynamic phenomena, atmospheric effects, and scientific computations.

In this paper, we mainly address the problem of collision detection among moving objects, either rigid or deformable, using graphics hardware. Collision detection is an important problem in computer graphics, game development, virtual environments, robotics and engineering simulations. It is often one of the major computational bottlenecks in dynamic simulation of complex systems. Collision detection has been well studied over the last few decades. However, most of the efficient algorithms are limited to rigid bodies and require preprocessing. Although some algorithms have been proposed for deformable models, either they are limited to simple objects or closed sets, or they are designed for specialized models (e.g. cloth).

Many algorithms exploiting graphics hardware capabilities have been proposed for collision queries and proximity computations [Baciu and Wong 2002; Baciu et al. 1998; Hoff et al. 2001; Kim et al. 2002b; Myszkowski et al. 1995; Rossignac et al. 1992; Shinya and Forgue 1991; Vassilev et al. 2001]. At a broad level, these algorithms can be classified into two categories: use of depth and stencil buffer techniques for computing interference and fast computation of distance fields for proximity queries. These algorithms perform image-space computations, and are applicable to rigid and deformable models. However, they have three main limitations:

- **Bandwidth Issues:** Although graphics hardware is progressing at a rate faster than Moore's Law, the bandwidth to and from the graphics cards is not increasing as fast as computational power. Furthermore, many algorithms readback the frame-buffer or depth-buffer during each frame. These readbacks can be expensive on commodity graphics hardware, e.g. it takes 50 milli-seconds to read back the $1K \times 1K$ depth-buffer on a Dell 530 Workstation with NVIDIA GeForce 4 card.
- **Closed Objects:** Many of these algorithms are mainly restricted to closed objects, as they use graphics hardware stencil operations to perform virtual ray casting operations and determine whether a point is inside or outside.
- **Multiple Object-Pair Culling:** Most of the current algorithms are designed for a pair of objects and not intended for large environments composed of multiple moving objects.

Main Contribution: We present a novel algorithm for collision or interference detection among multiple moving objects in a large environment. Given an environment composed of triangulated objects, our algorithm computes a *potentially colliding set (PCS)*. The PCS consists of objects that are either overlapping or are in close proximity. We use visibility computations to prune the number of objects in the PCS. This is based on a linear time two-pass rendering algorithm that traverses the list of objects in forward and reverse order. The visibility relationships are computed using image-space occlusion queries, which are available on current graphics hardware.

The pruning algorithm proceeds in multiple stages. Initially it compute a PCS of objects. Next it considers all *sub-objects* (i.e. bounding boxes, groups of triangles, or individual triangles) in these objects and computes a PCS of sub-objects. Finally, it uses an exact collision detection algorithm to compute the overlapping triangles. The complexity of the algorithm is a linear function of the input and output size, as well as the size of PCS after each stage. Its accuracy is governed by the image-precision and depth-buffer resolution. Since there are no depth-buffer readbacks, it is possible to perform the image-space occlusion queries at a higher resolution without significant degradation in performance. The additional overhead is in terms of fill-rate and not the bandwidth.

We have implemented the algorithm on a Dell 530 Workstation with NVIDIA GeForce FX 5800 Ultra graphics card and a Pentium IV processor, and have applied to three complex environments: 100 moving deformable objects in a cube, 6 deforming tori (each composed of $20,000$ polygons), and two complex breakable objects composed of $35,000$ and

$250,000$ triangles. In each case, the algorithm can compute all the overlapping triangles in just a few milliseconds.

Advantages: As compared to earlier approaches, our algorithm offers the following benefits. It is relatively simple and makes no assumption about the input model. It can even handle "polygon soups". It involves no precomputation or additional data structures (e.g. hierarchies). As a result, its memory overhead is low. It can easily handle deformable models and breakable objects with changing geometry and topology. Our algorithm doesn't make any assumptions on object motion and temporal coherence between successive frames. It can efficiently compute all the contacts among multiple objects or a pair of highly tessellated models at interactive rates.

Organization: The rest of the paper is organized as follows. We give a brief survey of prior work on collision detection and hardware accelerated computations in Section 2. We give an overview of PCS computation using visibility queries in Section 3. We present our two-stage algorithm in Section 4. In Section 5, we describe its implementation and highlight its performance on different environments. We also analyze its accuracy and performance.

2 Previous Work

In this section, we give a brief survey of prior work on collision detection and graphics-hardware-accelerated approaches.

2.1 Collision Detection

Typically for a simulated environment consisting of multiple moving objects, collision queries consist of two phases: the "broad phase" where collision culling is performed to reduce the number of pairwise tests, and the "narrow phase" where the pairs of object in proximity are checked for collision [Cohen et al. 1995; Hubbard 1993].

Algorithms for narrow phase can be further subdivided into efficient algorithms for convex objects and general-purpose algorithms based on spatial partitioning and bounding volume hierarchies for polygonal or spline models (please see survey in [Klosowski et al. 1998; Lin and Gottschalk 1998; Redon et al. 2002]). However, these algorithms often involve precomputation and are mainly designed for rigid models.

2.2 Acceleration Using Graphics Hardware

Graphics hardware has been increasingly utilized to accelerate a number of geometric computations, including visualization of constructive solid geometry models [Goldfeather et al. 1989; Rossignac and Wu 1990], interferences and cross-sections [Baciu and Wong 2002; Baciu et al. 1998; Myszkowski et al. 1995; Rossignac et al. 1992; Shinya and Forgue 1991], distance fields and proximity queries [Hoff et al. 1999; Hoff et al. 2001], Minkowski sums [Kaul and Rossignac 1992; Kim et al. 2002a], and specialized algorithms including collision detection for cloth animation [Vassilev et al. 2001] and virtual surgery [Lombardo et al. 1999]. All of these algorithms perform image-space computations and involve no preprocessing. As a result, they are directly applicable to rigid as well as deformable models. However, the interference detection algorithms based on depth and stencil buffers [Baciu et al. 1998; Myszkowski et al. 1995; Rossignac et al. 1992] are limited to closed objects. The approaches based on distance field computations [Hoff et al. 1999; Hoff et al. 2001] can also perform distance and penetration computation between two objects. But, they require depth-buffer readbacks, which can be expensive on commodity graphics hardware.

3 Collision Detection Using Visibility Queries

In this section, we give an overview of our collision detection algorithm. We show how PCS can be computed using image-space visibility queries, followed by exact collision detection between the primitives.

Given an environment composed of n objects, $O_1, O_2, \ldots, O_n$. We assume that each object is represented as a collection of triangles. Our goal is to check which objects overlap and also compute the overlapping triangles in each intersecting pair. Our algorithm makes no assumption about the motion of objects or any coherence between successive frames. In fact, the number of objects as well as the number of triangles in each object can change between successive frames.

3.1 Potentially Colliding Set (PCS)

We compute a PCS of objects that are either overlapping or are in close proximity. If an object, O_i, does not belong to the PCS, it implies that O_i does not collide with any object in the PCS. Based on this property, we can prune the number of object pairs that need to be checked for exact collision. This is similar to the concept of computing the potentially visible set (PVS) of primitives from a viewpoint for occlusion culling [Cohen-Or et al. 2001].

We perform visibility computations between the objects in image space to check whether they are potentially colliding or not. Given a set S of objects, we test the relative visibility of an object O with respect to S using an image-space visibility query. The query checks whether any part of O is occluded by S. It rasterizes all the objects belonging to S. O is considered *fully-visible* if the fragments generated by rasterization of O have a depth value less than the corresponding pixels in frame buffer. We do not consider self-occlusion of an object (O) in determining its visibility status. We use the following lemma to check whether O is overlapping with any object in S.

Lemma 1: *An object O does not collide with a set of objects S if O is fully-visible with respect to S.*

Proof: The proof of this lemma is quite obvious. If O is overlapping with any object in S, then some part of O is occluded by S. We also note that this property is independent of the projection plane.

The accuracy of the algorithm is governed by the underlying precision of the visibility query. Moreover, this lemma only provides a sufficient condition for not colliding and not a necessary condition.

3.2 Visibility Based Pruning

We use Lemma 1 for PCS computation. Given n objects $O_1, ..., O_n$, we check if O_i potentially intersects with at least one of $O_1, .., O_{i-1}, O_{i+1}, ..., O_n$, $1 \leq i \leq n$. Instead of checking all possible pairs (which can be $O(n^2)$), we use the following lemma to design a linear-time algorithm to perform a conservative check.

Lemma 2: *Given n objects $O_1, O_2, ..., O_n$, an object O_i does not belong to PCS if it does not intersect with $O_1, .., O_{i-1}, O_{i+1}, ..., O_n$, $1 \leq i \leq n$. This test can be easily decomposed as: an object O_i does not belong to PCS if it does not intersect with $O_1, .., O_{i-1}$ and with $O_{i+1}, ..., O_n$, $1 \leq i \leq n$.*

Proof: Follows trivially from the definition of PCS.

We use Lemma 2 to determine if an object belongs to PCS. Our algorithm uses a two pass rendering approach to compute the PCS. In the first pass, we check if O_i potentially intersects with at least one of the objects $O_1, .., O_{i-1}$. In the second pass, we check if it potentially intersects with one of $O_{i+1}, ..., O_n$. If an object does not intersect in either of the

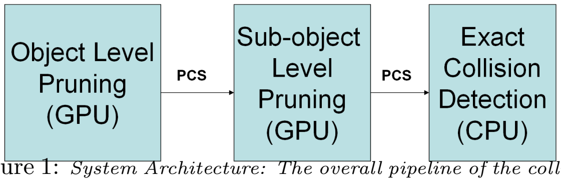

Object Level Pruning (GPU)
PCS
Sub-object Level Pruning (GPU)
PCS
Exact Collision Detection (CPU)

Figure 1: *System Architecture: The overall pipeline of the collision detection algorithm for large environments*

two passes, then it does not belong to the PCS.

Each pass requires the object representation for an object to be rendered twice. We can either render all the triangles used to represent an object or a bounding box of the object. Initially, the PCS consists of all the objects in the scene. We perform these two passes to prune objects from the PCS. Furthermore, we repeat the process by using each coordinate axis as the axis of projection to further prune the PCS. We use Lemma 1 to determine if an object potentially intersects with a set of objects or not.

It should be noted that our GPU based pruning algorithm is quite different from algorithms which prune PCS using 2D overlap tests and is *not* an extension to avoid frame-buffer readbacks using occlusion queries. Our algorithm does not involve frame-buffer readbacks and computes a PCS which is less conservative than 2D overlap algorithms.

3.3 Localizing the Overlapping Features

Many applications need to compute the exact overlapping features (e.g. triangles) for collision response. We initially compute the PCS of objects based on the algorithm highlighted above. Instead of testing each object pair in the PCS for exact overlap, we again use the visibility formulation to identify the potentially intersecting regions among the objects in the PCS. Specifically we use a fast global pruning algorithm to localize these regions of interest.

We decompose each object into sub-objects. A sub-object can be a bounding box, a group of triangles (say a constant k) or a single triangle. We extend the approach discussed in section 3.1 to sub-object level and compute the potentially intersecting regions based on the following lemma.

Lemma 3: *Given n objects $O_1, O_2, ..., O_n$ and each object O_i is composed of m_i sub-objects $T^i_1, T^i_2, ..., T^i_{m_i}$, a sub-object T^i_k of O_i does not belong to the object's potentially intersecting region if it does not intersect with the sub-objects of $O_1, .., O_{i-1}, O_{i+1}, ..., O_n$, $1 \leq i \leq n$. Moreover, a sub-object T^i_k of O_i does not belong to the potentially intersecting region of the object if it does not intersect with the sub-objects of $O_1, .., O_{i-1}$ and $O_{i+1}, ..., O_n$, $1 \leq i \leq n$.*

Proof: Follows trivially from Lemma 2.

In this case we again use visibility queries to resolve the intersections among sub-objects of different objects. However, we do not check an object for self-intersections or self-occlusion and therefore, do not perform visibility queries among the sub-objects of the same parent object.

3.4 Collision Detection

Our overall algorithm performs pruning at two stages, object level and sub-object level, and eventually checks the primitives for exact collision.

- **Object Pruning**: We perform object level pruning by computing the PCS of objects. We first use AABBs of the objects to prune this set. Next we use the exact triangulated representation of the objects to further prune the PCS. If the PCS is large, we use the sweep-and-prune algorithm [Cohen et al. 1995] to compute potentially colliding pairs and decompose the PCS into smaller subsets. The sweep-and-prune algorithm projects the bounding boxes on each axis, sorts them and check them for possible overlap.
- **Sub-Object Pruning**: We perform sub-object pruning to identify potential regions of each object in PCS that may be involved in collision detection.
- **Exact Collision Detection**: We perform exact triangle-triangle intersection tests on the CPU to check if two objects collide or not.

The architecture of the overall system is shown in Fig. 1, where the first two stages are performed using image-space visibility queries (on the GPU) and the last stage is performed on the CPU.

4 Interactive Collision Detection

In this section, we present our overall collision detection algorithm for computing all the contacts between multiple moving objects in a large environment. It uses the visibility pruning algorithm described in Section 3.2. The overall algorithm is general and applicable to all environments. We also highlight many optimizations and the visibility queries used to accelerate the performance of our algorithm.

4.1 Pruning Algorithm

We use a two-pass rendering algorithm based on the visibility formulation defined in section 3.2 to perform linear time PCS pruning. In particular, we use Lemma 2 to compute the PCS. In the first pass, we clear the depth buffer and render the objects in the order $O_1, .., O_n$ along with image space occlusion queries. In other words, for each object in $O_1, .., O_n$, we render the object and test if it is fully visible with respect to the objects rendered prior to it. In the second pass, we clear the depth buffer and render the objects in the reverse order $O_n, O_{n-1}, ...O_1$ along with image space occlusion queries. We perform the same operations as in the first pass while rendering each object. At the end of each pass, we test if an object is fully visible or not. An object classified as fully-visible in both the passes does not belong to PCS. It should also be noted that our pruning algorithm is less conservative as compared to an algorithm that only uses 2D screen-space overlap tests.

4.2 Visibility Queries

Our visibility based PCS computation algorithm needs a hardware visibility query which determines if a primitive is *fully-visible* or not. Current commodity graphics hardware supports an image-space occlusion query that checks whether a primitive is visible or not. These queries scan-convert the specified primitives and determine if the depth of any pixel changes. Various implementations are provided by different hardware vendors and each implementation varies in its performance as well as functionality. Some of the well-known occlusion queries based on the OpenGL extensions include the GL_HP_occlusion_test (`http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion\_test.txt`) and the NVIDIA OpenGL extension GL_NV_occlusion_query (`http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion\_query.txt`). The GL_HP_occlusion_test returns a boolean determining if any incoming fragment passed the depth test, whereas the GL_NV_occlusion_query returns the number of incoming fragments which passed the depth test.

We need a query that tests if all the incoming fragments of a primitive have a depth value *less* than the depth of the corresponding fragments in the frame buffer. In order to support such a query, we change the depth test to pass only if the depth of the incoming fragment is greater than or equal to the depth of the corresponding fragment in the frame buffer. With this depth comparison function, we use an image space occlusion query to test if a primitive is not visible when rendered against the depth buffer. If the primitive is classified as not visible, then each incoming fragment has a depth value less than the corresponding depth value in the frame buffer, thus providing a visibility query to test if a primitive is fully visible. Note that we need to disable the depth writes so that the change of depth function does not affect the depth buffer. We refer to these queries as *fully-visibility* queries in the rest of the paper.

These queries can sometimes stall the graphics pipeline while waiting for results. We describe techniques to avoid these stalls (discussed in section 4.5).

Occlusion queries perform at the rate of rasterization hardware and involve very low bandwidth requirements in comparison to frame buffer readbacks. If we perform n occlusion queries, we readback n integers for a total of $4n$ bytes, sent to the host CPU using PCI interface. Also, the bandwidth requirement for n occlusion queries is independent of the resolution of the frame buffer.

4.3 Multiple Level Pruning

We extend the visibility pruning algorithm to sub-objects, to identify the potentially intersecting regions among the objects in PCS. We use Lemma 3 to perform sub-object level pruning. We render each sub-object for every object in the PCS with a full visibility query. The sub-object could be a bounding box, a group of triangles, or even a single triangle.

The following is the pseudocode of the algorithm.

```
• First pass:

    1. Clear the depth buffer (use orthographic projection)

    2. For each object $O_i$, $i = 1, .., n$
        – Disable depth mask and set the depth function to
          GL_GEQUAL.
        – For each sub-object $T^i_k$ in $O_i$
              Render $T^i_k$ using an occlusion query
        – Enable the depth mask and set the depth function
          to GL_LEQUAL.
        – For each sub-object $T^i_k$ in $O_i$
              Render $T^i_k$ using an occlusion query
    3. For each object $O_i$, $i = 1, .., n$
        – For each sub-object $T^i_k$ in $O_i$
              Test if $T^i_k$ is not visible with respect to the
              depth buffer. If it is not visible, set a tag
              to note it as fully visible.

• Second pass:

    Same as First pass, except that the two "For each
    object" loops are run with $i = n, .., 1$.
```

4.4 Collision Detection

The overall collision detection algorithm performs object level pruning, sub-object level pruning and triangle intersection tests among the objects in PCS.

4.4.1 Object level pruning

We perform object level pruning to compute the PCS of objects. Initially, all the objects belong to the PCS. We first perform pruning against each coordinate axis using the axis-aligned bounding boxes as the object's representation for collision detection. The pruning is performed till the PCS does not change between successive iterations. We also use the object's triangulated representation for further pruning the PCS. The size of the resulting set is usually small and a simple all-pair bounding box overlap tests are used to compute the potentially intersecting pairs. If the size of this set is large, then we use sweep-and-prune technique [Cohen et al. 1995] to prune this set instead of all-pair tests. The sweep-and-prune algorithm projects the bounding boxes on each axis, sorts them and check them for possible overlap.

4.4.2 Sub-Object level pruning

We perform multiple level pruning to identify the potentially intersecting triangles among the objects in the PCS. We group adjacent local triangles (say k triangles) to form a sub-object used in multi-level pruning and prune the potential regions considerably. This improves the performance of our algorithm as performing a fully-visibility query for each single triangle in the PCS of objects can be expensive. At the next level, we consider the PCS of sub-objects and perform pruning using each triangle as a sub-object. The multiple-level sub-object pruning is performed across each axis by using a visibility query for sub-object.

4.4.3 Intersection Tests

We perform exact collision detection between the objects involved in the potentially colliding pairs by testing their potentially intersecting triangles.

4.5 Optimizations

In this section, we highlight a number of optimizations used to improve the performance of the algorithm.

- **AABBs and Orthographic Projections**: We use orthographic projection of axis-aligned bounding boxes. These could potentially provide an improvement factor of six in the rendering performance. Orthographic projection is used for its speed and simplicity. Also, it reduces the resolution errors due to perspective transformation. In addition, we use axis-aligned bounding boxes to prune the objects for intersection tests.

- **Visibility Query returning Z-fail**: A hardware visibility query providing z-fail (in particular, a query to test if z-fail is non-zero) would reduce the amount of rendering by a factor of two for AABBs under orthographic projections. This query allows us to update depth buffer along with the occlusion query, thus providing an improvement of two times. We take additional care in terms of ordering the view-axis perpendicular faces of the bounding boxes, and ensure that the results are not affected by possible self-occlusion. thus not affecting the query result by self-occlusion.

- **Avoid Stalls**: We utilize GL_NV_occlusion_query to avoid stalls in the graphics pipeline. We query the results of the occlusion tests at the end of each pass, improving the performance by a factor of four when compared to a system using GL_HP_occlusion_test.

- **Rendering Acceleration**: We use vertex arrays in video memory to improve the rendering performance by copying the object representation to the video memory. The performance can be further improved by representing the objects in terms of triangle strips and using them along with vertex arrays.

5 Implementation and Performance

We have implemented our system on a Dell Precision Workstation consisting of a 2.4 GHz Pentium 4 CPU and a GeForce FX Ultra 5,800 GPU. We are able to perform around $400K$ image-space occlusion queries per second on this card. We have tested our system on four complex simulated environments.

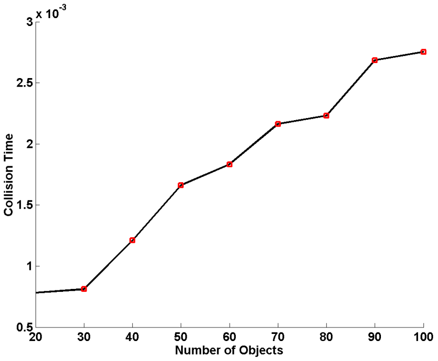


Figure 4: *Number of objects v/s Average collision pruning time: This graph highlights the relationship between number of objects in the scene and the average collision pruning time (object pruning and sub-object/triangle pruning). Each object is composed of* 200 *triangles. The graph indicates that the collision pruning time is linear to the number of objects.*

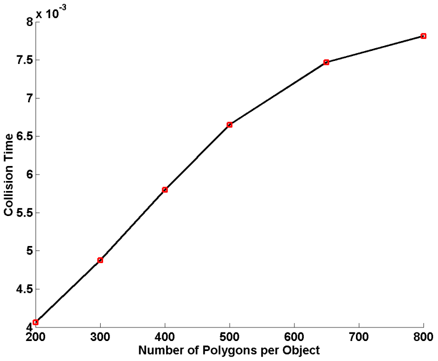


Figure 5: *Polygons per object vs/ Average collision query time : Graph shows the linear relationship between the number of polygons per object and the average collision pruning time. This scene is composed of* 100 *deforming cylinders and has a density of* $1-2\%$. *The collision pruning time is averaged over* 500 *frames and at an image-space resolution of* 800×800

- **Environment 1:** It consists of 100 deformable moving objects in a unit cube with a density of $5-6\%$. Each object consists of 200 triangles. The average collision pruning time is around 4ms at an image-space resolution of 800×800. A snapshot from this environment is shown in Fig. 2(a).

- **Environment 2:** It consists of six deformable cylinders, each composed of 20,000 triangles. The scene has an estimated density of $6-8\%$. The average collision pruning query time is around 8ms. A snapshot from this environment is shown in Fig. 2(b).

- **Environment 3:** It consists of two highly tessellated models: a bunny (35K triangles) and a dragon (250K triangles). In Fig. 2(c), we show a relative configuration of the two models and different colors are used to highlight the triangles that belong to the PCS. A zoomed-up view of the intersection region is shown in Fig. 2(d). It takes about 40 ms to perform the collision queries.

- **Breaking Objects:** We used our collision detection algorithm to generate a real-time simulation of breaking objects. Fig. 3 highlights a sequence from our dynamic simulation with the bunny and the dragon colliding and decomposing the dragon into multiple objects due to the impact. The total number of objects and the number of triangles in each object are changing over the course of the simulation. Earlier collision detection algorithms are unable to handle such scenarios in real-time. Our collision detection algorithm takes about 35ms (on average) to compute all the overlapping triangles during each frame.

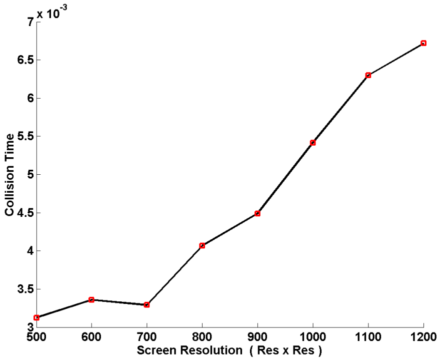


Figure 6: *Image-space resolution v/s Average collision query time : Graph indicating the linear relationship between screen resolution and average collision query time. The scene consists of* 100 *deformable cylinders and each object is composed of* 200 *triangles.*

5.1 Performance Analysis

We have tested the performance of our algorithm and system on different benchmarks. Its overall performance depends on a few key parameters: The following are the key parameters determining the performance of the system.

- **Number of objects :** Our object level pruning algorithm exhibits linear time performance in our benchmarks. We have performed timing analysis with varying number of deformable objects and Fig. 4 summarizes the results. In our simulations, we have observed that the pruning algorithm requires only a few iterations to converge (typically, it is two). Also, each iteration reduces the size of PCS. Therefore, the visibility based pruning algorithm traverses a smaller list of objects during subsequent iterations.

- **Triangle count per object :** The performance of our system depends upon the triangle count of the potentially interfering objects. We have tested our system with simulations on 100 deformable objects consisting of varying triangle count. Fig. 5 summarizes the results. The graph indicates a linear relationship between the polygon count and the average collision query time. Moreover, the number of polygons per object is much higher than the number of objects in the scene.

- **Accuracy and Image-Space Resolution :** The accuracy of the overall algorithm is governed by image-space resolution. Typically a higher resolution leads to higher fill-rate requirements, in terms of rendering the primitives, bounding boxes and performing occlusion queries. A lower image-space resolution can improve the query time, but can miss intersections between two objects, whose boundaries are barely touching. Figure 6 highlights the relationship between collision pruning time and the screen resolution.

- **Output Size:** The performance of any collision detection algorithm varies as a function of the output size, i.e. the number of overlapping triangle pairs. In our case, the performance varies as a linear function of the size of PCS after object level pruning and sub-object level pruning as well as the number of triangle pairs that need to be tested for exact collision. In case two objects have a deep penetration, the output size can be high and therefore the size of each PCS can be high as well.

- **Rasterization optimizations:** The performance of the system is accelerated using the rasterization optimizations discussed in section 4.5. We have used AABBs with orthographic projections for our pruning algorithms. We have used immediate mode for rendering the models and breakable objects and used GL_NV_occlusion_query to maximize the performance.

5.2 Comparison with Other Approaches

Collision detection is well-studied in the literature and a number of algorithms and public-domain systems are known. However, none of the earlier algorithms provide the same capabilities or features as our algorithm based on PCS computation. As a result, we have not performed any direct timing comparisons with the earlier systems. We just compare some of the features of our approach with the earlier algorithms.

Object-Space Algorithms: Algorithms based on sweep-and-prune are known for N-body collision detection [Cohen et al. 1995]. They have been used in I-COLLIDE, V-COLLIDE, SWIFT, SOLID and other systems. However, these algorithms were designed for rigid bodies and compute a tight fitting AABB for each object using incremental methods, followed by sorting their projections of AABBs along each axis. It is not clear whether they can perform real-time collision detection on large environments composed of deformable models. On the other hand, our algorithm performs two passes on the object list to perform the PCS. We expect that our PCS based algorithm is more conservative as compared to sweep-and-prune. Furthermore, the accuracy of our approach is governed by the image-space resolution.

A number of hierarchical methods have been proposed to check two highly tessellated models for overlap and some optimized systems (e.g. RAPID, QuickCD) are available. They involve considerable preprocessing and memory overhead in generating the hierarchy and won't work well on deformable models.

Image-Space Algorithms : These include algorithms based on stencil buffer techniques as well as distance field computations. Some systems such as PIVOT are able to perform other proximity queries including distance and local penetration computation, whereas our PCS based algorithm is limited to only check for interference. However, our algorithm only needs to readback the output of a visibility query and not the entire depth-buffer or stencil buffer. This significantly improves its performance, especially when we use higher image-space precision. Unlike earlier algorithms, our PCS-based algorithm is applicable to all triangulated 3D models (and not just closed objects) and can handle arbitrary number of objects in the environment and not just two objects.

5.3 Conclusions and Future Work

We have presented a novel algorithm for collision detection between multiple deformable objects in a large environment. Our algorithm is applicable to all triangulated models, makes no assumption about object motion and can compute all the contacts up to image-space resolution. It is based on a novel, linear-time PCS computation algorithm that is applied iteratively to the objects and sub-objects. The PCS is computed using image-space visibility queries that are widely available on commodity graphics hardware. It only needs to readback the results of a query and not the entire frame-buffer or depth-buffer.

Limitations: Our current approach has many limitations. Firstly it only check overlapping objects, and not the distance or penetration information that is needed for many applications. Secondly, its accuracy is governed by the image-space resolution. Finally, it cannot handle self-collisions within an object.

There are many avenues for future work. In addition to overcoming these limitations, we will to investigate some hybrid object and image-space combinations that can utilize the efficiency of the image-space methods with the accuracy of object-space approaches. We will use our PCS based collision detection for more applications and also like to evaluate its impact on the accuracy of the overall simulation. Finally, we will like to investigate use of the programmability features of graphics hardware to design improved algorithms for collision and proximity queries.

References

Baciu, G., and Wong, S. 2002. Image-based techniques in a hybrid collision detector. *IEEE Trans. on Visualization and Computer Graphics*.

Baciu, G., Wong, S., and Sun, H. 1998. Recode: An image-based collision detection algorithm. *Proc. of Pacific Graphics*, 497–512.

Cohen, J., Lin, M., Manocha, D., and Ponamgi, M. 1995. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proc. of ACM Interactive 3D Graphics Conference*, 189–196.

Cohen-Or, D., Chrysanthou, Y., and Silva, C. 2001. A survey of visibility for walkthrough applications. *SIGGRAPH Course Notes # 30*.

Goldfeather, J., Molnar, S., Turk, G., and Fuchs, H. 1989. Near real-time csg rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications 9*, 3, 20–28.

Hoff, K., Culver, T., Keyser, J., Lin, M., and Manocha, D. 1999. Fast computation of generalized voronoi diagrams using graphics hardw are. *Proceedings of ACM SIGGRAPH*, 277–286.

Hoff, K., Zaferakis, A., Lin, M., and Manocha, D. 2001. Fast and simple geometric proximity queries using graphics hardware. *Proc. of ACM Symposium on Interactive 3D Graphics*.

Hubbard, P. M. 1993. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality.*

Kaul, A., and Rossignac, J. 1992. Solid-interpolating deformations: construction and animation of pips. *Computer and Graphics 16*, 107–116.

Kim, Y., Lin, M., and Manocha, D. 2002. Deep: An incremental algorithm for penetration depth computation between convex polytopes. *Proc. of IEEE Conference on Robotics and Automation.*

Kim, Y., Otaduy, M., Lin, M., and Manocha, D. 2002. Fast penetration depth computation using rasterization hardware and hierarchical refinement. *Proc. of Workshop on Algorithmic Foundations of Robotics*.

Klosowski, J., Held, M., Mitchell, J., Sowizral, H., and Zikan, K. 1998. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics 4*, 1, 21–37.

Lin, M., and Gottschalk, S. 1998. Collision detection between geometric models: A survey. *Proc. of IMA Conference on Mathematics of Surfaces*.

Lombardo, J. C., Cani, M.-P., and Neyret, F. 1999. Real-time collision detection for virtual surgery. *Proc. of Computer Animation*.

Myszkowski, K., Okunev, O. G., and Kunii, T. L. 1995. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer 11*, 9, 497–512.

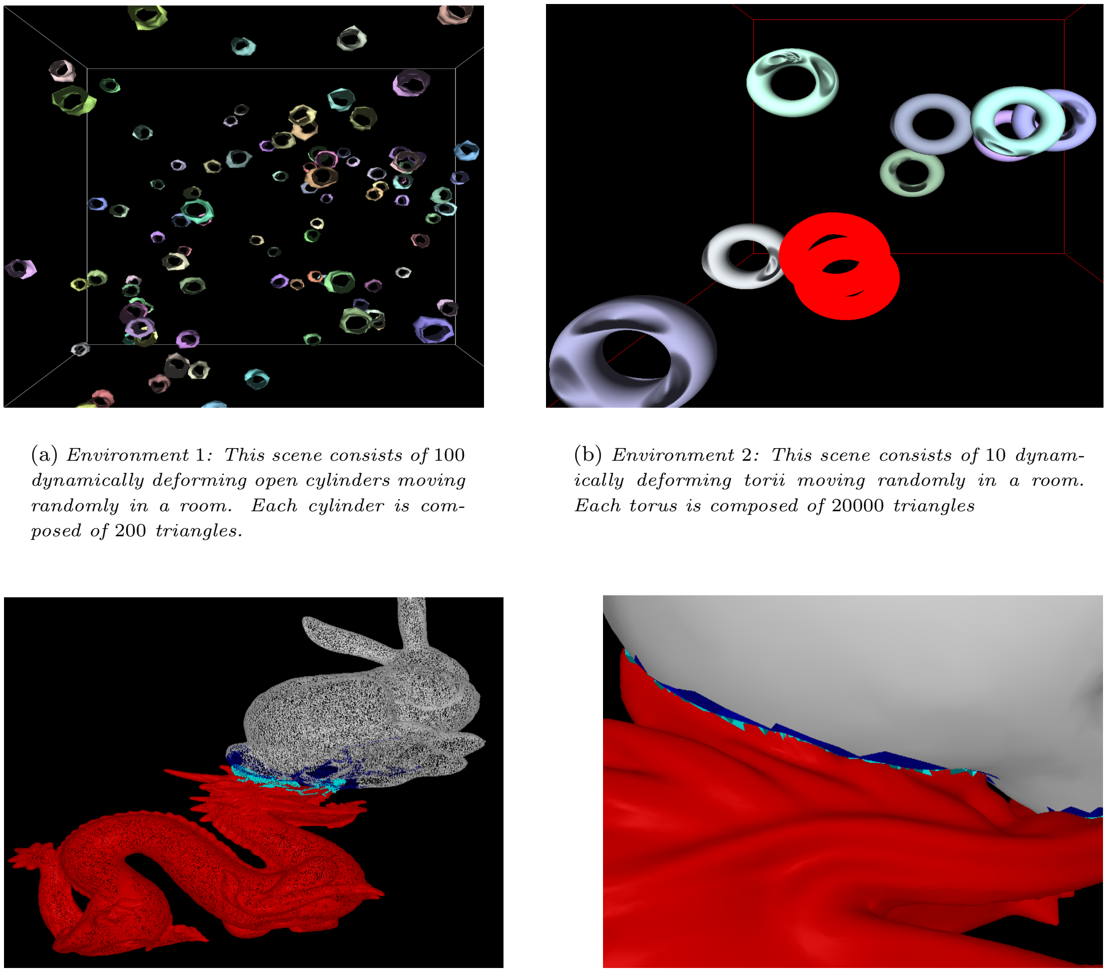
REDON, S., KHEDDAR, A., AND COQUILLART, S. 2002. Fast continuous collision detection between rigid bodies. *Proc. of Eurographics (Computer Graphics Forum)*.

ROSSIGNAC, J., AND WU, J. 1990. Correct shading of regularized csg solids using a depth-interval buffer. In *Eurographics Workshop on Graphics Hardware*.

ROSSIGNAC, J., MEGAHED, A., AND SCHNEIDER, B. 1992. Interactive inspection of solids: cross-sections and interferences. In *Proceedings of ACM Siggraph*, 353–60.

SHINYA, M., AND FORGUE, M. C. 1991. Interference detection through rasterization. *The Journal of Visualization and Computer Animation 2*, 4, 131–134.

VASSILEV, T., SPANLANG, B., AND CHRYSANTHOU, Y. 2001. Fast cloth animation on walking avatars. *Computer Graphics Forum (Proc. of Eurographics'01) 20*, 3, 260–267.



(a) *Environment* 1*: This scene consists of* 100 *dynamically deforming open cylinders moving randomly in a room. Each cylinder is composed of* 200 *triangles.*

(b) *Environment* 2*: This scene consists of* 10 *dynamically deforming torii moving randomly in a room. Each torus is composed of* 20000 *triangles*

(c) *Environment* 3*: Wired frame of dragon and bunny rendered in the following colors - {cyan,blue} highlight triangles in the PCS, {red, white} illustrate portions not in the PCS. The dragon consists of* $250K$ *triangles and the bunny consists of* $35K$ *faces*

(d) *Environment* 3*: Zoomed view highlighting the exact intersections between the triangles in the PCS. The configuration of the two objects is the same as Figure 2(c). The cyan and blue triangles are the overlapping triangles of the dragon and bunny, respectively.*

Figure 2: Snapshots of our interactive collision algorithm on three complex environments. The algorithm takes 4,8, 40ms respectively on each of the environments to perform collision queries on a GeForce FX 5800 Ultra with an image-space resolution of 800 × 800.

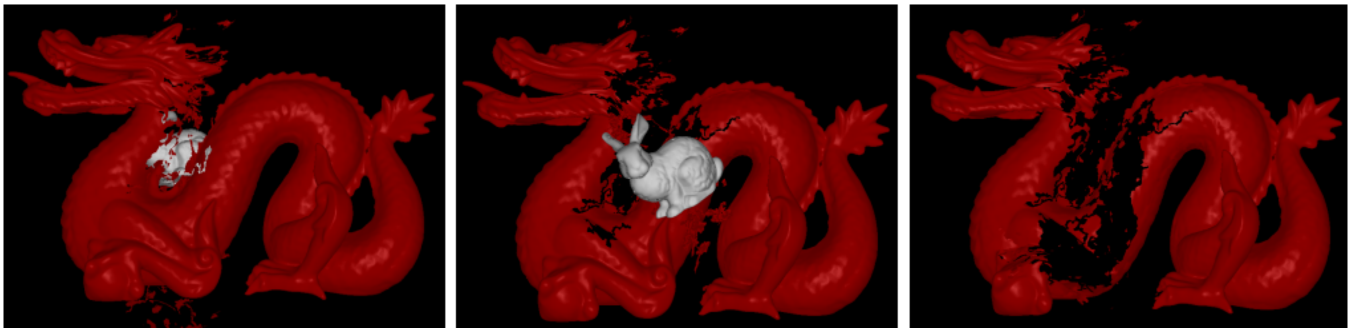


Figure 3: *Environment with breakable objects: As the bunny (with* $35K$ *triangles), falls through the dragon (with* $250K$*), the number of objects in the scene (shown with a blue outline) and the triangle count within each object changes. Our algorithm computes all the overlapping triangles during each frame. The average collision query time is* 35 *milli-seconds per frame.*

Ray Tracing and Global Illumination using Graphics Hardware

Timothy J. Purcell
Stanford University

Ray Tracing on Programmable Graphics Hardware

Timothy J. Purcell   Ian Buck   William R. Mark [*]   Pat Hanrahan

Stanford University [†]

Abstract

Recently a breakthrough has occurred in graphics hardware: fixed function pipelines have been replaced with programmable vertex and fragment processors. In the near future, the graphics pipeline is likely to evolve into a general programmable stream processor capable of more than simply feed-forward triangle rendering.

In this paper, we evaluate these trends in programmability of the graphics pipeline and explain how ray tracing can be mapped to graphics hardware. Using our simulator, we analyze the performance of a ray casting implementation on next generation programmable graphics hardware. In addition, we compare the performance difference between non-branching programmable hardware using a multipass implementation and an architecture that supports branching. We also show how this approach is applicable to other ray tracing algorithms such as Whitted ray tracing, path tracing, and hybrid rendering algorithms. Finally, we demonstrate that ray tracing on graphics hardware could prove to be faster than CPU based implementations as well as competitive with traditional hardware accelerated feed-forward triangle rendering.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

Keywords: Programmable Graphics Hardware, Ray Tracing

1 Introduction

Real-time ray tracing has been a goal of the computer-graphics community for many years. Recently VLSI technology has reached the point where the raw computational capability of a single chip is sufficient for real-time ray tracing. Real-time ray tracing has been demonstrated on small scenes on a single general-purpose CPU with SIMD floating point extensions [Wald et al. 2001b], and for larger scenes on a shared memory multiprocessor [Parker et al. 1998; Parker et al. 1999] and a cluster [Wald et al. 2001b; Wald et al. 2001a]. Various efforts are under way to develop chips specialized for ray tracing, and ray tracing chips that accelerate off-line rendering are commercially available [Hall 2001]. Given that real-time ray tracing is possible in the near future, it is worthwhile to study implementations on different architectures with the goal of providing maximum performance at the lowest cost.

In this paper, we describe an alternative approach to real-time ray tracing that has the potential to out perform CPU-based algorithms without requiring fundamentally new hardware: using commodity programmable graphics hardware to implement ray tracing. Graphics hardware has recently evolved from a fixed-function graphics pipeline optimized for rendering texture-mapped triangles to a graphics pipeline with programmable vertex and fragment stages. In the near-term (next year or two) the graphics processor (GPU) fragment program stage will likely be generalized to include floating point computation and a complete, orthogonal instruction set. These capabilities are being demanded by programmers using the current hardware. As we will show, these capabilities are also sufficient for us to write a complete ray tracer for this hardware. As the programmable stages become more general, the hardware can be considered to be a general-purpose stream processor. The stream processing model supports a variety of highly-parallelizable algorithms, including ray tracing.

In recent years, the performance of graphics hardware has increased more rapidly than that of CPUs. CPU designs are optimized for high performance on sequential code, and it is becoming increasingly difficult to use additional transistors to improve performance on this code. In contrast, programmable graphics hardware is optimized for highly-parallel vertex and fragment shading code [Lindholm et al. 2001]. As a result, GPUs can use additional transistors much more effectively than CPUs, and thus sustain a greater rate of performance improvement as semiconductor fabrication technology advances.

The convergence of these three separate trends – sufficient raw performance for single-chip real-time ray tracing; increasing GPU programmability; and faster performance improvements on GPUs than CPUs – make GPUs an attractive platform for real-time ray tracing. GPU-based ray tracing also allows for hybrid rendering algorithms; e.g. an algorithm that starts with a Z-buffered rendering pass for visibility, and then uses ray tracing for secondary shadow rays. Blurring the line between traditional triangle rendering and ray tracing allows for a natural evolution toward increased realism.

In this paper, we show how to efficiently implement ray tracing on GPUs. The paper contains three main contributions:

- We show how ray tracing can be mapped to a stream processing model of parallel computation. As part of this mapping, we describe an efficient algorithm for mapping the innermost ray-triangle intersection loop to multiple rendering passes. We then show how the basic ray caster can be extended to include shadows, reflections, and path tracing.

- We analyze the streaming GPU-based ray caster's performance and show that it is competitive with current CPU-based ray casting. We also show initial results for a system including secondary rays. We believe that in the near future, GPU-based ray tracing will be much faster than CPU-based ray tracing.

- To guide future GPU implementations, we analyze the compute and memory bandwidth requirements of ray casting on GPUs. We study two basic architectures: one architecture without branching that requires multiple passes, and another with branching that requires only a single pass. We show that

[*] Currently at NVIDIA Corporation

[†] {tpurcell, ianbuck, billmark, hanrahan}@graphics.stanford.edu

the single pass version requires significantly less bandwidth, and is compute-limited. We also analyze the performance of the texture cache when used for ray casting and show that it is very effective at reducing bandwidth.

2 Programmable Graphics Hardware

2.1 The Current Programmable Graphics Pipeline

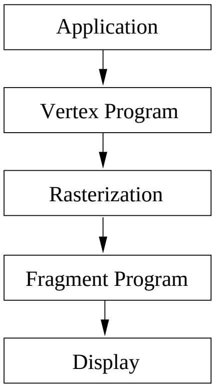


Figure 1: The programmable graphics pipeline.

A diagram of a modern graphics pipeline is shown in figure 1. Today's graphics chips, such as the NVIDIA GeForce3 [NVIDIA 2001] and the ATI Radeon 8500 [ATI 2001] replace the fixed-function vertex and fragment (including texture) stages with programmable stages. These programmable vertex and fragment engines execute user-defined programs and allow fine control over shading and texturing calculations. An NVIDIA vertex program consists of up to 128 4-way SIMD floating point instructions [Lindholm et al. 2001]. This vertex program is run on each incoming vertex and the computed results are passed on to the rasterization stage. The fragment stage is also programmable, either through NVIDIA register combiners [Spitzer 2001] or DirectX 8 pixel shaders [Microsoft 2001]. Pixel shaders, like vertex programs, provide a 4-way SIMD instruction set augmented with instructions for texturing, but unlike vertex programs operate on fixed-point values. In this paper, we will be primarily interested in the programmable fragment pipeline; it is designed to operate at the system fill rate (approximately 1 billion fragments per second).

Programmable shading is a recent innovation and the current hardware has many limitations:

- Vertex and fragment programs have simple, incomplete instruction sets. The fragment program instruction set is much simpler than the vertex instruction set.

- Fragment program data types are mostly fixed-point. The input textures and output framebuffer colors are typically 8-bits per color component. Intermediate values in registers have only slightly more precision.

- There are many resource limitations. Programs have a limited number of instructions and a small number of registers. Each stage has a limited number of inputs and outputs (e.g. the number of outputs from the vertex stage is constrained by the number of vertex interpolants).

- The number of active textures and the number of dependent textures is limited. Current hardware permits certain instructions for computing texture addresses only at certain points within the program. For example, a DirectX 8 PS 1.4 pixel shader has two stages: a first texture addressing stage consisting of four texture fetch instructions followed by eight color blending instructions, and then a color computation stage consisting of additional texture fetches followed by color combining arithmetic. This programming model permits a single level of dependent texturing.

- Only a single color value may be written to the framebuffer in each pass.

- Programs cannot loop and there are no conditional branching instructions.

2.2 Proposed Near-term Programmable Graphics Pipeline

The limitations of current hardware make it difficult to implement ray tracing in a fragment program. Fortunately, due to the interest in programmable shading for mainstream game applications, programmable pipelines are rapidly evolving and many hardware and software vendors are circulating proposals for future hardware. In fact, many of the current limitations are merely a result of the fact that they represent the very first generation of programmable hardware. In this paper, we show how to implement a ray tracer on an extended hardware model that we think approximates hardware available in the next year or two. Our model is based loosely on proposals by Microsoft for DirectX 9.0 [Marshall 2001] and by 3DLabs for OpenGL 2.0 [3DLabs 2001].

Our target baseline architecture has the following features:

- A programmable fragment stage with floating point instructions and registers. We also assume floating point texture and framebuffer formats.

- Enhanced fragment program assembly instructions. We include instructions which are now only available at the vertex level. Furthermore, we allow longer programs; long enough so that our basic ray tracing components may be downloaded as a single program (our longest program is on the order of 50 instructions).

- Texture lookups are allowed anywhere within a fragment program. There are no limits on the number of texture fetches or levels of texture dependencies within a program.

- Multiple outputs. We allow 1 or 2 floating point RGBA (4-vectors) to be written to the framebuffer by a fragment program. We also assume the fragment program can render directly to a texture or the stencil buffer.

We consider these enhancements a natural evolution of current graphics hardware. As already mentioned, all these features are actively under consideration by various vendors.

At the heart of any efficient ray tracing implementation is the ability to traverse an acceleration structure and test for an intersection of a ray against a list of triangles. Both these abilities require a looping construct. Note that the above architecture does not include data-dependent conditional branching in its instruction set. Despite this limitation, programs with loops and conditionals can be mapped to this baseline architecture using the multipass rendering technique presented by Peercy et al. [2000]. To implement a conditional using their technique, the conditional predicate is first evaluated using a sequence of rendering passes, and then a stencil bit is set to true or false depending on the result. The body of the conditional is then evaluated using additional rendering passes, but values are only written to the framebuffer if the corresponding fragment's stencil bit is true.

Although their algorithm was developed for a fixed-function graphics pipeline, it can be extended and used with a programmable pipeline. We assume the addition of two hardware features to make the Peercy et al. algorithm more efficient: direct setting of stencil bits and an early fragment kill similar to Z occlusion culling [Kirk 2001]. In the standard OpenGL pipeline, stencil bits may be set by testing the alpha value. The alpha value is computed by the fragment program and then written to the framebuffer. Setting the stencil bit from the computed alpha value requires an additional pass. Since fragment programs in our baseline architecture can modify the stencil values directly, we can eliminate this extra pass. Another important rendering optimization is an early fragment kill. With an early fragment kill, the depth or stencil test is executed before the fragment program stage and the fragment program is executed only if the fragment passes the stencil test. If the stencil bit is false, no instructions are executed and no texture or framebuffer bandwidth is used (except to read the 8-bit stencil value). Using the combination of these two techniques, multipass rendering using large fragment programs under the control of the stencil buffer is quite efficient.

As we will see, ray tracing involves significant looping. Although each rendering pass is efficient, extra passes still have a cost; each pass consumes extra bandwidth by reading and writing intermediate values to texture (each pass also requires bandwidth to read stencil values). Thus, fewer resources would be used if these inner loops over voxels and triangles were coalesced into a single pass. The obvious way to do this would be to add branching to the fragment processing hardware. However, adding support for branching increases the complexity of the GPU hardware. Non-branching GPUs may use a single instruction stream to feed several fragment pipelines simultaneously (SIMD computation). GPUs that support branching require a separate instruction stream for each processing unit (MIMD computation). Therefore, graphics architects would like to avoid branching if possible. As a concrete example of this trade off, we evaluate the efficiency of ray casting on two architectures, one with and one without branching:

- **Multipass Architecture**. Supports arbitrary texture reads, floating-point texture and framebuffer formats, general floating point instructions, and two floating point 4-vector outputs. Branching is implemented via multipass rendering.

- **Branching Architecture**. Multipass architecture enhanced to include support for conditional branching instructions for loops and control flow.

2.3 The Streaming Graphics Processor Abstraction

As the graphics processor evolves to include a complete instruction set and larger data types, it appears more and more like a general-purpose processor. However, the challenge is to introduce programmability without compromising performance, for otherwise the GPU would become more like the CPU and lose its cost-performance advantages. In order to guide the mapping of new applications to graphics architectures, we propose that we view next-generation graphics hardware as a *streaming processor*. Stream processing is not a new idea. Media processors transform streams of digital information as in MPEG video decode. The IMAGINE processor is an example of a general-purpose streaming processor [Khailany et al. 2000].

Streaming computing differs from traditional computing in that the system reads the data required for a computation as a sequential *stream* of elements. Each element of a stream is a record of data requiring a similar computation. The system executes a program or *kernel* on each element of the input stream placing the result on an output stream. In this sense, a programmable graphics processor executes a vertex program on a stream of vertices, and a fragment program on a stream of fragments. Since, for the most part we are ignoring vertex programs and rasterization, we are treating the graphics chip as basically a streaming fragment processor.

The streaming model of computation leads to efficient implementations for three reasons. First, since each stream element's computation is independent from any other, designers can add additional pipelines that process elements of the stream in parallel. Second, kernels achieve high arithmetic intensity. That is, they perform a lot of computation per small fixed-size record. As a result the computation to memory bandwidth ratio is high. Third, streaming hardware can hide the memory latency of texture fetches by using prefetching [Torborg and Kajiya 1996; Anderson et al. 1997; Igehy et al. 1998]. When the hardware fetches a texture for a fragment, the fragment registers are placed in a FIFO and the fragment processor starts processing another fragment. Only after the texture is fetched does the processor return to that fragment. This method of hiding latency is similar to multithreading [Alverson et al. 1990] and works because of the abundant parallelism in streams. In summary, the streaming model allows graphics hardware to exploit parallelism, to utilize bandwidth efficiently, and to hide memory latency. As a result, graphics hardware makes efficient use of VLSI resources.

The challenge is then to map ray tracing onto a streaming model of computation. This is done by breaking the ray tracer into kernels. These kernels are chained together by streams of data, originating from data stored in textures and the framebuffer.

3 Streaming Ray Tracing

In this section, we show how to reformulate ray tracing as a streaming computation. A flow diagram for a streaming ray tracer is found in figure 2.

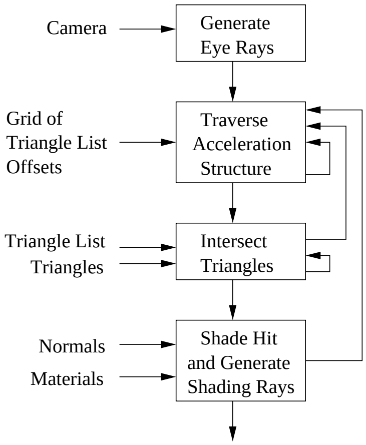

Camera
Generate Eye Rays
Grid of Triangle List Offsets
Traverse Acceleration Structure
Triangle List
Triangles
Intersect Triangles
Normals
Materials
Shade Hit and Generate Shading Rays

Figure 2: A streaming ray tracer.

In this paper, we assume that all scene geometry is represented as triangles stored in an acceleration data structure before rendering begins. In a typical scenario, an application would specify the scene geometry using a display list, and the graphics library would place the display list geometry into the acceleration data structure. We will not consider the cost of building this data structure. Since this may be an expensive operation, this assumption implies that the algorithm described in this paper may not be efficient for dynamic scenes.

The second design decision was to use a uniform grid to accelerate ray tracing. There are many possible acceleration data structures to choose from: bounding volume hierarchies, bsp trees, k-d trees, octrees, uniform grids, adaptive grids, hierarchical grids, etc. We chose uniform grids for two reasons. First, many experiments have been performed using different acceleration data structures on different scenes (for an excellent recent study see Havran et al. [2000]). From these studies no single acceleration data structure appears to be most efficient; all appear to be within a factor of two of each other. Second, uniform grids are particularly simple for hardware implementations. Accesses to grid data structures require constant time; hierarchical data structures, in contrast, require variable time per access and involve pointer chasing. Code for grid traversal is also very simple and can be highly optimized in hardware. In our system, a grid is represented as a 3D texture map, a memory organization currently supported by graphics hardware. We will discuss further the pros and cons of the grid in section 5.

We have split the streaming ray tracer into four kernels: eye ray generation, grid traversal, ray-triangle intersection, and shading. The eye ray generator kernel produces a stream of viewing rays. Each viewing ray is a single ray corresponding to a pixel in the image. The traversal kernel reads the stream of rays produced by the eye ray generator. The traversal kernel steps rays through the grid until the ray encounters a voxel containing triangles. The ray and voxel address are output and passed to the intersection kernel. The intersection kernel is responsible for testing ray intersections with all the triangles contained in the voxel. The intersector has two types of output. If ray-triangle intersection (hit) occurs in that voxel, the ray and the triangle that is hit is output for shading. If no hit occurs, the ray is passed back to the traversal kernel and the search for voxels containing triangles continues. The shading kernel computes a color. If a ray terminates at this hit, then the color is written to the accumulated image. Additionally, the shading kernel may generate shadow or secondary rays; in this case, these new rays are passed back to the traversal stage.

We implement ray tracing kernels as fragment programs. We execute these programs by rendering a screen-sized rectangle. Constant inputs are placed within the kernel code. Stream inputs are fetched from screen-aligned textures. The results of a kernel are then written back into textures. The stencil buffer controls which fragments in the screen-sized rectangle and screen-aligned textures are active. The 8-bit stencil value associated with each ray contains the ray's state. A ray's state can be *traversing*, *intersecting*, *shading*, or *done*. Specifying the correct stencil test with a rendering pass, we can allow the kernel to be run on only those rays which are in a particular state.

The following sections detail the implementation of each ray tracing kernel and the memory layout for the scene. We then describe several variations including ray casting, Whitted ray tracing [Whitted 1980], path tracing, and shadow casting.

3.1 Ray Tracing Kernels

3.1.1 Eye Ray Generator

The eye ray generator is the simplest kernel of the ray tracer. Given camera parameters, including viewpoint and a view direction, it computes an eye ray for each screen pixel. The fragment program is invoked for each pixel on the screen, generating an eye ray for each. The eye ray generator also tests the ray against the scene bounding box. Rays that intersect the scene bounding box are processed further, while those that miss are terminated.

3.1.2 Traverser

The traversal stage searches for voxels containing triangles. The first part of the traversal stage sets up the traversal calculation. The second part steps along the ray enumerating those voxels pierced by the ray. Traversal is equivalent to 3D line drawing and has a per-ray setup cost and a per-voxel rasterization cost.

We use a 3D-DDA algorithm [Fujimoto et al. 1986] for this traversal. After each step, the kernel queries the grid data structure which is stored as a 3D texture. If the grid contains a null pointer, then that voxel is empty. If the pointer is not null, the voxel contains triangles. In this case, a ray-voxel pair is output and the ray is marked so that it is tested for intersection with the triangles in that voxel.

Implementing the traverser loop on the multipass architecture requires multiple passes. The once per ray setup is done as two passes and each step through a voxel requires an additional pass. At the end of each pass, the fragment program must store all the stepping parameters used within the loop to textures, which then must be read for the next pass. We will discuss the multipass implementation further after we discuss the intersection stage.

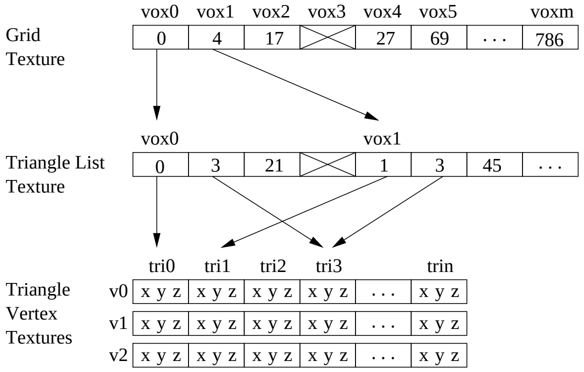

Grid Texture
vox0 vox1 vox2 vox3 vox4 vox5 voxm
0 4 17 27 69 . . . 786
Triangle List Texture
vox0 vox1
0 3 21 1 3 45 . . .
Triangle Vertex Textures
tri0 tri1 tri2 tri3 trin
v0 x y z x y z x y z x y z . . . x y z
v1 x y z x y z x y z x y z . . . x y z
v2 x y z x y z x y z x y z . . . x y z

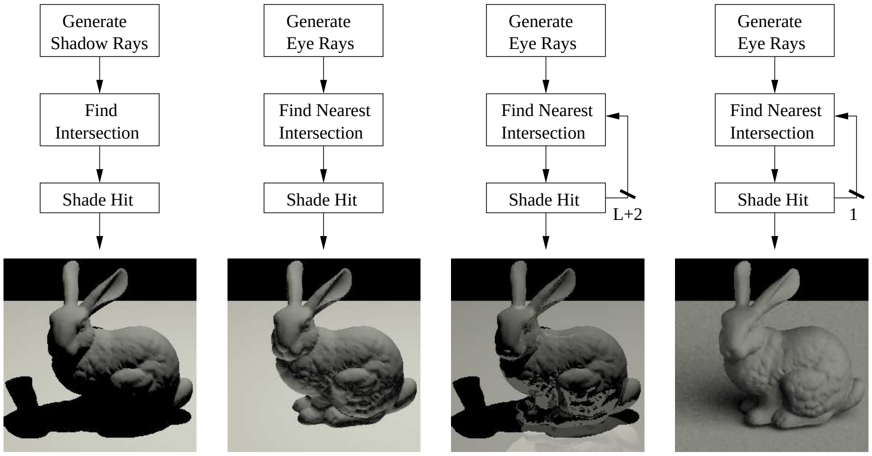
Figure 4: The grid and triangle data structures stored in texture memory. Each grid cell contains a pointer to a list of triangles. If this pointer is null, then no triangles are stored in that voxel. Grids are stored as 3D textures. Triangle lists are stored in another texture. Voxels containing triangles point to the beginning of a triangle list in the triangle list texture. The triangle list consists of a set of pointers to vertex data. The end of the triangle list is indicated by a null pointer. Finally, vertex positions are stored in textures.

3.1.3 Intersector

The triangle intersection stage takes a stream of ray-voxel pairs and outputs ray-triangle hits. It does this by performing ray-triangle intersection tests with all the triangles within a voxel. If a hit occurs, a ray-triangle pair is passed to the shading stage. The code for computing a single ray-triangle intersection is shown in figure 5. The code is similar to that used by Carr et al. [2002] for their DirectX 8 PS 1.4 ray-triangle intersector. We discuss their system further in section 5.

Because triangles can overlap multiple grid cells, it is possible for an intersection point to lie outside the current voxel. The intersection kernel checks for this case and treats it as a miss. Note that rejecting intersections in this way may cause a ray to be tested against the same triangle multiple times (in different voxels). It is possible to use a mailbox algorithm to prevent these extra intersection calculations [Amanatides and Woo 1987], but mailboxing is difficult to implement when multiple rays are traced in parallel.

The layout of the grid and triangles in texture memory is shown in figure 4. As mentioned above, each voxel contains an offset into a triangle-list texture. The triangle-list texture contains a delimited list of offsets into triangle-vertex textures. Note that the triangle-list texture and the triangle-vertex textures are 1D textures. In fact, these textures are being used as a random-access read-only memory. We represent integer offsets as 1-component floating point textures and vertex positions in three floating point RGB textures. Thus, theoretically, four billion triangles could be addressed in texture memory with 32-bit integer addressing. However, much less texture memory is actually available on current graphics cards. Limitations on the size of 1D textures can be overcome by using 2D textures



Shadow Caster (a) Ray Caster (b) Whitted Ray Tracer (c) Path Tracer (d)

Figure 3: Data flow diagrams for the ray tracing algorithms we implement. The algorithms depicted are (a) shadow casting, (b) ray casting, (c) Whitted ray tracing, and (d) path tracing. For ray tracing, each ray-surface intersection generates $L+2$ rays, where L is the number of lights in a scene, corresponding to the number of shadow rays to be tested, and the other two are reflection and refraction rays. Path tracing randomly chooses one ray bounce to follow and the feedback path is only one ray wide.

```
float4 IntersectTriangle( float3 ro, float3 rd, int list_pos, float4 h ){
    float tri_id = texture( list_pos, trilist );
    float3 v0 = texture( tri_id, v0 );
    float3 v1 = texture( tri_id, v1 );
    float3 v2 = texture( tri_id, v2 );
    float3 edge1 = v1 - v0;
    float3 edge2 = v2 - v0;
    float3 pvec = Cross( rd, edge2 );
    float det = Dot( edge1, pvec );
    float inv_det = 1/det;
    float3 tvec = ro - v0;
    float u = Dot( tvec, pvec ) * inv_det;
    float3 qvec = Cross( tvec, edge1 );
    float v = Dot( rd, qvec ) * inv_det;
    float t = Dot( edge2, qvec ) * inv_det;
    bool validhit = select( u >= 0.0f, true, false );
    validhit = select( v >= 0, validhit, false );
    validhit = select( u+v <= 1, validhit, false );
    validhit = select( t < h[0], validhit, false );
    validhit = select( t >= 0, validhit, false );
    t = select( validhit, t, h[0] );
    u = select( validhit, u, h[1] );
    v = select( validhit, v, h[2] );
    float id = select( validhit, tri_id, h[3] );
    return float4( {t, u, v, id} );
}
```

Figure 5: Code for ray-triangle intersection.

with the proper address translation, as well as segmenting the data across multiple textures.

As with the traversal stage, the inner loop over all the triangles in a voxel involves multiple passes. Each ray requires a single pass per ray-triangle intersection.

3.1.4 Shader

The shading kernel evaluates the color contribution of a given ray at the hit point. The shading calculations are exactly like those in the standard graphics pipeline. Shading data is stored in memory much like triangle data. A set of three RGB textures, with 32-bits per channel, contains the vertex normals and vertex colors for each triangle. The hit information that is passed to the shader includes the triangle number. We access the shading information by a simple dependent texture lookup for the particular triangle specified.

By choosing different shading rays, we can implement several flavors of ray tracing using our streaming algorithm. We will look at ray casting, Whitted-style ray tracing, path tracing, and shadow casting. Figure 3 shows a simplified flow diagram for each of the methods discussed, along with an example image produced by our system.

The shading kernel optionally generates shadow, reflection, refraction, or randomly generated rays. These secondary rays are placed in texture locations for future rendering passes. Each ray is also assigned a weight, so that when it is finally terminated, its contribution to the final image may be simply added into the image [Kajiya 1986]. This technique of assigning a weight to a ray eliminates recursion and simplifies the control flow.

Ray Caster. A ray caster generates images that are identical to those generated by the standard graphics pipeline. For each pixel on the screen, an eye ray is generated. This ray is fired into the scene and returns the color of the nearest triangle it hits. No secondary rays are generated, including no shadow rays. Most previous efforts to implement interactive ray tracing have focused on this type of computation, and it will serve as our basic implementation.

Whitted Ray Tracer. The classic Whitted-style ray tracer [Whitted 1980] generates eye rays and sends them out into the scene. Upon finding a hit, the reflection model for that surface is evaluated, and then a pair of reflection and refraction rays, and a set of shadow rays – one per light source – are generated and sent out into the scene.

Path Tracer. In path tracing, rays are randomly scattered from surfaces until they hit a light source. Our path tracer emulates the Arnold renderer [Fajardo 2001]. One path is generated per sample and each path contains 2 bounces.

Shadow Caster. We simulate a hybrid system that uses the standard graphics pipeline to perform hidden surface calculation in the first pass, and then uses ray tracing algorithm to evaluate shadows. Shadow casting is useful as a replacement for shadow maps and shadow volumes. Shadow volumes can be extremely expensive to compute, while for shadow maps, it tends to be difficult to set the proper resolution. A shadow caster can be viewed as a deferred shading pass [Molnar et al. 1992]. The shadow caster pass generates shadow rays for each light source and adds that light's contribution to the final image only if no blockers are found.

Kernel	Multipass						Branching			
	Instr.	Memory Words			Stencil		Instr.	Memory Words		
	Count	R	W	M	RS	WS	Count	R	W	M
Generate Eye Ray	28	0	5	0	0	1	26	0	4	0
Traverse										
Setup	38	11	12	0	1	0	22	7	0	0
Step	20	14	9	1	1	1	12	0	0	1
Intersect	41	14	5	10	1	1	36	0	0	10
Shade										
Color	36	10	3	21	1	0	25	0	3	21
Shadow	16	11	8	0	1	1	10	0	0	0
Reflected	26	11	9	9	1	1	12	0	0	0
Path	17	14	9	9	1	1	11	3	0	0

Table 1: Ray tracing kernel summary. We show the number of instructions required to implement each of our kernels, along with the number of 32-bit words of memory that must be read and written between rendering passes (R, W) and the number of memory words read from random-access textures (M). Two sets of statistics are shown, one for the multipass architecture and another for the branching architecture. For the multipass architecture, we also show the number of 8-bit stencil reads (RS) and writes (WS) for each kernel. Stencil read overhead is charged for all rays, whether the kernel is executed or not.

3.2 Implementation

To evaluate the computation and bandwidth requirements of our streaming ray tracer, we implemented each kernel as an assembly language fragment program. The NVIDIA vertex program instruction set is used for fragment programs, with the addition of a few instructions as described previously. The assembly language implementation provides estimates for the number of instructions required for each kernel invocation. We also calculate the bandwidth required by each kernel; we break down the bandwidth as stream input bandwidth, stream output bandwidth, and memory (random-access read) bandwidth.

Table 1 summarizes the computation and bandwidth required for each kernel in the ray tracer, for both the multipass and the branching architectures. For the traversal and intersection kernels that involve looping, the counts for the setup and the loop body are shown separately. The branching architecture allows us to combine individual kernels together; as a result the branching kernels are slightly smaller since some initialization and termination instructions are removed. The branching architecture permits all kernels to be run together within a single rendering pass.

Using table 1, we can compute the total compute and bandwidth costs for the scene.

$$C = R * (C_r + vC_v + tC_t + sC_s) + R * P * C_{stencil}$$

Here R is the total number of rays traced. C_r is the cost to generate a ray; C_v is the cost to walk a ray through a voxel; C_t is the cost of performing a ray-triangle intersection; and C_s is the cost of shading. P is the total number of rendering passes, and $C_{stencil}$ is the cost of reading the stencil buffer. The total cost associated with each stage is determined by the number of times that kernel is invoked. This number depends on scene statistics: v is the average number of voxels pierced by a ray; t is the average number of triangles intersected by a ray; and s is the average number of shading calculations per ray. The branching architecture has no stencil buffer checks, so $C_{stencil}$ is zero. The multipass architecture must pay the stencil read cost for all rays over all rendering passes. The cost of our ray tracer on various scenes will be presented in the results section.

Finally, we present an optimization to minimize the total number of passes motivated in part by Delany's implementation of a ray tracer for the Connection Machine [Delany 1988]. The traversal and intersection kernels both involve loops. There are various strategies for nesting these loops. The simplest algorithm would be to step through voxels until any ray encounters a voxel containing triangles, and then intersect that ray with those triangles. However, this strategy would be very inefficient, since during intersection only a few rays will have encountered voxels with triangles. On a SIMD machine like the Connection Machine, this results in very low processor utilization. For graphics hardware, this yields an excessive number of passes resulting in large number of stencil read operations dominating the performance. The following is a more efficient algorithm:

```
generate eye ray
while (any(active(ray))) {
    if (oracle(ray))
        traverse(ray)
    else
        intersect(ray)
}
shade(ray)
```

After eye ray generation, the ray tracer enters a while loop which tests whether any rays are *active*. Active rays require either further traversals or intersections; inactive rays have either hit triangles or traversed the entire grid. Before each pass, an oracle is called. The oracle chooses whether to run a traverse or an intersect pass. Various oracles are possible. The simple algorithm above runs an intersect pass if *any* rays require intersection tests. A better oracle, first proposed by Delany, is to choose the pass which will perform the most work. This can be done by calculating the percentage of rays requiring intersection vs. traversal. In our experiments, we found that performing intersections once 20% of the rays require intersection tests produced the minimal number of passes, and is within a factor of two to three of optimal for a SIMD algorithm performing a single computation per rendering pass.

To implement this oracle, we assume graphics hardware maintains a small set of counters over the stencil buffer, which contains the state of each ray. A total of eight counters (one per stencil bit) would be more than sufficient for our needs since we only have four states. Alternatively, we could use the OpenGL histogram operation for the oracle if this operation were to be implemented with high performance for the stencil buffer.

4 Results

4.1 Methodology

We have implemented functional simulators of our streaming ray tracer for both the multipass and branching architectures. These simulators are high level simulations of the architectures, written in the C++ programming language. These simulators compute images and gather scene statistics. Example statistics include the average number of traversal steps taken per ray, or the average number of

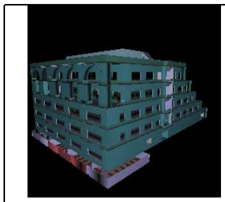
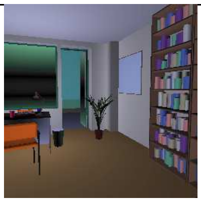
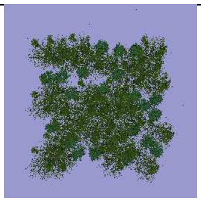
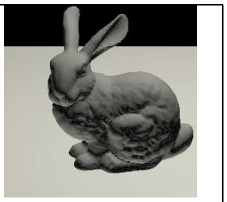
														
Soda Hall Outside			Soda Hall Inside			Forest Top Down			Forest Inside			Bunny Ray Cast		
v	*t*	*s*	*v*	*t*	*s*	*v*	*t*	*s*	*v*	*t*	*s*	*v*	*t*	*s*
14.41	2.52	0.44	26.11	40.46	1.00	81.29	34.07	0.96	130.7	47.90	0.97	93.93	13.88	0.82

Figure 6: Fundamental scene statistics for our test scenes. The statistics shown match the cost model formula presented in section 3.2. Recall that *v* is the average number of voxels pierced by a ray; *t* is the average number of triangles intersected by a ray; and *s* is the average number of shading calculations per ray. Soda hall has 1.5M triangles, the forest has 1.0M triangles, and the Stanford bunny has 70K triangles. All scenes are rendered at 1024x1024 pixels.

ray-triangle intersection tests performed per ray. The multipass architecture simulator also tracks the number and type of rendering passes performed, as well as stencil buffer activity. These statistics allow us to compute the cost for rendering a scene by using the cost model described in section 3.

Both the multipass and the branching architecture simulators generate a trace file of the memory reference stream for processing by our texture cache simulator. In our cache simulations we used a 64KB direct-mapped texture cache with a 48-byte line size. This line size holds four floating point RGB texels, or three floating point RGBA texels with no wasted space. The execution order of fragment programs effects the caching behavior. We execute kernels as though there were a single pixel wide graphics pipeline. It is likely that a GPU implementation will include multiple parallel fragment pipelines executing concurrently, and thus their accesses will be interleaved. Our architectures are not specified at that level of detail, and we are therefore not able to take such effects into account in our cache simulator.

We analyze the performance of our ray tracer on five viewpoints from three different scenes, shown in figure 6.

- Soda Hall is a relatively complex model that has been used to evaluate other real-time ray tracing systems [Wald et al. 2001b]. It has walls made of large polygons and furnishings made from very small polygons. This scene has high depth complexity.

- The forest scene includes trees with millions of tiny triangles. This scene has moderate depth complexity, and it is difficult to perform occlusion culling. We analyze the cache behavior of shadow and reflection rays using this scene.

- The bunny was chosen to demonstrate the extension of our ray tracer to support shadows, reflections, and path tracing.

Figure 7 shows the computation and bandwidth requirements of our test scenes. The computation and bandwidth utilized is broken down by kernel. These graphs clearly show that the computation and bandwidth for both architectures is dominated by grid traversal and triangle intersection.

Choosing an optimal grid resolution for scenes is difficult. A finer grid yields fewer ray-triangle intersection tests, but leads to more traversal steps. A coarser grid reduces the number of traversal steps, but increases the number of ray-triangle intersection tests. We attempt to keep voxels near cubical shape, and specify grid resolution by the minimal grid dimension acceptable along any dimension of the scene bounding box. For the bunny, our minimal grid dimension is 64, yielding a final resolution of 98 × 64 × 163. For the larger Soda Hall and forest models, this minimal dimension is 128, yielding grid resolutions of 250 × 198 × 128 and 581 × 128 × 581 respectively. These resolutions allow our algorithms to spend equal amounts of time in the traversal and intersection kernels.

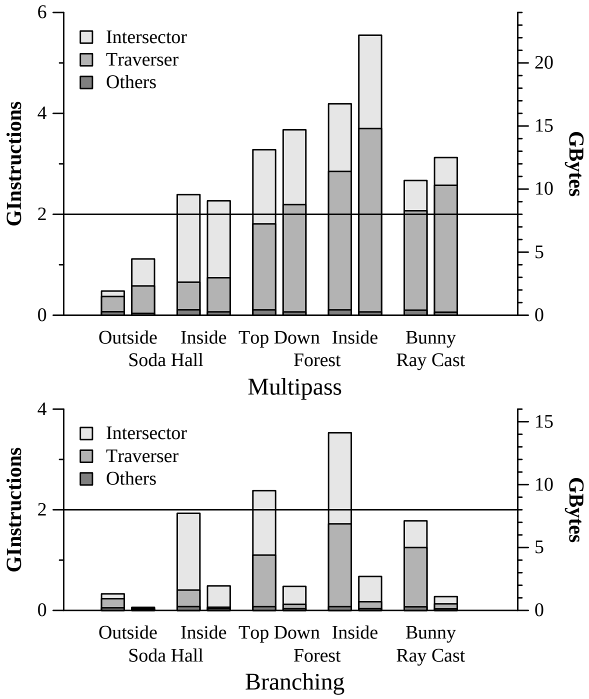


Figure 7: Compute and bandwidth usage for our scenes. Each column shows the contribution from each kernel. Left bar on each plot is compute, right is bandwidth. The horizontal line represents the per-second bandwidth and compute performance of our hypothetical architecture. All scenes were rendered at 1024 × 1024 pixels.

4.2 Architectural Comparisons

We now compare the multipass and branching architectures. First, we investigate the implementation of the ray caster on the multipass architecture. Table 2 shows the total number of rendering passes and the distribution of passes amongst the various kernels. The total number of passes varies between 1000-3000. Although the number of passes seems high, this is the total number needed to render the scene. In the conventional graphics pipeline, many fewer passes per object are used, but many more objects are drawn. In our system, each pass only draws a single rectangle, so the speed of the geometry processing part of the pipeline is not a factor.

We also evaluate the efficiency of the multipass algorithm. Recall that rays may be traversing, intersecting, shading, or done. The efficiency of a pass depends on the percentage of rays processed in that pass. In these scenes, the efficiency is between 6-10% for all of the test scenes except for the outside view of Soda Hall. This

	Pass Breakdown				Per Ray Maximum		SIMD
	Total	Traversal	Intersection	Other	Traversals	Intersections	Efficiency
Soda Hall Outside	2443	692	1747	4	384	1123	0.009
Soda Hall Inside	1198	70	1124	4	60	1039	0.061
Forest Top Down	1999	311	1684	4	137	1435	0.062
Forest Inside	2835	1363	1468	4	898	990	0.068
Bunny Ray Cast	1085	610	471	4	221	328	0.105

Table 2: Breakdown of passes in the multipass system. Intersection and traversal make up the bulk of passes in the systems, with the rest of the passes coming from ray generation, traversal setup, and shading. We also show the maximum number of traversal steps and intersection tests for per ray. Finally, SIMD efficiency measures the average fraction of rays doing useful work for any given pass.

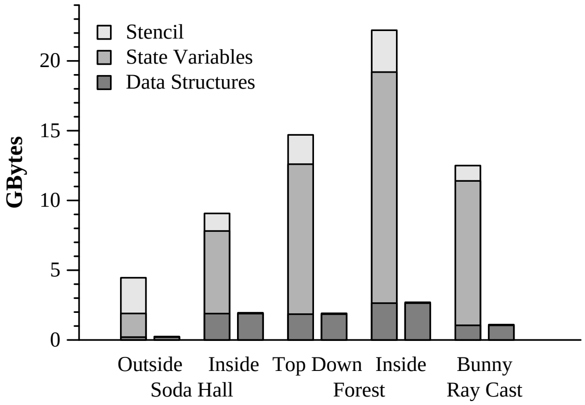


Figure 8: Bandwidth consumption by data type. Left bars are for multipass, right bars for branching. Overhead for reading the 8-bit stencil value is shown on top. State variables are data written to and read from texture between passes. Data structure bandwidth comes from read-only data: triangles, triangle lists, grid cells, and shading data. All scenes were rendered at 1024 × 1024 pixels.

viewpoint contains several rays that miss the scene bounding box entirely. As expected, the resulting efficiency is much lower since these rays never do any useful work during the rest of the computation. Although 10% efficiency may seem low, the fragment processor utilization is much higher because we are using early fragment kill to avoid consuming compute resources and non-stencil bandwidth for the fragment. Finally, table 2 shows the maximum number of traversal steps and intersection tests that are performed per ray. Since the total number of passes depends on the worst case ray, these numbers provide lower bounds on the number of passes needed. Our multipass algorithm interleaves traversal and intersection passes and comes within a factor of two to three of the optimal number of rendering passes. The naive algorithm, which performs an intersection as soon as any ray hits a full voxel, requires at least a factor of five times more passes than optimal on these scenes.

We are now ready to compare the computation and bandwidth requirements of our test scenes on the two architectures. Figure 8 shows the same bandwidth measurements shown in figure 7 broken down by data type instead of by kernel. The graph shows that, as expected, all of the bandwidth required by the branching architecture is for reading voxel and triangle data structures from memory. The multipass architecture, conversely, uses most of its bandwidth for writing and reading intermediate values to and from texture memory between passes. Similarly, saving and restoring these intermediates requires extra instructions. In addition, significant bandwidth is devoted to reading the stencil buffer. This extra computation and bandwidth consumption is the fundamental limitation of the multipass algorithm.

One way to reduce both the number of rendering passes and the bandwidth consumed by intermediate values in the multipass architecture is to unroll the inner loops. We have presented data for a single traversal step or a single intersection test performed per ray in a rendering pass. If we instead unroll our kernels to perform four traversal steps or two intersection tests, all of our test scenes reduce their total bandwidth usage by 50%. If we assume we can suppress triangle and voxel memory references if a ray finishes in the middle of the pass, the total bandwidth reduction reaches 60%. At the same time, the total instruction count required to render each scene increases by less than 10%. With more aggressive loop unrolling the bandwidth savings continue, but the total instruction count increase varies by a factor of two or more between our scenes. These results indicate that loop unrolling can make up for some of the overhead inherent in the multipass architecture, but unrolling does not achieve the compute to bandwidth ratio obtained by the branching architecture.

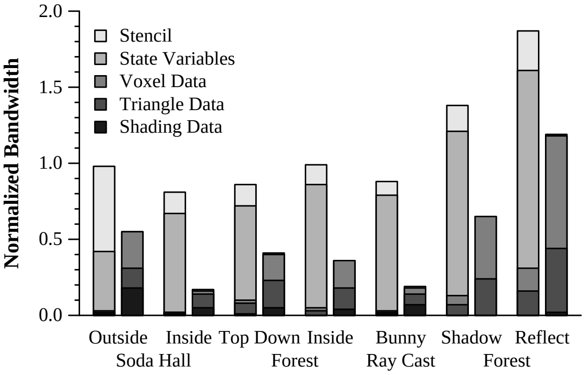


Figure 9: Ratio of bandwidth with a texture cache to bandwidth without a texture cache. Left bars are for multipass, right bars for branching. Within each bar, the bandwidth consumed with a texture cache is broken down by data type. All scenes were rendered at 1024 × 1024 pixels.

Finally, we compare the caching behavior of the two implementations. Figure 9 shows the bandwidth requirements when a texture cache is used. The bandwidth consumption is normalized by dividing by the non-caching bandwidth reported earlier. Inspecting this graph we see that the multipass system does not benefit very much from texture caching. Most of the bandwidth is being used for streaming data, in particular, for either the stencil buffer or for intermediate results. Since this data is unique to each kernel invocation, there is no reuse. In contrast, the branching architecture utilizes the texture cache effectively. Since most of its bandwidth is devoted to reading shared data structures, there is reuse. Studying the caching behavior of triangle data only, we see that a 96-99% hit rate is achieved by both the multipass and the branching system. This high hit rate suggests that triangle data caches well, and that we have a fairly small working set size.

In summary, the implementation of the ray caster on the multipass architecture has achieved a very good balance between computation and bandwidth. The ratio of instruction count to bandwidth matches the capabilities of a modern GPU. For example, the

	Relative	
Extension	Instructions	Bandwidth
Shadow Caster	0.85	1.15
Whitted Ray Tracer	2.62	3.00
Path Tracer	3.24	4.06

Table 3: Number of instructions and amount of bandwidth consumed by the extended algorithms to render the bunny scene using the branching architecture, normalized by the ray casting cost.

NVIDIA GeForce3 is able to execute approximately 2G instructions/s in its fragment processor, and has roughly 8GB/s of memory bandwidth. Expanding the traversal and intersection kernels to perform multiple traversal steps or intersection tests per pass reduces the bandwidth required for the scene at the cost of increasing the computational requirements. The amount of loop unrolling can be changed to match the computation and bandwidth capabilities of the underlying hardware. In comparison, the branching architecture consumes fewer instructions and significantly less bandwidth. As a result, the branching architecture is severely compute-limited based on today's GPU bandwidth and compute rates. However, the branching architecture will become more attractive in the future as the compute to bandwidth ratio on graphics chips increases with the introduction of more parallel fragment pipelines.

4.3 Extended Algorithms

With an efficient ray caster in place, implementing extensions such as shadow casting, full Whitted ray tracing, or path tracing is quite simple. Each method utilizes the same ray-triangle intersection loop we have analyzed with the ray caster, but implements a different shading kernel which generates new rays to be fed back through our system. Figure 3 shows images of the bunny produced by our system for each of the ray casting extensions we simulate. The total cost of rendering a scene depends on both the number of rays traced and the cache performance.

Table 3 shows the number of instructions and bandwidth required to produce each image of the bunny relative to the ray casting cost, all using the branching architecture. The path traced bunny was rendered at 256 × 256 pixels with 64 samples and 2 bounces per pixel while the others were rendered at 1024 × 1024 pixels. The ray cast bunny finds a valid hit for 82% of its pixels and hence 82% of the primary rays generate secondary rays. If all rays were equal, one would expect the shadow caster to consume 82% of the instructions and bandwidth of the ray caster; likewise the path tracer would consume 3.2 times that of the ray caster. Note that the instruction usage is very close to the expected value, but that the bandwidth consumed is more.

Additionally, secondary rays do not cache as well as eye rays, due to their generally incoherent nature. The last two columns of figure 9 illustrate the cache effectiveness on secondary rays, measured separately from primary rays. For these tests, we render the inside forest scene in two different styles. "Shadow" is rendered with three light sources with each hit producing three shadow rays. "Reflect" applies a two bounce reflection and single light source shading model to each primitive in the scene. For the multipass rendering system, the texture cache is unable to reduce the total bandwidth consumed by the system. Once again the streaming data destroys any locality present in the triangle and voxel data. The branching architecture results demonstrate that scenes with secondary rays can benefit from caching. The system achieves a 35% bandwidth reduction for the shadow computation. However caching for the reflective forest does not reduce the required bandwidth. We are currently investigating ways to improve the performance of our system for secondary rays.

5 Discussion

In this section, we discuss limitations of the current system and future work.

5.1 Acceleration Data Structures

A major limitation of our system is that we rely on a preprocessing step to build the grid. Many applications contain dynamic geometry, and to support these applications we need fast incremental updates to the grid. Building acceleration data structures for dynamic scenes is an active area of research [Reinhard et al. 2000]. An interesting possibility would be to use graphics hardware to build the acceleration data structure. The graphics hardware could "scan convert" the geometry into a grid. However, the architectures we have studied in this paper cannot do this efficiently; to do operations like rasterization within the fragment processor they would need the ability to write to arbitrary memory locations. This is a classic scatter operation and would move the hardware even closer to a general stream processor.

In this research we assumed a uniform grid. Uniform grids, however, may fail for scenes containing geometry and empty space at many levels of detail. Since we view texture memory as random-access memory, hierarchical grids could be added to our system.

Currently graphics boards contain relatively small amounts of memory (in 2001 a typical board contains 64MB). Some of the scenes we have looked at require 200MB - 300MB of texture memory to store the scene. An interesting direction for future work would be to study hierarchical caching of the geometry as is commonly done for textures. The trend towards unified system and graphics memory may ultimately eliminate this problem.

5.2 CPU vs. GPU

Wald et al. have developed an optimized ray tracer for a PC with SIMD floating point extensions [Wald et al. 2001b]. On an 800 MHz Pentium III, they report a ray-triangle intersection rate of 20M intersections/s. Carr et al. [2002] achieve 114M ray-triangle intersections/s on an ATI Radeon 8500 using limited fixed point precision. Assuming our proposed hardware ran at the same speed as a GeForce3 (2G instructions/s), we could compute 56M ray-triangle intersections/s. Our branching architecture is compute limited; if we increase the instruction issue rate by a factor of four (8G instructions/s) then we would still not use all the bandwidth available on a GeForce3 (8GB/s). This would allow us to compute 222M ray-triangle intersections per second. We believe because of the inherently parallel nature of fragment programs, the number of GPU instructions that can be executed per second will increase much faster than the number of CPU SIMD instructions.

Once the basic feasibility of ray tracing on a GPU has been demonstrated, it is interesting to consider modifications to the GPU that support ray tracing more efficiently. Many possibilities immediately suggest themselves. Since rays are streamed through the system, it would be more efficient to store them in a stream buffer than a texture map. This would eliminate the need for a stencil buffer to control conditional execution. Stream buffers are quite similar to F-buffers which have other uses in multipass rendering [Mark and Proudfoot 2001]. Our current implementation of the grid traversal code does not map well to the vertex program instruction set, and is thus quite inefficient. Since grid traversal is so similar to rasterization, it might be possible to modify the rasterizer to walk through the grid. Finally, the vertex program instruction set could be optimized so that ray-triangle intersection could be performed in fewer instructions.

Carr et al. [2002] have independently developed a method of using the GPU to accelerate ray tracing. In their system the GPU is only used to accelerate ray-triangle intersection tests. As in our system, GPU memory is used to hold the state for many active rays. In their system each triangle in turn is fed into the GPU and tested for intersection with all the active rays. Our system differs from theirs in that we store all the scene triangles in a 3D grid on the GPU; theirs stores the acceleration structure on the CPU. We also run the entire ray tracer on the GPU. Our system is much more efficient than theirs since we eliminate the GPU-CPU communication bottleneck.

5.3 Tiled Rendering

In the multipass architecture, the majority of the memory bandwidth was consumed by saving and restoring temporary variables. Since these streaming temporaries are only used once, there is no bandwidth savings due to the cache. Unfortunately, when these streaming variables are accessed as texture, they displace cacheable data structures. The size of the cache we used is not large enough to store the working set if it includes both temporary variables and data structures. The best way to deal with this problem is to separate streaming variables from cacheable variables.

Another solution to this problem is to break the image into small tiles. Each tile is rendered to completion before proceeding to the next tile. Tiling reduces the working set size, and if the tile size is chosen so that the working set fits into the cache, then the streaming variables will not displace the cacheable data structures. We have performed some preliminary experiments along these lines and the results are encouraging.

6 Conclusions

We have shown how viewing a programmable graphics processor as a general parallel computation device can help us leverage the graphics processor performance curve and apply it to more general parallel computations, specifically ray tracing. We have shown that ray casting can be done efficiently in graphics hardware. We hope to encourage graphics hardware to evolve toward a more general programmable stream architecture.

While many believe a fundamentally different architecture would be required for real-time ray tracing in hardware, this work demonstrates that a gradual convergence between ray tracing and the feed-forward hardware pipeline is possible.

7 Acknowledgments

We would like to thank everyone in the Stanford Graphics Lab for contributing ideas to this work. We thank Matt Papakipos from NVIDIA for his thoughts on next generation graphics hardware, and Kurt Akeley and our reviewers for their comments. Katie Tillman stayed late and helped with editing. We would like to thank Hanspeter Pfister and MERL for additional support. This work was sponsored by DARPA (contracts DABT63-95-C-0085 and MDA904-98-C-A933), ATI, NVIDIA, Sony, and Sun.

References

3DLABS, 2001. OpenGL 2.0 whitepapers web site. http://www.3dlabs.com/support/developer/ogl2/index.htm.

ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. 1990. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, 1–6.

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, 3–10.

ANDERSON, B., STEWART, A., MACAULAY, R., AND WHITTED, T. 1997. Accommodating memory latency in a low-cost rasterizer. In *1997 SIGGRAPH / Eurographics Workshop on Graphics hardware*, 97–102.

ATI, 2001. RADEON 8500 product web site. http://www.ati.com/products/pc/radeon8500128/index.html.

CARR, N. A., HALL, J. D., AND HART, J. C. 2002. The ray engine. Tech. Rep. UIUCDCS-R-2002-2269, Department of Computer Science, University of Illinois.

DELANY, H. C. 1988. Ray tracing on a connection machine. In *Proceedings of the 1988 International Conference on Supercomputing*, 659–667.

FAJARDO, M. 2001. Monte carlo ray tracing in action. In *State of the Art in Monte Carlo Ray Tracing for Realistic Image Synthesis - SIGGRAPH 2001 Course 29*. 151–162.

FUJIMOTO, A., TANAKA, T., AND IWATA, K. 1986. ARTS: Accelerated ray tracing system. *IEEE Computer Graphics and Applications 6*, 4, 16–26.

HALL, D., 2001. The AR350: Today's ray trace rendering processor. 2001 SIGGRAPH / Eurographics Workshop On Graphics Hardware - Hot 3D Session 1. http://graphicshardware.org/previous/www_2001/presentations/ Hot3D_Daniel_Hall.pdf.

HAVRAN, V., PRIKRYL, J., AND PURGATHOFER, W. 2000. Statistical comparison of ray-shooting efficiency schemes. Tech. Rep. TR-186-2-00-14, Institute of Computer Graphics, Vienna University of Technology.

IGEHY, H., ELDRIDGE, M., AND PROUDFOOT, K. 1998. Prefetching in a texture cache architecture. In *1998 SIGGRAPH / Eurographics Workshop on Graphics hardware*, 133–ff.

KAJIYA, J. T. 1986. The rendering equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, 143–150.

KHAILANY, B., DALLY, W. J., RIXNER, S., KAPASI, U. J., MATTSON, P., NAMKOONG, J., OWENS, J. D., AND TOWLES, B. 2000. IMAGINE: Signal and image processing using streams. In *Hot Chips 12*. IEEE Computer Society Press.

KIRK, D., 2001. GeForce3 architecture overview. http://developer.nvidia.com/docs/IO/1271/ATT/GF3ArchitectureOverview.ppt.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, 149–158.

MARK, W. R., AND PROUDFOOT, K. 2001. The F-buffer: A rasterization-order FIFO buffer for multi-pass rendering. In *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware*.

MARSHALL, B., 2001. DirectX graphics future. Meltdown 2001 Conference. http://www.microsoft.com/mscorp/corpevents/meltdown2001/ppt/DXG9.ppt.

MICROSOFT, 2001. DirectX product web site. http://www.microsoft.com/directx/.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. PixelFlow: High-speed rendering using image composition. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, 231–240.

NVIDIA, 2001. GeForce3 Ti Family: Product overview. 10.01v1. http://www.nvidia.com/docs/lo/1050/SUPP/gf3ti_overview.pdf.

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. 1998. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, 233–238.

PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *1999 ACM Symposium on Interactive 3D Graphics*, 119–126.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000*, 425–432.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, 299–306.

SPITZER, J., 2001. Texture compositing with register combiners. http://developer.nvidia.com/docs/IO/1382/ATT/RegisterCombiners.pdf.

TORBORG, J., AND KAJIYA, J. T. 1996. Talisman: Commodity realtime 3D graphics for the PC. In *Proceedings of ACM SIGGRAPH 96*, 353–363.

WALD, I., SLUSALLEK, P., AND BENTHIN, C. 2001. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 277–288.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3, 153–164.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6, 343–349.

Graphics Hardware (2002), pp. 1–10
Thomas Ertl, Wolfgang Heidrich, and Michael Doggett (Editors)

The Ray Engine

Nathan A. Carr Jesse D. Hall John C. Hart

University of Illinois

Abstract

Assisted by recent advances in programmable graphics hardware, fast rasterization-based techniques have made significant progress in photorealistic rendering, but still only render a subset of the effects possible with ray tracing. We are closing this gap with the implementation of ray-triangle intersection as a pixel shader on existing hardware. This GPU ray-intersection implementation reconfigures the geometry engine into a ray engine that efficiently intersects caches of rays for a wide variety of host-based rendering tasks, including ray tracing, path tracing, form factor computation, photon mapping, subsurface scattering and general visibility processing.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: *Hardware acceleration, ray caching, ray classification, ray coherence, ray tracing, pixel shaders.*

1. Introduction

Hardware-accelerated rasterization has made great strides in simulating global illumination effects, such as shadows[35, 25, 7], reflection[3], multiple-bounce reflection[5], refraction[9], caustics[29] and even radiosity[13]. Nonetheless some global illumination effects have eluded rasterization solutions, and may continue to do so indefinitely. The environment map provides polygon rasterization with limited global illumination capabilities by approximating the irradiance of all points on an object surface with the irradiance at a single point[3]. This single-point irradiance approximation can result in some visually obvious errors, such as the boat in wavy water shown in Figure 1.

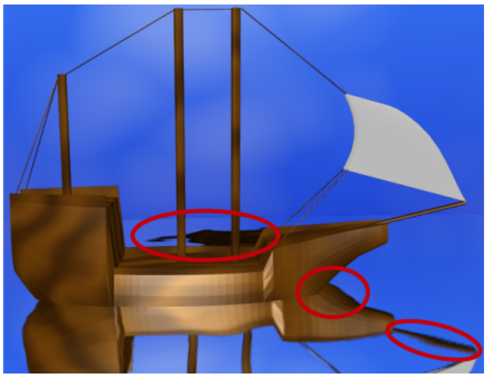


Figure 1: *What is wrong with this environment-mapped picture? (1) The boat does not meet its reflection, (2) the boat is reflected in the water behind it, and (3) some aliasing can be seen in the reflection.*

Ray tracing of course simulates all of these effects and more. It can provide true reflection and refraction, complete with local and multiple bounces. Complex camera models with compound lenses are easier to simulate using ray tracing[15]. Numerous global illumination methods are based on ray tracing including path tracing[12], Monte-Carlo ray tracing[33] and photon mapping[10].

Ray tracing is classically one of the most time consuming operations on the CPU, and the graphics community has been eager to accelerate it using whatever methods possible. Hardware-based accelerations have included CPU-specific tuning, distribution across parallel processors and even construction of special purpose hardware, as reviewed in Section 2.

Graphics cards have recently included support for programmable shading in an effort to increase the realism of their rasterization-based renderers[16]. This added flexibility is transforming the already fast graphics processing unit (GPU) into a supercomputing coprocessor, and its power is being applied to a wider variety of applications than its developers originally intended.

One such application is ray tracing. Section 3 shows how to configure the graphics processing unit (GPU) to compute ray-triangle intersections, and Section 4 details an implementation. This GPU ray-triangle intersection reconfigures the graphics accelerator into a *ray engine*, described in Section 5, that hides the details of its back-end GPU ray-triangle intersection, allowing the ray engine to be more easily integrated into existing rendering software systems.

The ray engine can make existing rasterization-based renderers look better. A rasterization renderer augmented with the ray engine could trace the rays necessary to achieve effects currently impossible with rasterization-only rendering, including local reflections (Figure 1), true refractions and sub-surface scattering[11].

The ray engine is also designed to be efficiently integrated into existing ray-tracing applications. The ray engine performs best when intersecting caches of coherent rays[21] from host-based rendering tasks. This is a form of load balancing that allows the GPU to do what it does best (perform the same computation on arrays of data), and lets the CPU do what the GPU does worst (reorganize the data into efficient structures whose processing requires flow control). A simple ray tracing system we built using the ray engine is already running at speeds comparable to the fastest known ray tracer, which was carefully tuned to a specific CPU[32]. The ray engine could likewise accelerate Monte Carlo ray tracing, photon mapping, form factor computation and visibility preprocessing.

2. Previous Work

Although classic ray tracing systems support a wide variety of geometric primitives, some recent ray tracers designed to achieve interactive rates (including ours) have limited themselves to triangles. This has not been a severe limitation as geometric models can be tessellated, and the simplicity of the ray-triangle intersection has led to highly efficient implementations[2, 18].

Hardware z-buffered rasterization can quickly determine the visibility of triangles. One early hardware optimization for ray tracing was the first-hit speedup, which replaced eye-ray intersections with a z-buffered rasterization of the scene using object ID as the color[34]. Eye rays are a special case of a coherent bundle of rays. Such rays can likewise be efficiently intersected through z-buffered rasterization for hardware accelerated global illumination[28], of which ray tracing is a subset.

One obvious hardware acceleration of ray tracing is to optimize its implementation for a specific CPU. The current fastest CPU implementation we are aware of is a coherent ray tracer tuned for the Intel Pentium III processor[32]. This ray tracer capitalized on a variety of spatial, ray and memory coherencies to best utilize CPU optimizations such as caching, branch prediction, instruction reordering, speculative execution and SSE instructions. Their implementation ran at an average of 30 million intersections per second on an 800 Mhz Pentium III. They were able to trace between 200K and 1.5M rays per second, which was over ten times faster than POV-Ray and Rayshade.

There have been a large number of implementations of ray tracers on MIMD computers[26]. These implementations focus on issues of load balancing and memory utilization. One recent implementation on 60 processors of an SGI Origin 2000 was able to render at 512^2 resolution scenes of from 20 to 2K patches at rates ranging from two to 20 Hz[19].

Special purpose hardware has also been designed for ray tracing. The AR350 is a production graphics accelerator designed for the off-line (non-real-time) rendering of scenes with sophisticated Renderman shaders[8]. A ray tracing system designed around multiprocessors with smart memory is also in progress[23].

Our ray engine is similar in spirit to another GPU-based ray tracing implementation that simulates a state machine[24]. This state-based approach breaks ray tracing down into several states, including grid traversal, ray-triangle intersection and shading. This approach performs the entire ray tracing algorithm on the GPU, avoiding the slow readback process required for GPU-CPU communication that our approach must deal with. The state-based method however is not particularly efficient on present and near-future GPU's due to the lack of control flow in the fragment program, resulting in a large portion of pixels (from 90% to 99%) remaining idle if they are in a different state than the one currently being executed. Our approach has been designed to organize ray tracing to achieve full utilization of the GPU.

3. Ray Tracing with the GPU

3.1. Ray Casting

The core of any ray tracer is the intersection of rays with geometry. Rays are represented parametrically as $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ where $\mathbf{o}$ is the ray origin, $\mathbf{d}$ is the ray direction and $t \geq 0$ is a real parameter corresponding to points along the ray. The classic implementation of recursive ray tracing casts each ray individually and intersects it against the scene geometry. This process generates a list of parameters t_i corresponding to points of intersection with the scene's geometric primitives. The least positive element of this list is returned as the first intersection, the one closest to the ray origin.

Figure 2(a) illustrates ray casting as a crossbar. This illustration represents the rays with horizontal lines and the (unorganized) geometric primitives (e.g. triangles) with vertical lines. The crossing points of the horizontal and vertical lines represent intersection tests between rays and triangles.

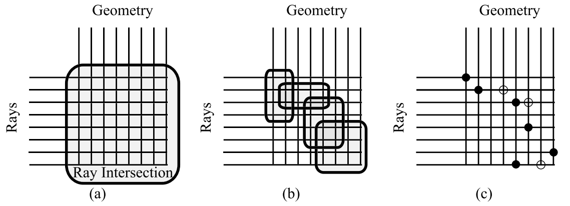


Figure 2: *Ray intersection is a crossbar.*

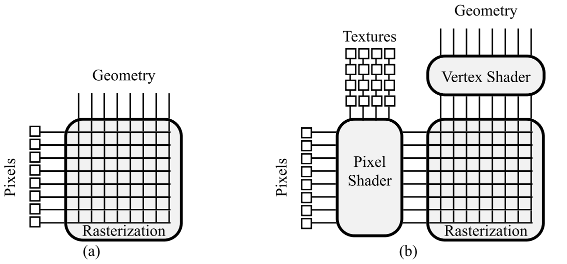


Figure 3: *Programmable pixel shading is a crossbar.*

This crossbar represents an all-pairs check of every ray against every triangle. Since their inception, ray tracers have avoided checking every triangle against every primitive through the use of spatial coherent data structures on both the rays and the geometry. These data structures reorganize the crossbar into a sparse overlapping block structure, as shown in Figure 2(b). Nevertheless the individual blocks are themselves full crossbars that perform an all pairs comparison on their subset of the rays and geometry.

The result of ray casting is the identification of the geometry (if any) intersected first by each ray. This result is a series of points in the crossbar, no greater than one per horizontal line (ray). These first intersections are shown as black disks in Figure 2(c). The other ray-triangle intersections are indicated with open circles and are ignored in simple ray casting.

3.2. Programmable Shading Hardware

Graphics accelerators have been designed to implement a pipeline that converts polygons vertices from model coordinates to viewport coordinates. Once in viewport coordinates, rasterization fills the polygon with pixels, interpolating the depth, color and texture coordinates in a perspective-correct fashion. During rasterization, interpolated texture coordinates index into texture memory to map an image texture onto the polygon.

This rasterization process can also be viewed as a crossbar, as shown in Figure 3(a). The vertical lines represent individual polygons passing through the graphics pipeline whereas the horizontal lines represent the screen pixels.

Consider the case where each polygon, a quadrilateral, exactly covers all of the screen pixels. Then rasterization of these polygons performs an all-pairs combination of every pixel with every polygon.

While even early graphics accelerators were programmable through firmware[4], modern graphics accelerators contain user-programmable elements designed specifically for advanced shading[16]. These programmable elements can be separated into two components, the vertex shader and the pixel shader, as shown in Figure 3(b). The vertex shader is a user-programmable stream processor that can alter the attributes (but not the number) of vertices sent to the rasterizer. The pixel shader can perform arithmetic operations on multiple texture coordinates and fetched texture samples, but does so in isolation and cannot access data stored at any other pixel. Pixel shaders run about an order of magnitude faster than vertex shaders.

3.3. Mapping Ray Casting to Programmable Shading Hardware

We map the ray casting crossbar in Figure 2 to the rasterization crossbar in Figure 3 by distributing the rays across the pixels and broadcasting a stream of triangles to each pixel by sending their coordinates down the geometry pipeline as the vertex attribute data (e.g. color, texture coordinates) of screen filling quadrilaterals.

The rays are stored in two screen-resolution textures. The color of each pixel of the ray-origins texture stores the coordinates of the origin of the ray. The color of each pixel of the ray-directions texture stores the coordinates of the ray direction vector.

An identical copy of the triangle data is stored at each vertex of a screen-filling quadrilateral. Rasterization of this quadrilateral interpolates these attributes at each pixel of its screen projection. Since the attributes are identical at all four vertices, interpolation simply distributes a copy of the triangle data to each pixel.

A pixel shader performs the ray-triangle intersection computation by merging the ray data stored per-pixel in the texture maps with the triangle data distributed per-pixel by the interpolation of the attribute data stored at the vertices of the quadrilateral. The specifics of this implementation will be described further in Section 4.

3.4. Discussion

The decision to store rays in texture and triangles as vertex attributes was based initially on precision. Since rays can be specified with five real values whereas triangles require nine we found it easier and more accurate to store the ray values at the lower texture precisions.

We also chose to implement ray-triangle intersection as a pixel shader instead of a vertex shader. Vertex shaders do not have direct access to the rasterization crossbar, and hence needed to store ray data as constants in the vertex shader's local memory. The vertex shader is also slower, and was able to compute 4.1M ray-triangle intersections per second, which is much less than what the CPU is currently capable of performing.

Viewing the GPU as a SIMD processor[20] allowed us to compare other SIMD ray tracing implementations. SIMD ray tracers typically distribute rays to the processors and broadcast the geometry, or distribute geometry and broadcast the rays. The AR350 ray tracing hardware utilized a fine-grain ray distribution to isolated processors[8], which improved load balancing, but inhibited the possible advantages of ray coherence. The coherent ray tracer[32] also distributed rays at its lowest level, intersecting each triangle with four coherent rays using SSE whereas an axis-aligned BSP-tree coherently organized the triangles (but required special implemention to efficiently intersect four-ray bundles). Geometry distribution on the other hand seems better suited for handling the special problems due to ray tracing large scene databases[31].

4. Ray-Triangle Intersection on the GPU

The pixel shader implementation of ray-triangle intersection treats the GPU as a SIMD parallel processor[20]. In this model, the framebuffer is treated as an accumulator data array of 5-vectors (r, g, b, α, z), and texture maps are used as data arrays for input and variables. Pixel shaders perform sequences of operations that combine the textures and the framebuffer. While compilers exist for multipass programming[20, 22], the current limitations of pixel shaders required complete knowledge and control of the available instructions and registers to implement ray intersection.

4.1. Input

Ray Data. As mentioned in Section 3.3, the GPU component of the ray engine intersects multiple rays with a single triangle. Every pixel in the data array corresponds to an individual ray. Our implementation stores ray data in two textures: a ray-origins texture and a ray-directions texture. Batches of rays cast from the eyepoint or a point light source will have a constant color ray-origins texture and their texture map could be stored as a single pixel or a pixel shader constant.

Triangle Data. The triangle data is encapsulated in the attributes of the four vertices of a screen filling quad. Let $\mathbf{a}, \mathbf{b}, \mathbf{c}$ denote the three vertices of the triangle, and $\mathbf{n}$ denote the triangles front facing normal. The triangle id was stored as the quad's color, and the vectors $\mathbf{a}, \mathbf{b}, \mathbf{n}, \mathbf{ab}(= \mathbf{b} - \mathbf{a}), \mathbf{ac}, \mathbf{bc}$ were mapped to multi-texture coordinate vectors. The redundant vector information includes ray-independent precomputation that reduces the size and workload of the pixel shader. Our implementation passes only the three vertices of the triangle from the host, and computes the additional redundant values in the vertex shader.

The texture coordinates for texture zero (s_0, t_0) are special and are not constant across the quadrilateral. They are instead set to $(0,0), (1,0), (1,1), (0,1)$ at the four vertices, and rasterization interpolates these values linearly across the quad's pixels. These texture coordinates are required by the pixel shader to access each pixel's corresponding ray in the screen-sized ray-origins and ray-directions textures.

4.2. Output

The output of the ray-triangle intersection needs to be queried by the CPU, which can be an expensive operation due to the asymmetric AGP bus on personal computers (which sends data to the graphics card much faster than it can receive it). The following output format is designed to return as little data as necessary, limiting itself to the index of the triangle that intersects the ray closest to its origin, using the z-buffer to manage the ray parameter t of the intersection.

Color. The color channel contains the color of the first triangle the ray intersects (if any). For typical ray tracing applications, this color will be a unique triangle id. These triangle id's can index into an appearance model for the subsequent shading of the ray-intersection results.

Alpha. Our pixel shader intersection routine conditionally sets the fragments alpha value to indicate ray intersection. The alpha channel can then be used as a mask by other applications if the rays are coherent (e.g. like eye rays through the pixels in the frame buffer).

The t-Buffer. The t-value of each intersection is computed and replaces the pixel's z-value. The built-in z-test is used so the new t-value will overwrite the existing t-value stored in the z-buffer if the new value is smaller. This allows the z-buffer to maintain the least positive solution t for each ray. Since the returned t value is always non-negative, the t-value maintained by the z-buffer always corresponds to the first triangle the ray intersects.

4.3. Intersection

We examined a number of efficient ray-triangle intersection tests[6, 2, 18], and managed to reorganize one[18] to fit in a pixel shader.

Our revised ray-triangle intersection is evaluated as

$$\mathbf{ao} = \mathbf{o} - \mathbf{a}, \tag{1}$$

$$\mathbf{bo} = \mathbf{o} - \mathbf{b}, \tag{2}$$

$$t = -\frac{\mathbf{n} \cdot \mathbf{ao}}{\mathbf{n} \cdot \mathbf{d}}, \tag{3}$$

$$\mathbf{aod} = \mathbf{ao} \times \mathbf{d}, \tag{4}$$

$$\mathbf{bod} = \mathbf{bo} \times \mathbf{d}, \tag{5}$$

$$u = \mathbf{ac} \cdot \mathbf{aod}, \tag{6}$$

$$v = -\mathbf{ab} \cdot \mathbf{aod}, \tag{7}$$

$$w = \mathbf{bc} \cdot \mathbf{bod}. \tag{8}$$

The intersection passes only if all three (unnormalized) barycentric coordinates u, v and w are non-negative. If the ray does not intersect the triangles, the alpha channel for that

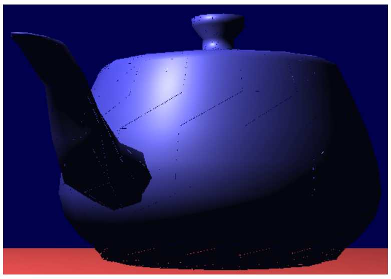


Figure 4: *Leaky teapot, due to the low precision implementation on PS1.4 pixel shaders used to test the performance of ray-triangle intersection. Our simulations using the precision available on upcoming hardware are indistinguishable from software renderings.*

pixel is set to zero and the pixel is killed. The parameter t is also tested against the current value in the z-buffer, and if it fails the pixel is also killed. Surviving pixels are written to the framebuffer as the ray intersection currently closest to the ray origin.

This implementation reduces cross products, which require multiple pixel shader operations to compute. The quotient (3) was implemented using the *texdepth* instruction, which implements the "depth replace" texture shader.

4.4. Results

We tested the PS1.4 implementation of the ray-triangle intersection using the ATI R200 chipset on the Radeon 8500 graphics card. The limited numerical precision of its pixel shader (16-bit fixed point, with a range of ±8) led to some image artifacts shown in Figure 4, this implementation did suffice to measure the speed of an actual hardware pixel shader on the task of ray intersection.

We clocked our GPU implementation of ray intersection at 114M intersections per second. The fastest CPU-based ray tracer was able to compute between 20M and 40M intersections per second on an 800Mhz Pentium III[32]. Even doubling the CPU numbers to estimate performance on today's computers, our GPU ray-triangle intersection performance already exceeds that of the CPU, and we expect the gap to widen as GPU performance growth continues to outpace CPU performance growth.

5. Ray Engine Organization

This section outlines the encapsulation of the GPU ray-intersection into a ray engine. It begins with a discussion of why the CPU is a better choice for the management of rays during the rendering process. Since the CPU is managing the rays, the ray engine is packaged to provide easy access to the GPU ray-intersection acceleration through a front-end interface. This interface accepts rays in coherent bundles, which can be efficiently traced by the GPU ray-intersection implementation.

5.1. The Role of the CPU

We structured the ray engine to perform ray intersection on the GPU and let the host organize the casting of rays and manage the resulting radiance samples. Since the bulk of the computational resources used by a ray tracer are spent on ray intersection, the management of rays and their results is a relatively small overhead for the CPU, certainly smaller than performing the entire ray tracing on the CPU.

The pixel shader on the GPU is a streaming SIMD processor good at running the same algorithm on all elements of a data array. The CPU is a fast scalar processor that is better at organizing and querying more sophisticated data structures, and is capable of more sophisticated algorithmic tools such as recursion. Others have implemented the entire ray tracer on the GPU[24], but such implementations can be cumbersome and inefficient.

For example, recursive ray tracing uses a stack. While some have proposed the addition of state stacks in programmable shader hardware[17], such hardware is not currently available. Recursive ray tracing can be implemented completely on the GPU[24], but apparently at the expense of generating two frame buffers full of reflection and refraction rays at each intersection, which are then managed by the host.

The need for a stack can be avoided by path tracing[12]. Paths originating from the eyepoint passing through a pixel can accumulate its intermediate results at the same location in texture maps. Path tracing requires importance sampling to be efficient, even with fast ray intersection. Sophisticated importance sampling methods[30] use global queries into the scene database, as well as queries into previous radiance results in the scene. Such queries are still performed more efficiently on the CPU than on the GPU.

Some ray tracers also organize rays and geometry into coherent caches that are cast in an arbitrary order to more efficiently render large scenes[21]. The management of ray caches and the radiances resulting

rom their batched tracing requires a lot of data shuffling. An implementation on the GPU would require all of the pixels in the image returned by the batch ray intersection algorithm to be shuffled to contribute to the radiance of the previously cast rays. While dependent texturing can be used to perform this shuffling[24], the GPU is ill-designed to organize and set up this mapping.

We used the NV_FENCE extension to overlap the computation of the CPU and GPU. This allows the CPU to test whether the GPU is busy working on a ray-triangle bundle so the CPU can continue to work simultaneously on ray caching.

5.2. The Ray Engine Interface

Organizing high-performance rendering services to be transparent makes them easier to integrate into existing rendering systems[14]. We structured the ray engine as both a front end driver that runs on the host and interfaces with the application, and a back end component that runs on the GPU to perform ray intersections.

The front end of the ray engine accepts a cache of rays from a host application. This front end converts the ray cache into the texture map data for the pixel shader to use for intersection. The front end then sends the geometry (from a shared database with the application) down the geometry pipeline to the pixel shader. The pixel shader is treated as a back end of this system that intersects the rays with the triangles passed to it. The front end grabs the results of ray intersection (triangle id, t-value and, if supported, the barycentric coordinates) and returns them to the application in a more appropriate format.

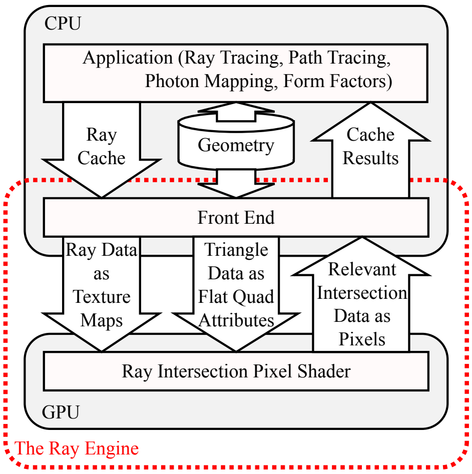

CPU
Application (Ray Tracing, Path Tracing, Photon Mapping, Form Factors)
Ray Cache
Geometry
Cache Results
Front End
Ray Data as Texture Maps
Triangle Data as Flat Quad Attributes
Relevant Intersection Data as Pixels
Ray Intersection Pixel Shader
GPU
The Ray Engine

Figure 5: *The organization of the ray engine.*

The main drawback of implementing ray casting applications on the host is the slow readback bandwidth of the AGP bus when transferring data from the GPU back to the CPU. This bottleneck is addressed by the ray engine system with compact data that is returned infrequenty (once after all triangles have been sent to the GPU).

5.3. The Ray Cache

Accelerating the implementation of ray intersection is not enough to make ray tracing efficient. The number of ray intersections needs to be reduced as well. The ray engine uses an octree to maintain geometry coherence and a 5-D ray tree[1] to maintain ray coherence.

The ray engine works more efficiently when groups of similar rays intersect a collection of spatially coeherent triangles. In order to maintain full buckets of coherent rays, we utilize a ray cache[21]. Ray caching allows some rays to be cast in arbitrary order such that intersection computations can be performed as batch processes.

As rays are generated, they are added to the cache, which collects them into buckets of rays with coherent origins and directions. For maximum performance on the ray engine, each bucket should contain some optimal hardware-dependent number of rays. Our bucket size was 256 rays, organized as two 64×4 ray-origin and ray-direction textures. Textures on graphics cards are commonly stored in small blocks instead of in scanline order to better capitalize on spatial coherence by placing more relevant texture samples into the texture cache of the GPU. The size of these texture blocks is GPU-dependent and can be found through experimentation.

If adding a ray makes a bucket larger than the optimal bucket size then the node is split into four subnodes along the axis of greatest variance centered at the using the mean values of the ray origins and directions. We also add rays to the cache in random order which helps keep the tree balanced.

When the ray tracer needs a result or the entire ray cache becomes full, a bucket is sent to the ray engine to be intersected with geometry. We send the fullest buckets first to maximize utilization of the ray engine resources. Each node of the tree contains the total number of rays contained in the buckets below it. Our search traverses down the largest valued nodes until a bucket is reached. While this simple greedy search is not guaranteed to find the largest bucket, it is fast and works well in practice since the buckets share the same maximum size. This greedy search also tends to balance the tree.

Once the search has chosen a bucket, rays are stolen from that node's siblings to fill the bucket to avoid wasting intersection computations. Due to the greedy search and the node merging described next, this ensures that buckets sent to the ray engine are always as full as possible, even though in the ray tree they are typically only 50-80% full.

Once a bucket has been removed from the tree and traced, it can often leave neighboring buckets containing only a few rays. Our algorithm walks back up the tree from the removed bucket leaf node, collecting subtrees into a single bucket leaf node if the number of rays in the subtree has fallen below a threshold. Our tests showed that this process typically merged only a single level of the tree.

The CPU performs a ray bucket intersection test[1] against the octree cells to determine which should be sent to the GPU. We also used the vertex shader to cull back-facing triangles as well as triangles outside the ray bucket from intersection consideration. The vertex shader cannot change the number of vertices passing through it, but it can transform the screen-filling quad containing the triangle data to an off-screen location which causes it to be clipped.

5.4. Results

We implemented the ray engine on a simulator for an upcoming GPU based on the expected precision and capabilities needed to support the Direct3D 9 specification. These capabilities allow us to produce full precision images that lack the artifacts shown earlier in Figure 4.

We used the ray engine to classically ray trace a teapot room and an office scene, shown in Figure 6(a) and (c). We applied the ray engine to a Monte-Carlo ray tracer that implemented distributed ray tracing and photon mapping, which resulted in Figure 6(b). The ray engine was also used to ray trace two views of one floor from the Soda Hall dataset, shown in Figures 6(d) and (e).

The performance is shown in Figure 1. Since our implementation is on a non-realtime simulator, we have estimated our performance using the execution rates measured on the GeForce 4. We measured the performance in rays per second, which measures the number of rays intersected with the entire scene per second. This figure includes the expensive traversal of the ray-tree and triangle octree as well as the ray-triangle intersections.

Scene	Polygons	Rays/sec.
Teapot Room Classical	2,650	206,905
Teapot Room Monte-Carlo	2,650	149,233
Office	34,000	114,499
Soda Hall Top View	11,052	129,485
Soda Hall Side View	11,052	131,302

Table 1: *Rays per second across a variety of scenes and applications.*

This perfomance meets the low end performance of the coherent ray tracer, which was able to trace from 200K to 1.5M rays per second[32]. It too used coherent data structures to increase performance, in this case an axis aligned BSP tree organized specifically to be efficiently traversed by the CPU. Our ray traversal implementation is likely not as carefully optimized as theirs.

6. Analysis and Tuning

Suppose we are given a set of R rays and a set of T triangles for performing ray-triangle intersection tests. We denote the time to run the tests on the GPU and CPU respectively as $\mathrm{GPU}(R,T)$ and $\mathrm{CPU}(R,T)$. To acheive improved performance, we are only interested in values of R and T for which $\mathrm{GPU}(R,T) \leq \mathrm{CPU}(R,T)$, suggesting the right problem granularity for which the GPU performs best.

Since the GPU performs all pairs intersection test between the rays and triangles passed to it, its performance is independent of scene structure

$$\mathrm{GPU}(R,T) = O(RT). \tag{9}$$

The running time for $\mathrm{CPU}(R,T)$ is dependent on both scene and camera (sampling) structure since partitioning structures in both triangle and ray space may be used to reduce computation

$$\mathrm{CPU}(R,T) \leq O(RT). \tag{10}$$

As Section 4.4 shows, the constant of proportionality in the $O(RT)$ in (9) is smaller (by at least a factor of two) than the one in (10). Tuning the ray engine will require balancing the raw speed of $\mathrm{GPU}(R,T)$ with the efficiency of $\mathrm{CPU}(R,T)$.

6.1. The Readback Bottleneck

We can model $\mathrm{GPU}(R,T)$ by analyzing the steps in the GPU ray-triangle intersection in terms of GPU operations, and empirically measuring the speed of these operatrions. A simple version of this model sufficient for our analysis is

$$\mathrm{GPU}(R,T) = TR\,\mathrm{fill}^{-1} + R\gamma\,\mathrm{readback}^{-1}, \tag{11}$$

where γ is the number of bytes read back from the graphics card per ray. This model shows that the GPU ray-triangle intersection time is linearly dependent on the number of rays and affinely dependent on the number of triangles. This model does not include the triangle rate, which would add a negligible term proportianal to T to the model. Once we determine values for fill and readback we can then determine the smallest number of triangles T_{min} needed to make GPU ray-triangle intersection practical.

The fill rate is measured in pixels per second (which includes the cost of the fragment shader execution) whereas the readback rate is measured in bytes per second. The fill rate is measured pixels per second instead of bytes per second because it is non-linear in the number of bytes transferred (modern graphics cards can for example multitexture two textures simultaneously). Since our ray engine uses two ray textures (an origins texture and a directions texture) we simply divide the number of rays (pixels) by the fill rate (pixels per second) to get the fragment shader execution time.

We determine values for the fill and readback rates empirically. For example, the GeForce3 achieves a fill rate of 390 MP/sec. (dual-textured pixels) and an AGP 4x readback rate of 250 MB/sec (which is only one quarter of the 1 GB/sec that should be available on the AGP bus). Returning a single 64-bit triangle ID uses a γ of four, whereas returning an additional three single-precision floating-point barycentric coordinates sets γ to 16. Hence we can return triangle ID's at a rate of 62.5M/sec., but when we include barycentrics the rate drops to 15.6M/sec. We can further increase performance by reducing the number of bytes used for the index of each triangle, especially since the ray engine sends smaller buckets of coherent triangles to the GPU.

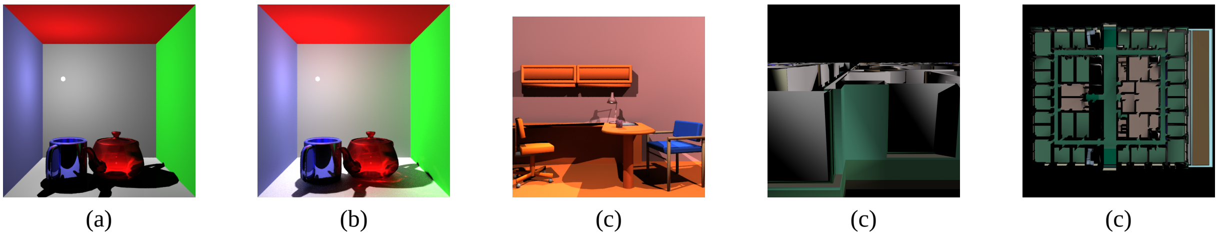


Figure 6: *Images tested by the ray engine: teapot Cornell box ray traced classically (a) and Monte Carlo (b), office (c), and Soda Hall side (d) and top (e) views.*

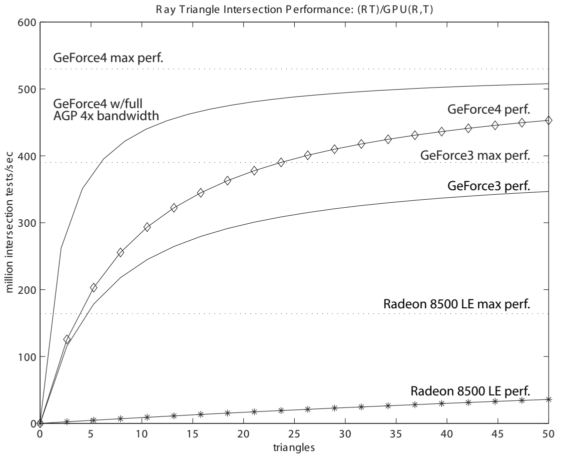


Figure 7: *Theoretical performance in millions of ray-triangle intersection tests per second on the GPU with* $\gamma = 4$.

For small values of T the performance is limited by the readback rate. As T increases, the constant cost of readback is amortized over a larger number of intersections tests. (When we measured peak ray-triangle intersection rates on the Radeon 8500, we sent thousands of triangles to the GPU.) In each case, the curve asymptotically approaches the fill rate, which is listed as the maximum performance possible. Realistically, only smaller values of T should be considered since the GPU intersection routine is an inefficient all-pairs $O(RT)$ solution and our goal is to only send coherent rays and triangles to it.

Figure 7 shows that even for small value of T, the performance is quite competitive with that of a CPU based implementation in spite of the read back rate limitation. For example, the ray-triangle intersections per second for ten triangles clock at 240M on the GeForce3 and 286M on the GeForce4 Ti4600 (if they had the necessary fragment processing capabilities). The recent availability of AGP 8x, and the upcoming AGP 3.0 standard will further reduce the impact of the readback bottleneck and further validate this form of general GPU processing.

6.2. Avoiding Forced Coherence

The previous section constructed a model for the efficiency of the GPU ray-triangle intersection. We must now determine when it is more efficient to use the CPU instead of the GPU.

It is important to exploit triangle and ray coherence only where it exists, and not to force it where it does not. We hence identify the locations in the triangle octree where the ray-triangle coherence is high enough to support efficient GPU intersection. This preprocess occurs after the triangle octree construction, and involves an additional traversal of the octree, identifying cells that represent at least $T_{\min}$ triangles. Since these cells are ideal for GPU processing we refer to these cells as GPU cells.

Rendering employs a standard recursive octree traversal routine. When a ray ray traverses through a cell not tagged as a GPU cell, the standard CPU based ray-triangle intersection is performed. If a ray encounters a GPU cell during its traversal, the ray's traversal is terminated and it is placed in the ray cache for that cell for future processing. When the ray cache corresponding to a given GPU cell reaches $R_{\min}$ rays, its rays and triangles are sent to the GPU for processing using the ray-intersection kernel.

A point may be reached where the ray engine has receive all known rays from the application to be processed. At this poing there may exist GPU cells whose ray cache is non-empty, but containing less than $R_{\min}$ rays. A policy may be chose to select a GPU cell and force its ray cache to be send to the CPU instead of the GPU. This allows the ray engine to continually advance towards completion for rendering the scene.

6.3. Results

We have performed numerous tests to tune the parameters of the geometry engine to eek out the highest performance.

Table 2 demonstrates the utilization of the GPU. As mentioned earlier, only reasonably sized collections of coherent

Scene	% GPU Rays
Teapot Room Classical	89%
Teapot Room Monte-Carlo	71%
Office	65%
Soda Hall Top View	70%
Soda Hall Side View	89%

Table 2: *Percentage of rays sent to the GPU across a variety of scenes and applications.*

rays and triangles are sent to the GPU. The remaining rays and triangles are traced by the CPU. The best performers resulted from classical ray tracing of the teapot room and the ray casting of the Soda Hall side view. The numerous bounces from Monte Carlo ray tracing likely reduce the coherence on all but the eye rays. Coherence was reduced in the office scene due to the numerous small triangles that filled the triangle cache before the ray cache could be optimally filled. The Soda Hall top view contains a lot of disjoint small "silhouette" wall polygons that likely failed to fill the triangle cache for a given optimally filled ray cache.

System	Rays/sec.	Speedup
CPU only	135,812	
plus GPU	165,098	22%
Asynch. Readback	183,273	34%
Infinitely Fast GPU	234,102	73%

Table 3: *Speedup by using the GPU to render the teapot room.*

Table 3 illustrates the efficiency of the ray engine. The readback delay was only responsible for 12% of the potential speedup of 34%. One feature that would allow us to recover that 12% is to be able to issue an asynchronous readback (as is suggested in OpenGL 2.0), such that the CPU and GPU can continue to work during the readback process. The NV_FENCE mechanism could then report when the readback is complete. This feature could possibly be added through the use of threads, but this idea has been left for future research.

The last row of Table 3 shows the estimated speed if we had an infinitely fast GPU, which shows that most of our time is spent on the CPU reorganizing the geometry and rays into coherent structures. This effect has been observed in similar ray tracers[32], where BSP tree traversal is "typically 2-3 times as costly as ray-triangle intersection."

Table 4 shows the effect of tuning the number of triangles that get sent to the GPU. In each of these cases, the number of rays intersected by each GPU pass was set to 64.

Table 5 shows that the number of rays in each bucket can also be varied to achieve peak efficiency. Tuning the ray engine to assign more rays to the GPU frees the CPU to perform more caching. For example, for the teapot room classical ray tracing, we were able to achieve a 52% speedup over the CPU by setting R to 256 and hand tuning the octree resolution.

T	GPU Rays	Rays/sec.	Speedup
CPU		135,812	
4–16	78%	147,630	8%
5–12	81%	157,839	16%
5–15	89%	165,098	22%

Table 4: *Tuning the ray engine by varying the range of triangles T sent to the GPU, measured on the teapot room.*

R	Rays/sec.	Speedup
CPU	135,812	
64	165,098	22%
128	177,647	31%
256	180,558	33%
512	175,904	29%

Table 5: *Tuning the number of rays R sent to the GPU for intersection.*

7. Conclusions

We have added ray tracing to the growing list of applications accelerated by the programmable shaders found in modern graphics cards. Our ray engine performed at speeds comparable to the fastest CPU ray tracers. We expect the GPU will become the high-performance ray-tracing platform of choice due to the rapid growth rate of GPU performance.

By partitioning computation between the CPU and GPU, we combined the best features of both, at the expense of the slow readback of data and the overhead of ray caching. The AGP graphics bus supports high-bandwidth transmission from the CPU to the GPU, but less bandwidth for recovery of results. We expect future bus designs and driver implementations will soon ameliorate this roadblock.

The overhead of ray caching limited the performance speedup of GPU to less than double that of the CPU only, and this overhead as also burdened others[32]. Even though our method for processing the data structures is considered quite efficient[27], we are anxious to explore alternative structures that can more efficiently organize rays and geometry for batch processing by the GPU.

Acknowledgements

This research was supported in part by the NSF grant #ACI-0113968, and by NVidia. The idea of using fragment programs for ray-triangle intersection and the crossbar formalism resulted originally from conversations with Michael McCool.

References

1. Arvo, J., and Kirk, D. B. Fast ray tracing by ray classification. *Proc. SIGGRAPH 87* (July 1987), 55–64.

2. Badouel, D. An efficient ray-polygon intersection. In *Graphics Gems*. Academic Press, Boston, 1990, pp. 390–393, 735.

3. Blinn, J. F., and Newell, M. E. Texture and reflection in computer generated images. *Comm. ACM 19*, 10 (Oct. 1976), 542–547.

4. Clark, J. The geometry engine: A VLSI geometry system for graphics. *Proc. SIGGRAPH 82* (July 1982), 127–133.

5. Diefenbach, P. J., and Badler, N. I. Multi-pass pipeline rendering: Realism for dynamic environments. In *Proc. Symposium on Interactive 3D Graphics* (Apr. 1997), ACM SIGGRAPH, pp. 59–70.

6. Erickson, J. Pluecker coordinates. *Ray Tracing News 10*, 3 (1997), 11. www.acm.org-/tog/resources/RTNews/html/rtnv10n3.html#art11.

7. Fernando, R., Fernandez, S., Bala, K., and Greenberg, D. P. Adaptive shadow maps. *Proc. SIGGRAPH 2001* (Aug. 2001), 387–390.

8. Hall, D. The AR350: Today's ray trace rendering processor. In *Hot 3D Presentations*, P. N. Glagowski, Ed. Graphics Hardware 2001, Aug. 2001, pp. 13–19.

9. Heidrich, W., Lensch, H., and Seidel, H.-P. Light field-based reflections and refractions. *Eurographics Rendering Workshop* (1999).

10. Jensen, H. W. Importance driven path tracing using the photon map. *Proc. Eurographics Rendering Workshop* (Jun. 1995), 326–335.

11. Jensen, H. W., Marschner, S. R., Levoy, M., and Hanrahan, P. A practical model for subsurface light transport. *Proc. SIGGRAPH 2001* (Aug. 2001), 511–518.

12. Kajiya, J. T. The rendering equation. *Proc. SIGGRAPH 86* (Aug. 1986), 143–150.

13. Keller, A. Instant radiosity. *Proc. SIGGRAPH 97* (Aug. 1997), 49–56.

14. Kipfer, P., and Slusallek, P. Transparent distributed processing for rendering. *Proc. Parallel Visualization and Graphics Symposium* (1999), 39–46.

15. Kolb, C., Hanrahan, P. M., and Mitchell, D. A realistic camera model for computer graphics. *Proc. SIGGRAPH 95* (Aug. 1995), 317–324.

16. Lindholm, E., Kilgard, M. J., and Moreton, H. A user-programmable vertex engine. *Proc. SIGGRAPH 2001* (July 2001), 149–158.

17. McCool, M. D., and Heidrich, W. Texture shaders. In *Proc. Graphics Hardware 99* (August 1999), SIGGRAPH/Eurographics Workshop, pp. 117–126.

18. Möller, T., and Trumbore, B. Fast, minimum storage ray-triangle intersectuion. *Journal of Graphics Tools 2*, 1 (1997), 21–28.

19. Parker, S., Martin, W., Sloan, P.-P. J., Shirley, P. S., Smits, B., and Hansen, C. Interactive ray tracing. In *1999 ACM Symposium on Interactive 3D Graphics* (Apr. 1999), ACM SIGGRAPH, pp. 119–126.

20. Peercy, M. S., Olano, M., Airey, J., and Ungar, P. J. Interactive multi-pass programmable shading. *Proc. SIGGRAPH 2000* (2000), 425–432.

21. Pharr, M., Kolb, C., Gershbein, R., and Hanrahan, P. M. Rendering complex scenes with memory-coherent ray tracing. *Proc. SIGGRAPH 97* (Aug. 1997), 101–108.

22. Proudfoot, K., Mark, W. R., Tzvetkov, S., and Hanrahan, P. A real-time procedural shading system for programmable graphics hardware. *Proc. SIGGRAPH 2001* (2001), 159–170.

23. Purcell, T. J. SHARP ray tracing architecture. SIGGRAPH 2001 Real-Time Ray Tracing Course Notes, Aug. 2001.

24. Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. Ray tracing on programmable graphics hardware. *Proc. SIGGRAPH 2002* (July 2002).

25. Reeves, W. T., Salesin, D. H., and Cook, R. L. Rendering antialiased shadows with depth maps. *Proc. of SIGGRAPH 87* (Jul. 1987), 283–291.

26. Reinhard, E., Chalmers, A., and Jansen, F. Overview of parallel photorealistic graphics. *Eurographics '98 STAR* (Sep. 1998), 1–25.

27. Revelles, J., Urena, C., and Lastra, M. An efficient parametric algorithm for octree traversal. *Proc. Winter School on Computer Graphics* (2000).

28. Szirmay-Kalos, L., and Purgathofer, W. Global ray-bundle tracing with hardware acceleration. *Proc. Eurographics Rendering Workshop* (June 1998), 247–258.

29. Trendall, C., and Stewart, A. J. General calculations using graphics hardware with applications to interactive caustics. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering* (Jun. 2000), Eurographics, pp. 287–298.

30. Veach, E., and Guibas, L. J. Metropolis light transport. *Proc. SIGGRAPH 97* (Aug. 1997), 65–76.

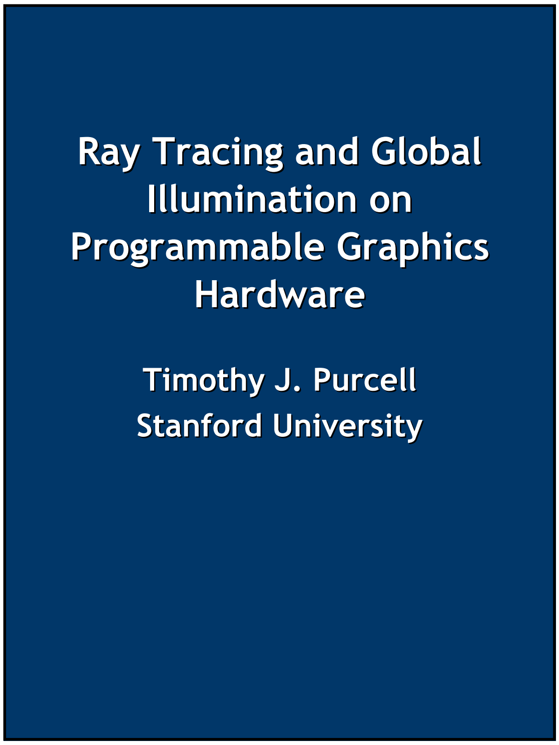
31. Wald, I., Slusallek, P., and Benthin, C. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001* (2001), Eurographics Rendering Workshop, pp. 277–288.

32. Wald, I., Slusallek, P., Benthin, C., and Wagner, M. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164.

33. Ward, G. J. The radiance lighting simulation and rendering system. *Proc. SIGGRAPH 94* (Jul. 1994), 459–472.

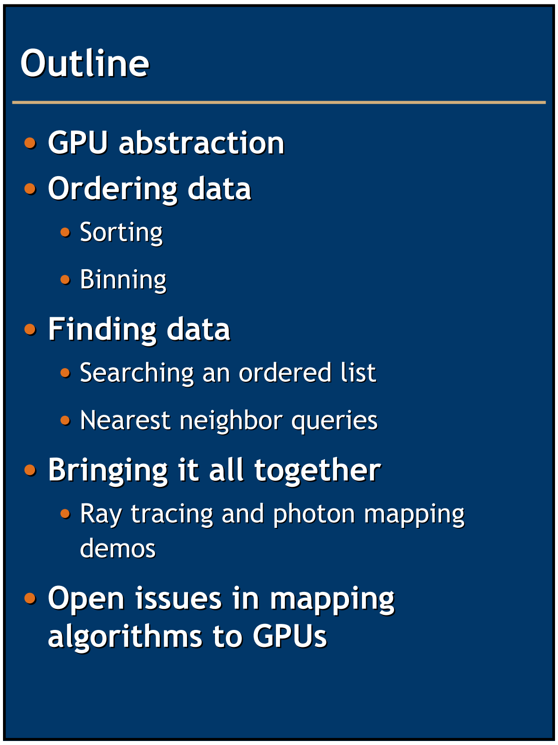
34. Weghorst, H., Hooper, G., and Greenberg, D. Improved computational methods for ray tracing. *ACM Trans. on Graphics 3*, 1 (Jan. 1984), 52–69.

35. Williams, L. Casting curved shadows on curved surfaces. *Proc. SIGGRAPH 78* (Aug. 1978), 270–274.



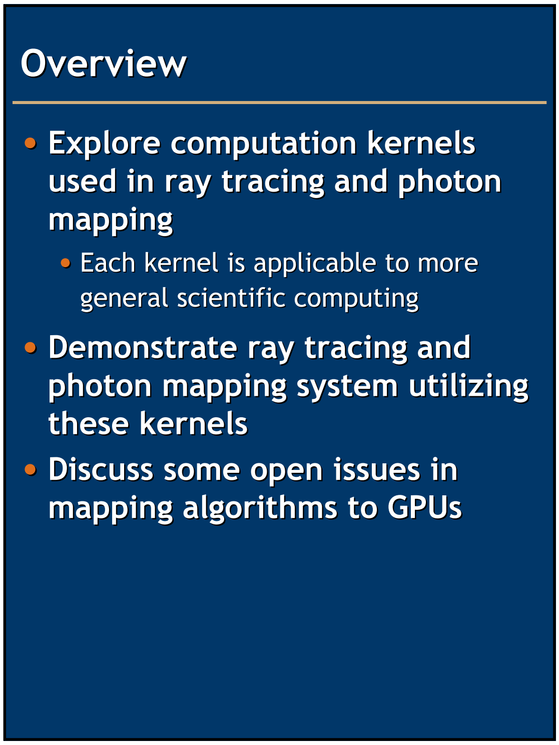
Ray Tracing and Global Illumination on Programmable Graphics Hardware

Timothy J. Purcell
Stanford University



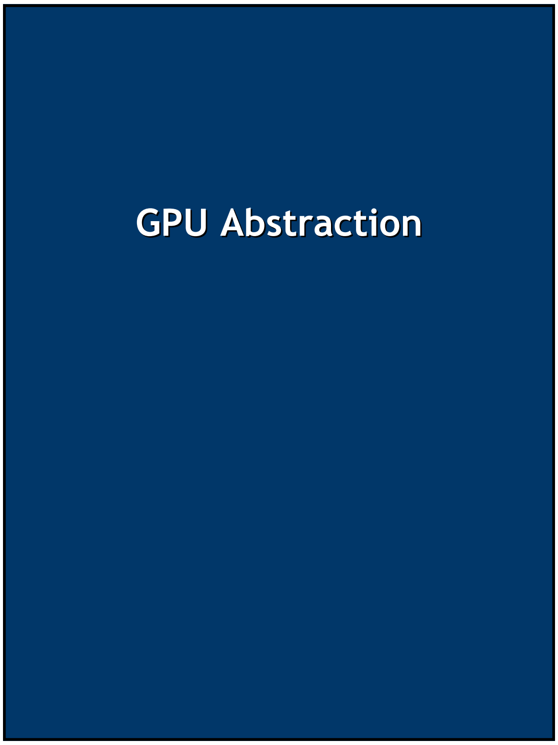
Outline

- GPU abstraction
- Ordering data
  - Sorting
  - Binning
- Finding data
  - Searching an ordered list
  - Nearest neighbor queries
- Bringing it all together
  - Ray tracing and photon mapping demos
- Open issues in mapping algorithms to GPUs

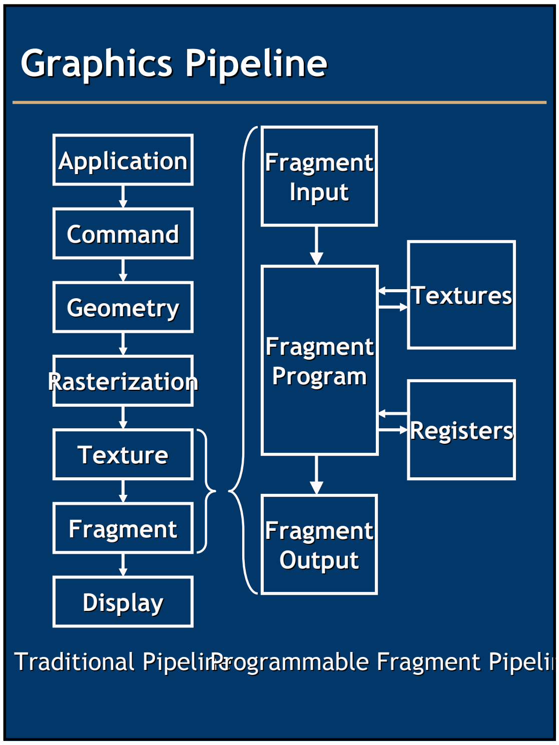


Overview

- Explore computation kernels used in ray tracing and photon mapping
  - Each kernel is applicable to more general scientific computing
- Demonstrate ray tracing and photon mapping system utilizing these kernels
- Discuss some open issues in mapping algorithms to GPUs



GPU Abstraction



Graphics Pipeline

Application

Command

Geometry

Rasterization

Texture

Fragment

Display

Fragment Input

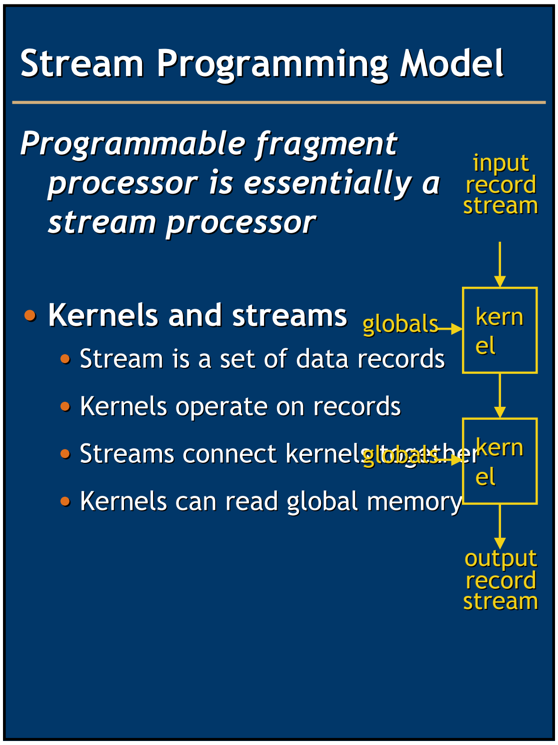
Fragment Program

Textures

Registers

Fragment Output

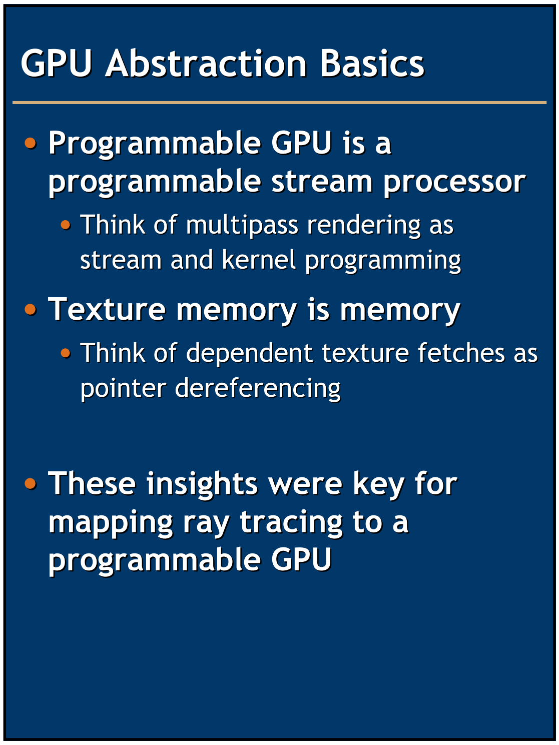
Traditional Pipeli Programmable Fragment Pipeli



Stream Programming Model

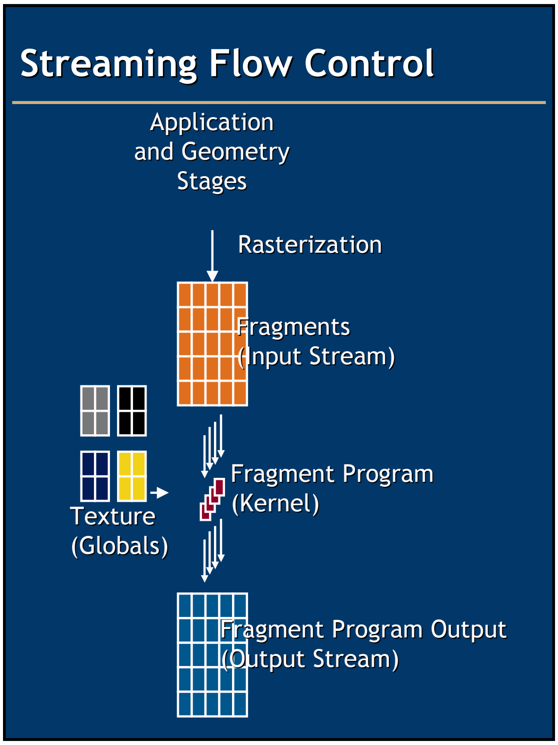
Programmable fragment processor is essentially a stream processor

- **Kernels and streams**
  - Stream is a set of data records
  - Kernels operate on records
  - Streams connect kernels together
  - Kernels can read global memory



GPU Abstraction Basics

- **Programmable GPU is a programmable stream processor**
  - Think of multipass rendering as stream and kernel programming
- **Texture memory is memory**
  - Think of dependent texture fetches as pointer dereferencing
- **These insights were key for mapping ray tracing to a programmable GPU**



Streaming Flow Control

Application and Geometry Stages

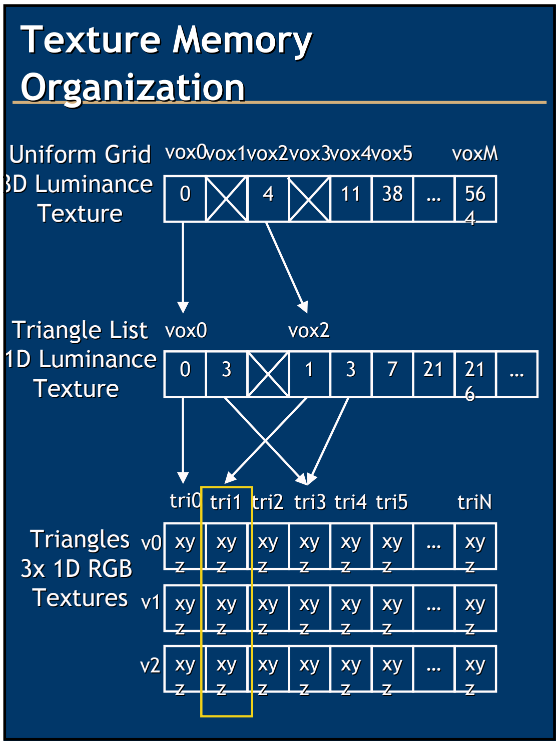
Rasterization

Fragments (Input Stream)

Texture (Globals)

Fragment Program (Kernel)

Fragment Program Output (Output Stream)



Texture Memory Organization

Uniform Grid 3D Luminance Texture

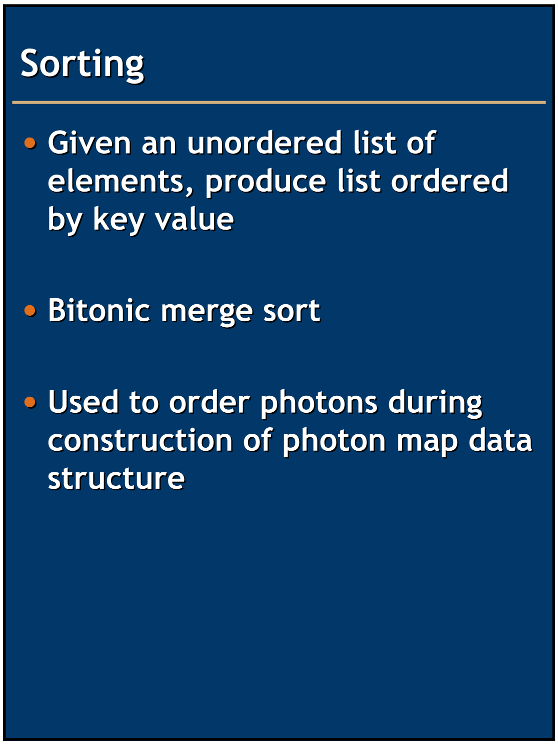
vox0	vox1	vox2	vox3	vox4	vox5		voxM
0	☒	4	☒	11	38	...	564

Triangle List 1D Luminance Texture

vox0			vox2					
0	3	☒	1	3	7	21	216	...

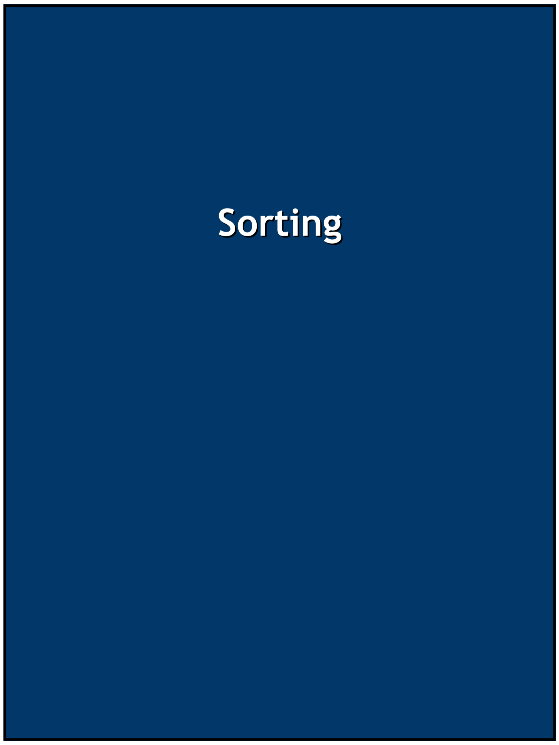
Triangles 3x 1D RGB Textures

	tri0	tri1	tri2	tri3	tri4	tri5		triN
v0	xyz	xyz	xyz	xyz	xyz	xyz	...	xyz
v1	xyz	xyz	xyz	xyz	xyz	xyz	...	xyz
v2	xyz	xyz	xyz	xyz	xyz	xyz	...	xyz

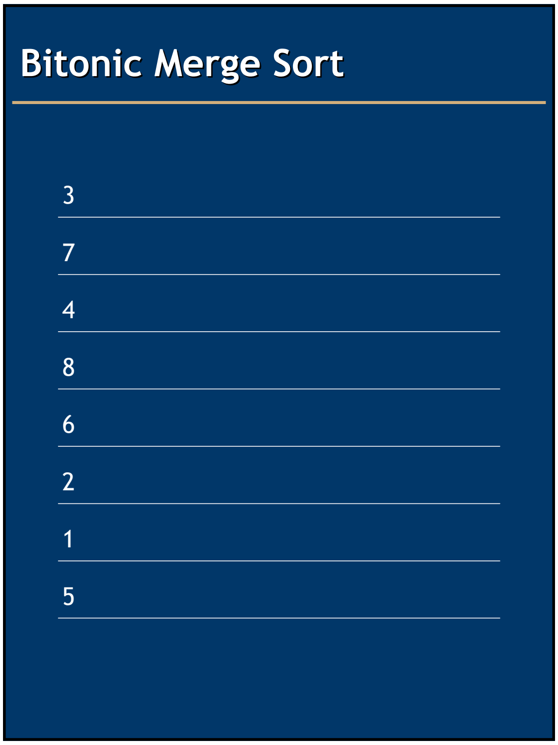


Sorting

- Given an unordered list of elements, produce list ordered by key value
- Bitonic merge sort
- Used to order photons during construction of photon map data structure



Sorting



Bitonic Merge Sort

3

7

4

8

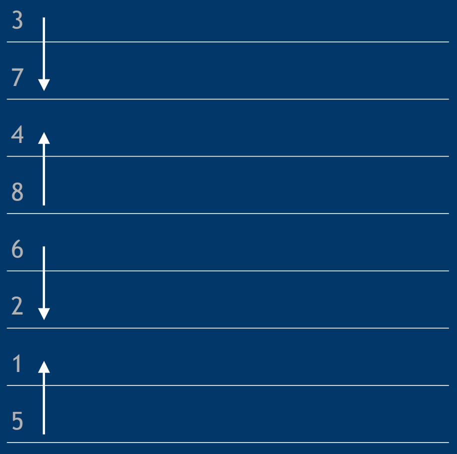
6

2

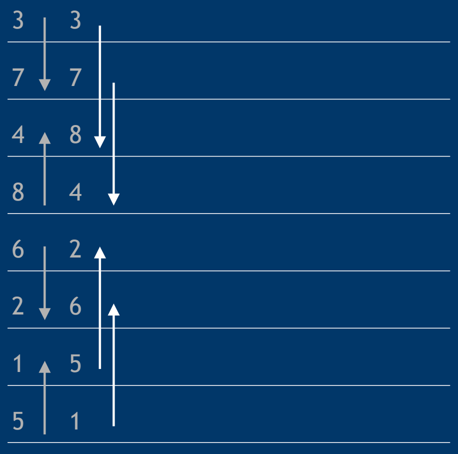
1

5

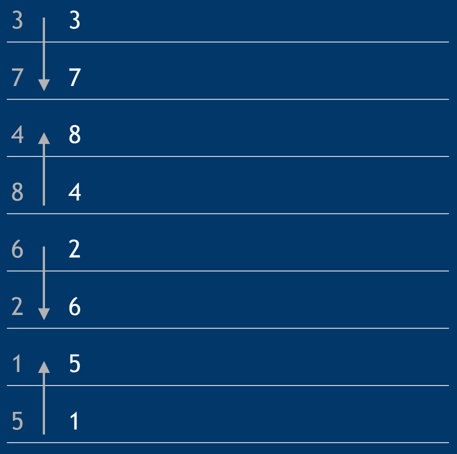
Bitonic Merge Sort



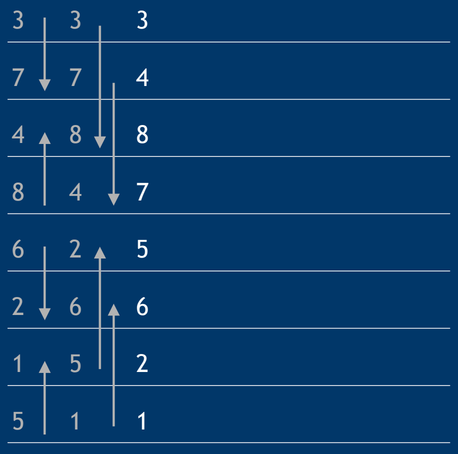
Bitonic Merge Sort

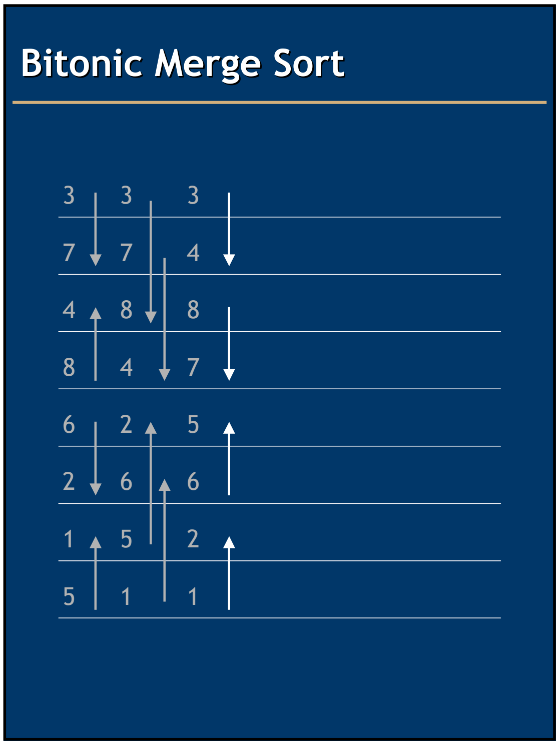


Bitonic Merge Sort



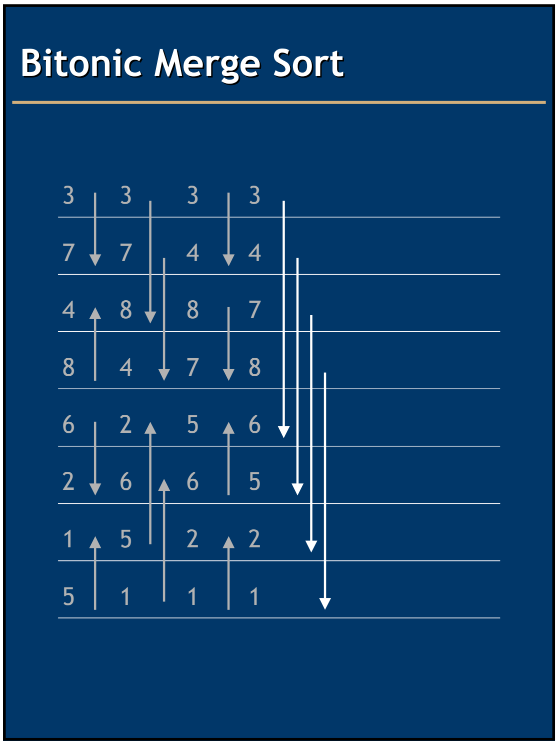
Bitonic Merge Sort





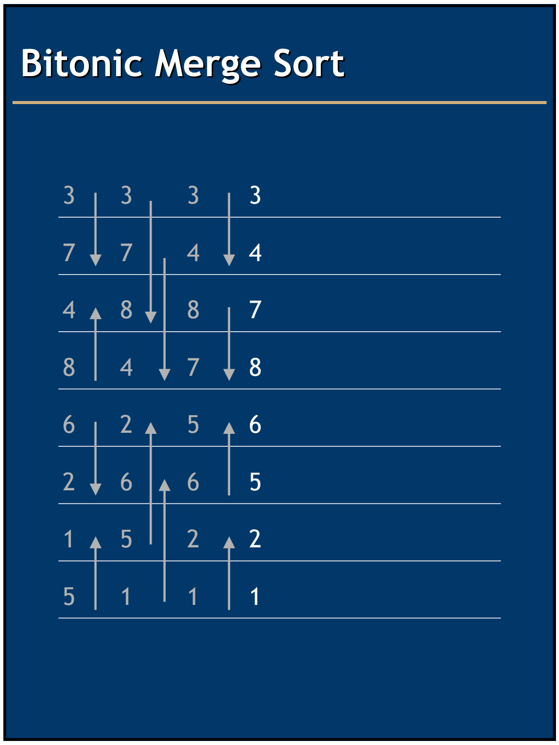
Bitonic Merge Sort

3	3	3
7	7	4
4	8	8
8	4	7
6	2	5
2	6	6
1	5	2
5	1	1



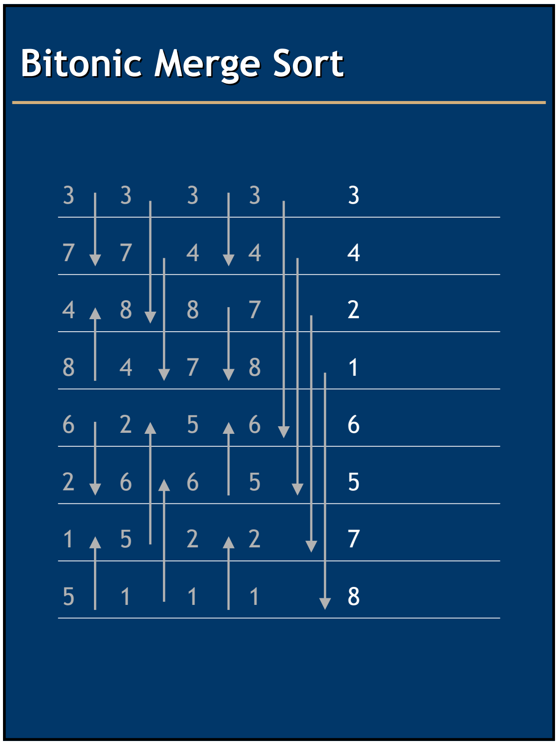
Bitonic Merge Sort

3	3	3	3
7	7	4	4
4	8	8	7
8	4	7	8
6	2	5	6
2	6	6	5
1	5	2	2
5	1	1	1



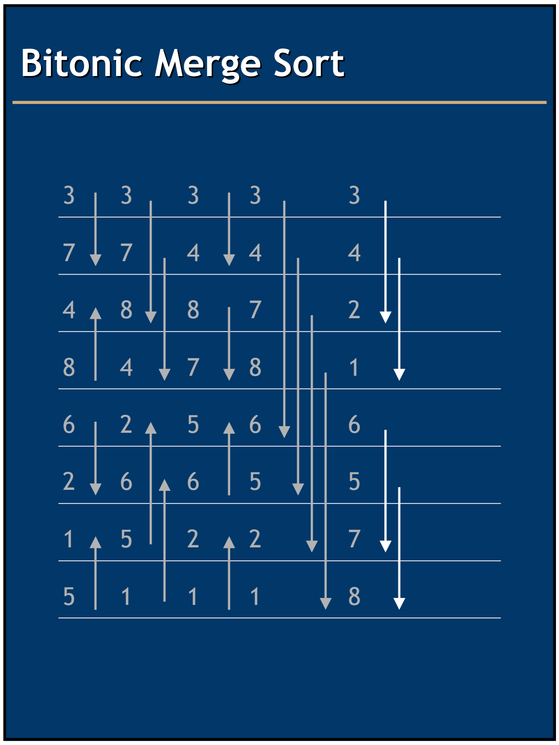
Bitonic Merge Sort

3	3	3	3
7	7	4	4
4	8	8	7
8	4	7	8
6	2	5	6
2	6	6	5
1	5	2	2
5	1	1	1



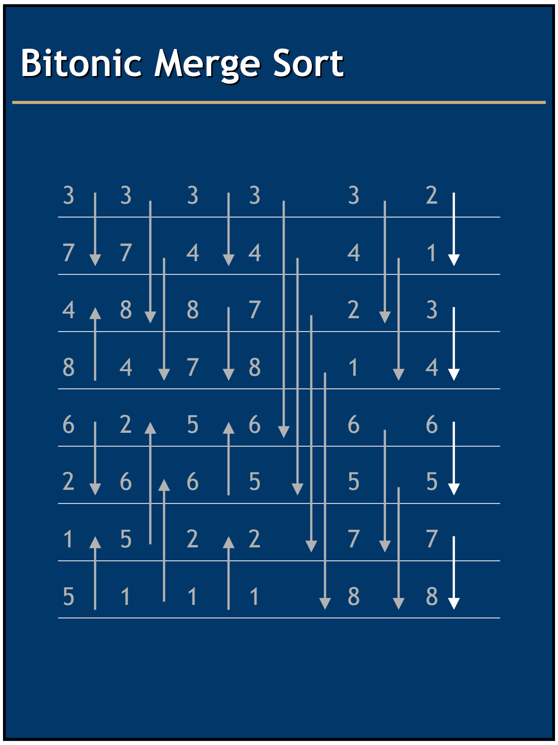
Bitonic Merge Sort

3	3	3	3	3
7	7	4	4	4
4	8	8	7	2
8	4	7	8	1
6	2	5	6	6
2	6	6	5	5
1	5	2	2	7
5	1	1	1	8



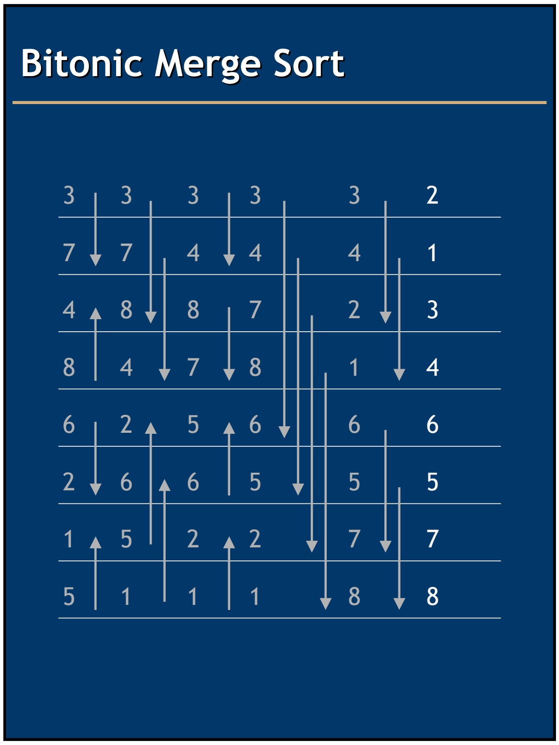
Bitonic Merge Sort

3	3	3	3	3
7	7	4	4	4
4	8	8	7	2
8	4	7	8	1
6	2	5	6	6
2	6	6	5	5
1	5	2	2	7
5	1	1	1	8



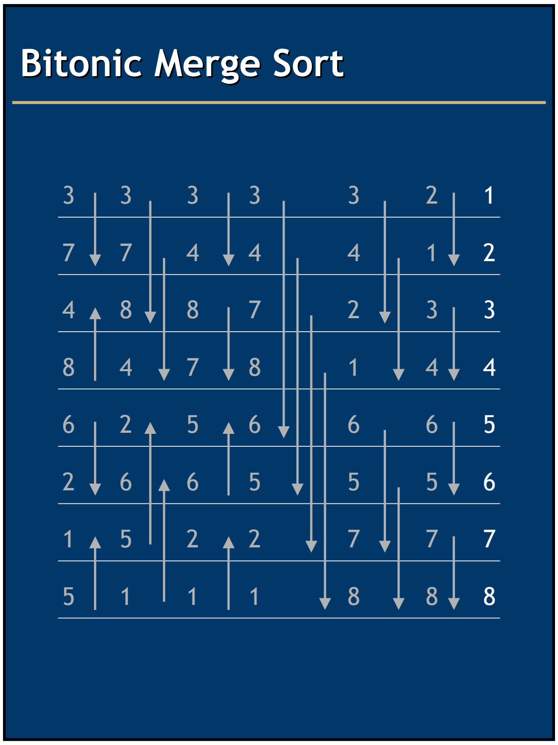
Bitonic Merge Sort

3	3	3	3	3	2
7	7	4	4	4	1
4	8	8	7	2	3
8	4	7	8	1	4
6	2	5	6	6	6
2	6	6	5	5	5
1	5	2	2	7	7
5	1	1	1	8	8



Bitonic Merge Sort

3	3	3	3	3	2
7	7	4	4	4	1
4	8	8	7	2	3
8	4	7	8	1	4
6	2	5	6	6	6
2	6	6	5	5	5
1	5	2	2	7	7
5	1	1	1	8	8



Bitonic Merge Sort

3	3	3	3	3	2	1
7	7	4	4	4	1	2
4	8	8	7	2	3	3
8	4	7	8	1	4	4
6	2	5	6	6	6	5
2	6	6	5	5	5	6
1	5	2	2	7	7	7
5	1	1	1	8	8	8

Bitonic Merge Sort Summary

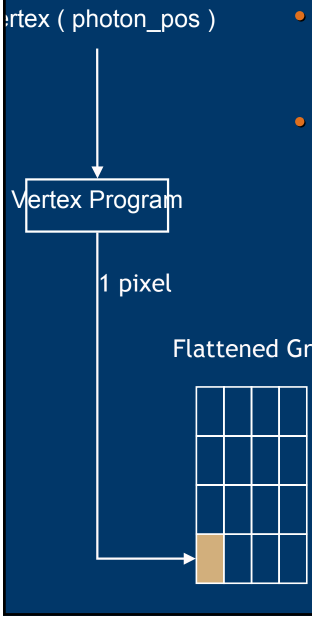
- Separate rendering pass for each swap
  - O($\log^2 n$) passes
- Hand coded to around 20 instructions
  - 512x512 elements - 3060 instructions per pixel

Binning

- Given an unordered list of elements, produce list ordered by key value
  - Multiple elements with same key are binned together as they have no ordering amongst themselves
- Stencil binning
- Alternate approach to building the photon map data structure

Binning

Stencil Binning



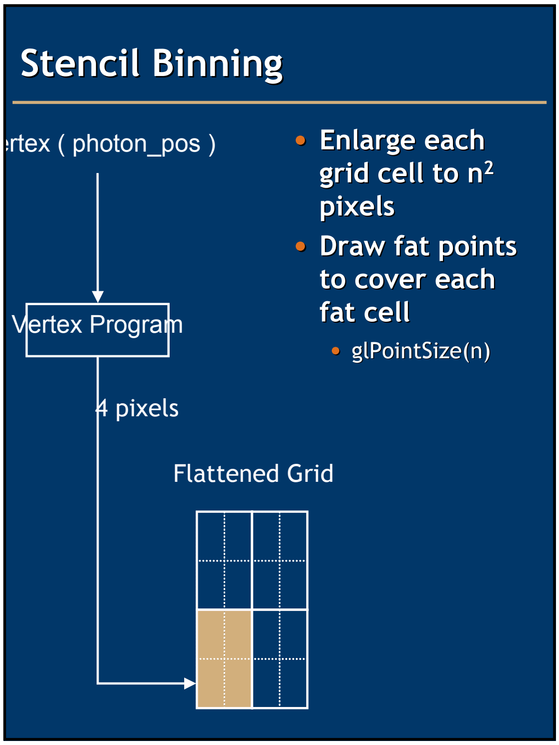
rtex (photon_pos)

Vertex Program

1 pixel

Flattened Grid

- Treat framebuffer as a flattened grid
- Vertex program sends each photon to its grid cell
  - Only the last photon in each cell is saved



Stencil Binning

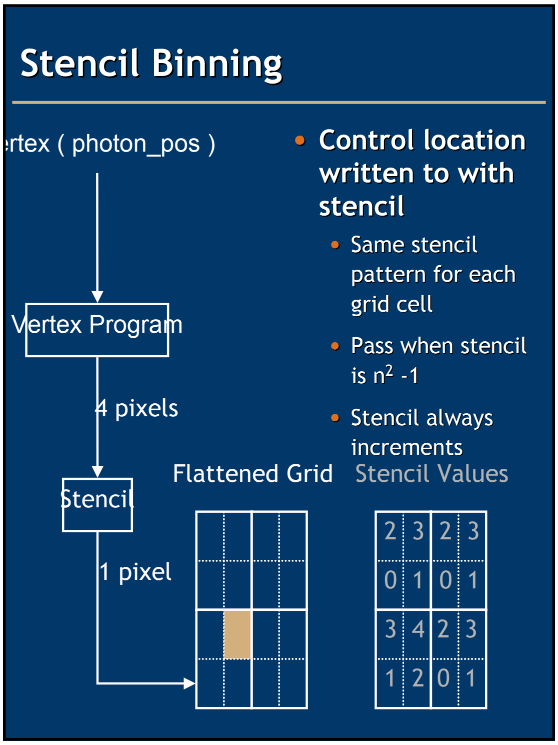
- Enlarge each grid cell to n^2 pixels
- Draw fat points to cover each fat cell
  - glPointSize(n)

rtex (photon_pos)

Vertex Program

4 pixels

Flattened Grid



Stencil Binning

- Control location written to with stencil
  - Same stencil pattern for each grid cell
  - Pass when stencil is n^2 -1
  - Stencil always increments

rtex (photon_pos)

Vertex Program

4 pixels

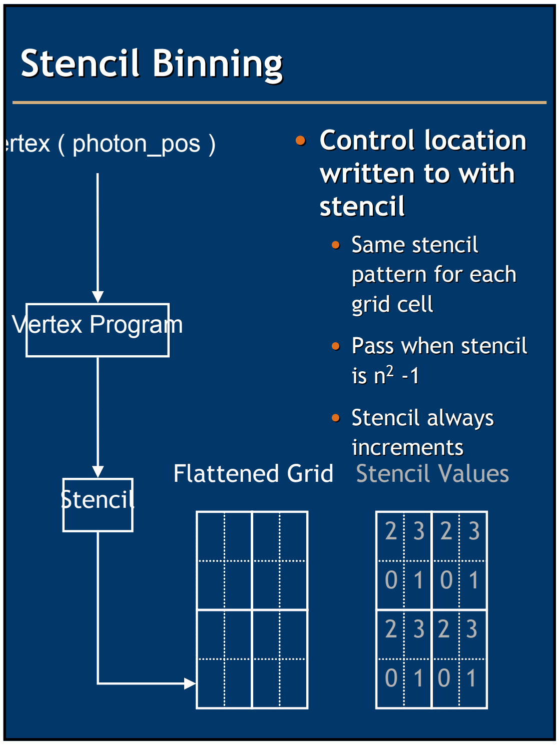
Stencil

1 pixel

Flattened Grid

Stencil Values

2	3	2	3
0	1	0	1
3	4	2	3
1	2	0	1



Stencil Binning

- Control location written to with stencil
  - Same stencil pattern for each grid cell
  - Pass when stencil is n^2 -1
  - Stencil always increments

rtex (photon_pos)

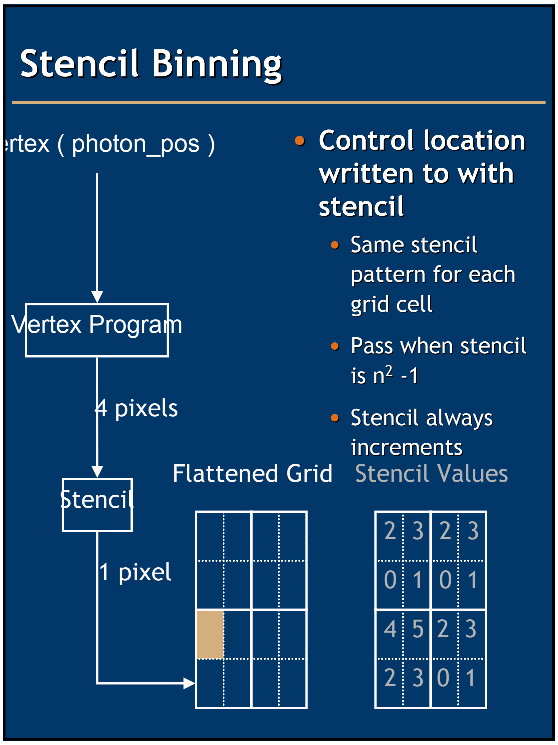
Vertex Program

Stencil

Flattened Grid

Stencil Values

2	3	2	3
0	1	0	1
2	3	2	3
0	1	0	1



Stencil Binning

- Control location written to with stencil
  - Same stencil pattern for each grid cell
  - Pass when stencil is n^2 -1
  - Stencil always increments

rtex (photon_pos)

Vertex Program

4 pixels

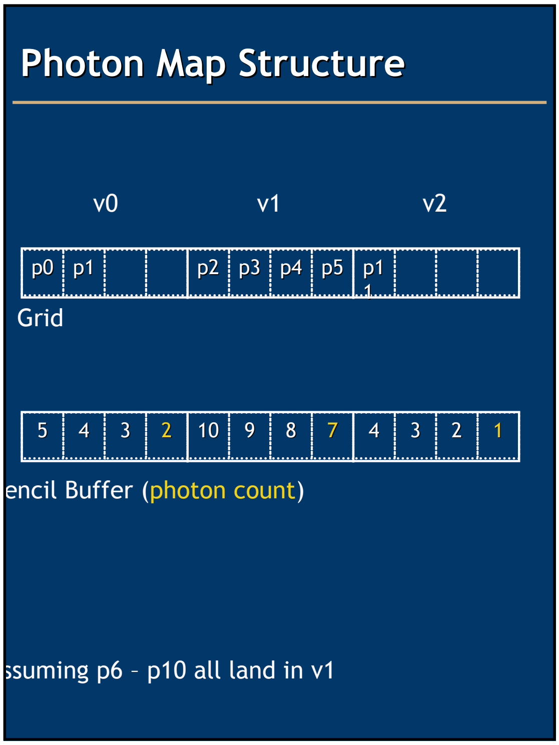
Stencil

1 pixel

Flattened Grid

Stencil Values

2	3	2	3
0	1	0	1
4	5	2	3
2	3	0	1



Photon Map Structure

v0				v1				v2			
p0	p1			p2	p3	p4	p5	p1 1			

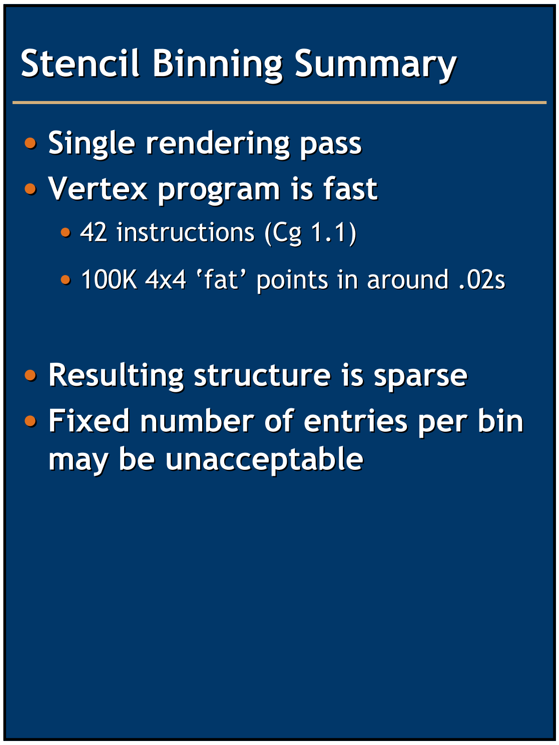
Grid

5	4	3	2	10	9	8	7	4	3	2	1

encil Buffer (photon count)

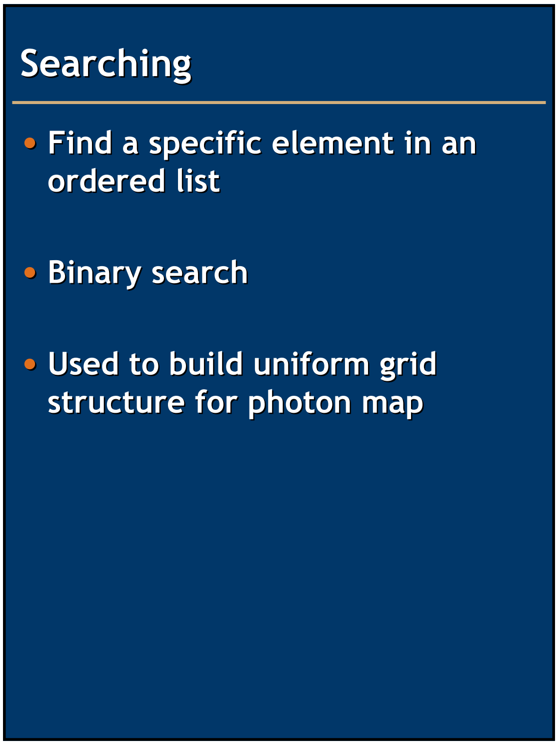
ssuming p6 - p10 all land in v1

Searching



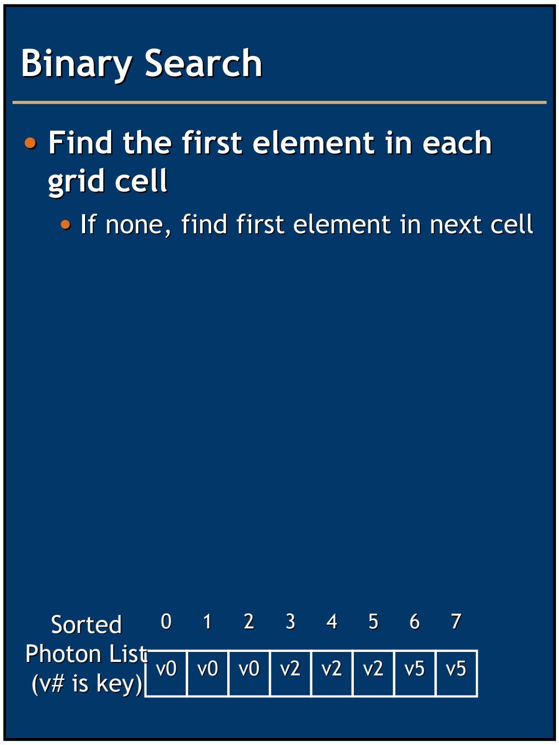
Stencil Binning Summary

- Single rendering pass
- Vertex program is fast
  - 42 instructions (Cg 1.1)
  - 100K 4x4 'fat' points in around .02s
- Resulting structure is sparse
- Fixed number of entries per bin may be unacceptable



Searching

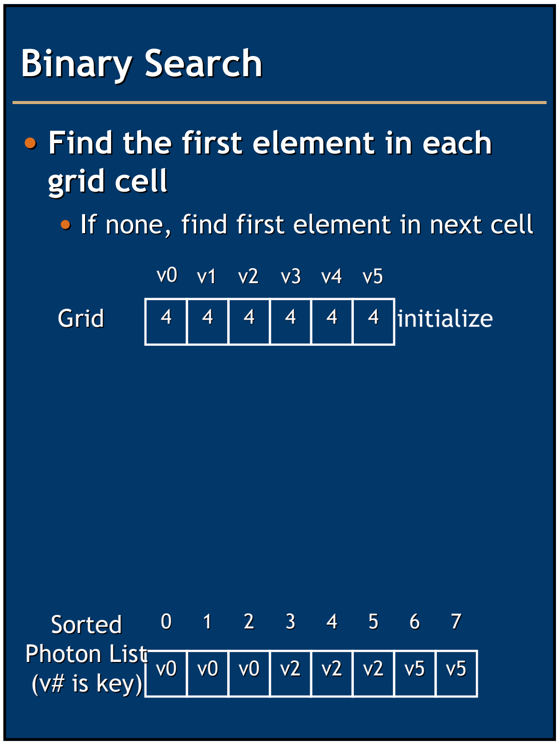
- Find a specific element in an ordered list
- Binary search
- Used to build uniform grid structure for photon map



Binary Search

- Find the first element in each grid cell
  - If none, find first element in next cell

Sorted Photon List (v# is key)	0	1	2	3	4	5	6	7
	v0	v0	v0	v2	v2	v2	v5	v5

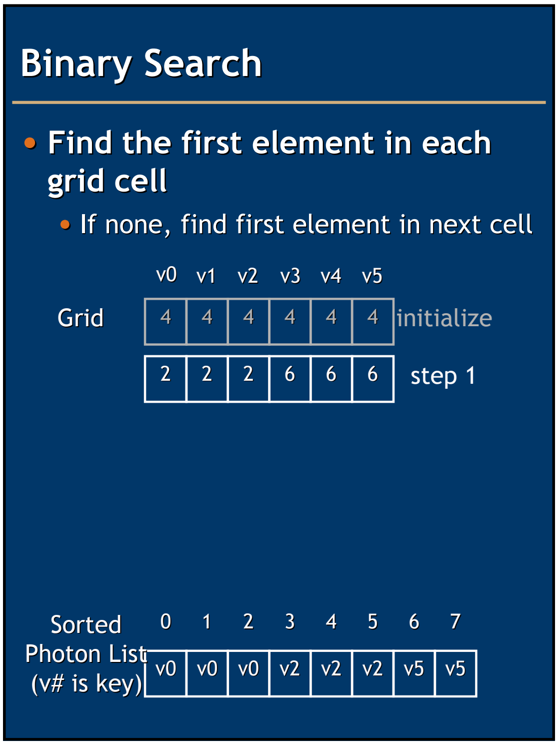


Binary Search

- Find the first element in each grid cell
  - If none, find first element in next cell

	v0	v1	v2	v3	v4	v5	
Grid	4	4	4	4	4	4	initialize

Sorted Photon List (v# is key)	0	1	2	3	4	5	6	7
	v0	v0	v0	v2	v2	v2	v5	v5

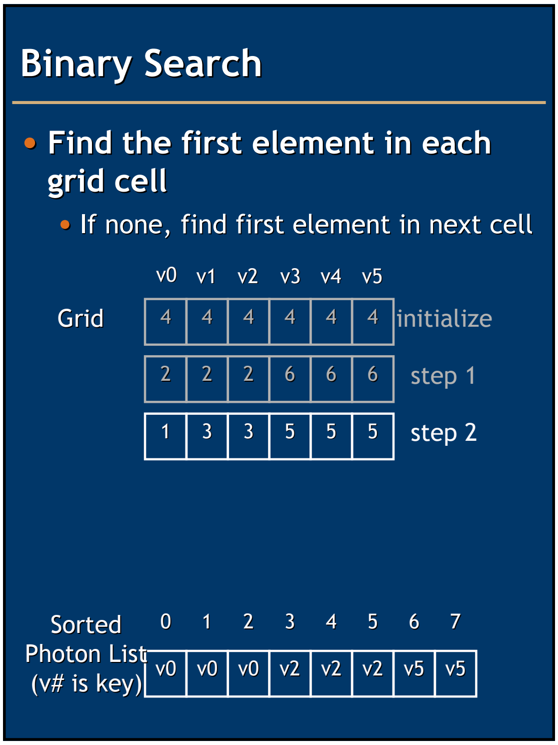


Binary Search

- Find the first element in each grid cell
  - If none, find first element in next cell

	v0	v1	v2	v3	v4	v5	
Grid	4	4	4	4	4	4	initialize
	2	2	2	6	6	6	step 1

Sorted Photon List (v# is key)	0	1	2	3	4	5	6	7
	v0	v0	v0	v2	v2	v2	v5	v5



Binary Search

- Find the first element in each grid cell
  - If none, find first element in next cell

	v0	v1	v2	v3	v4	v5	
Grid	4	4	4	4	4	4	initialize
	2	2	2	6	6	6	step 1
	1	3	3	5	5	5	step 2

Sorted Photon List (v# is key)	0	1	2	3	4	5	6	7
	v0	v0	v0	v2	v2	v2	v5	v5

Binary Search

- Find the first element in each grid cell
  - If none, find first element in next cell

Grid	v0	v1	v2	v3	v4	v5	
	4	4	4	4	4	4	initialize
	2	2	2	6	6	6	step 1
	1	3	3	5	5	5	step 2
	0	2	2	6	6	6	step 3

Sorted Photon List (v# is key)	0	1	2	3	4	5	6	7
	v0	v0	v0	v2	v2	v2	v5	v5

Binary Search Summary

- Single rendering pass
- O(log n) steps
  - 18 instructions per step (Cg 1.1)
  - 512x512 elements - 342 instructions per pixel

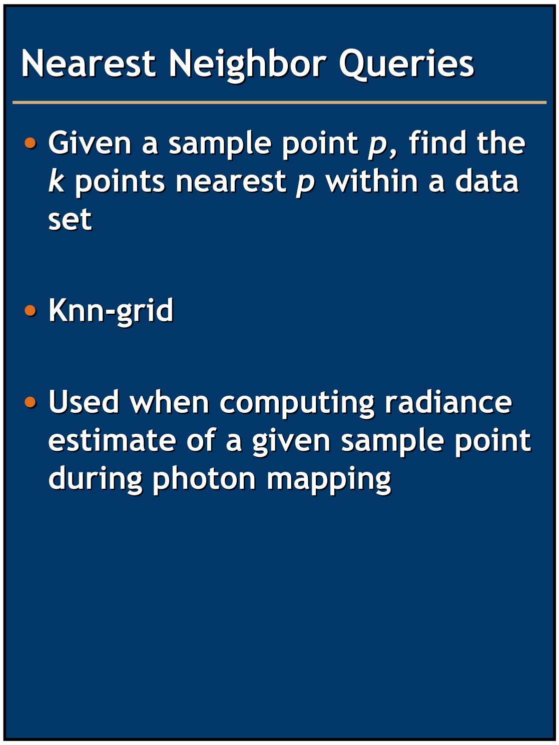
Binary Search

- Find the first element in each grid cell
  - If none, find first element in next cell

Grid	v0	v1	v2	v3	v4	v5	
	4	4	4	4	4	4	initialize
	2	2	2	6	6	6	step 1
	1	3	3	5	5	5	step 2
	0	2	2	6	6	6	step 3
	0	3	3	6	6	6	step 4

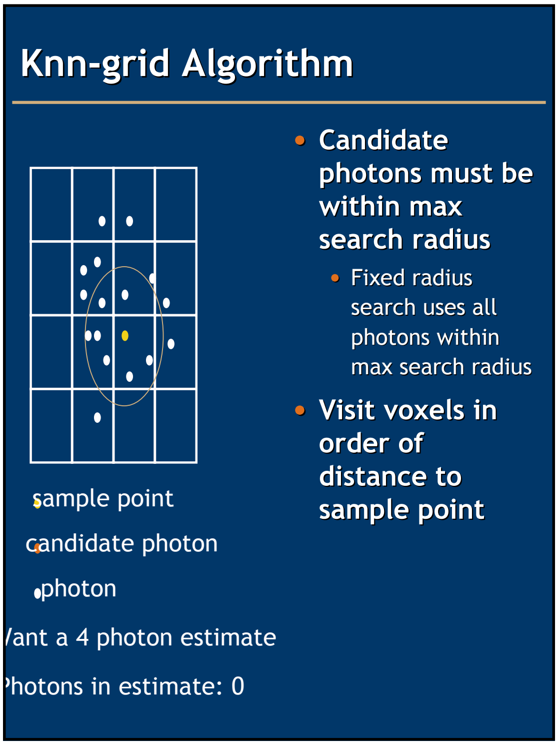
Sorted Photon List (v# is key)	0	1	2	3	4	5	6	7
	v0	v0	v0	v2	v2	v2	v5	v5

Nearest Neighbor Queries



Nearest Neighbor Queries

- Given a sample point *p*, find the *k* points nearest *p* within a data set
- Knn-grid
- Used when computing radiance estimate of a given sample point during photon mapping



Knn-grid Algorithm

- Candidate photons must be within max search radius
  - Fixed radius search uses all photons within max search radius
- Visit voxels in order of distance to sample point

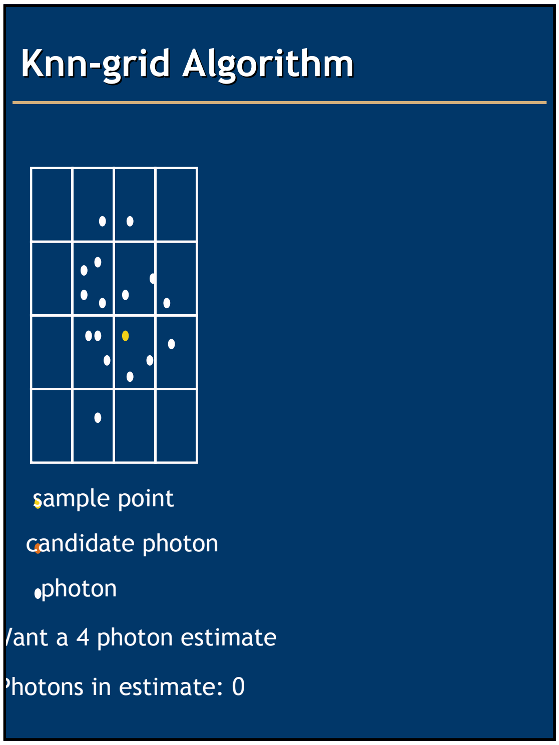
● sample point

● candidate photon

● photon

/ant a 4 photon estimate

'hotons in estimate: 0



Knn-grid Algorithm

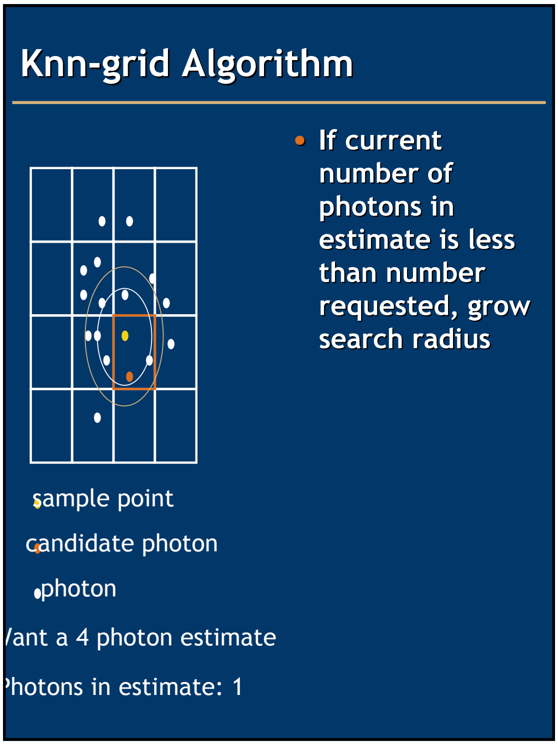
● sample point

● candidate photon

● photon

/ant a 4 photon estimate

'hotons in estimate: 0



Knn-grid Algorithm

- If current number of photons in estimate is less than number requested, grow search radius

● sample point

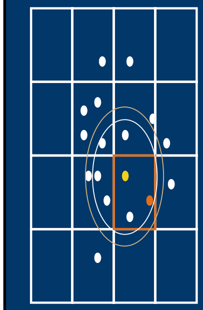
● candidate photon

● photon

/ant a 4 photon estimate

'hotons in estimate: 1

Knn-grid Algorithm



- If current number of photons in estimate is less than number requested, grow search radius

● sample point

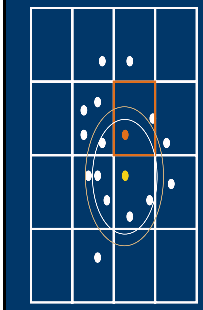
● candidate photon

● photon

/ant a 4 photon estimate

'hotons in estimate: 2

Knn-grid Algorithm



- Add photons within search radius

● sample point

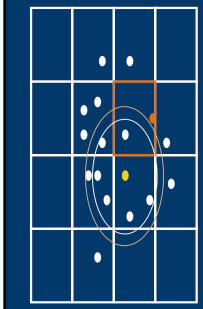
● candidate photon

● photon

/ant a 4 photon estimate

'hotons in estimate: 3

Knn-grid Algorithm



- Don’t add photons outside maximum search radius
- Don’t grow search radius when photon is outside maximum radius

● sample point

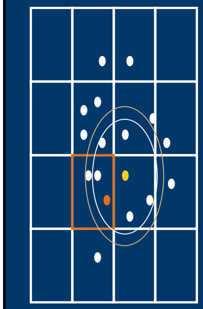
● candidate photon

● photon

/ant a 4 photon estimate

'hotons in estimate: 2

Knn-grid Algorithm



- Add photons within search radius

● sample point

● candidate photon

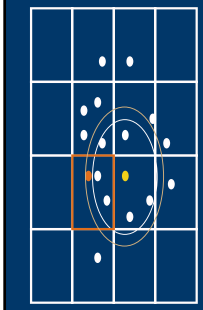
● photon

/ant a 4 photon estimate

'hotons in estimate: 4

Knn-grid Algorithm

- Don’t expand search radius if enough photons already found



● sample point

● candidate photon

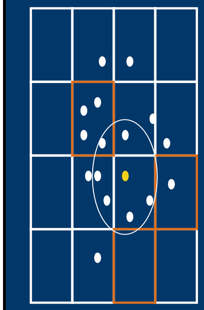
● photon

/ant a 4 photon estimate

'hotons in estimate: 4

Knn-grid Algorithm

- Visit all other voxels accessible within determined search radius
- Add photons within search radius



● sample point

● candidate photon

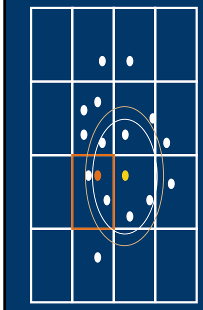
● photon

/ant a 4 photon estimate

'hotons in estimate: 6

Knn-grid Algorithm

- Add photons within search radius



● sample point

● candidate photon

● photon

/ant a 4 photon estimate

'hotons in estimate: 5

Knn-grid Summary

- Requires 95 instructions per candidate neighbor
  - Plus 65 instructions per pass overhead
  - Compiled under Cg 1.1
- Locates more than *k* nearest neighbors

Bringing It All Together

Ray Tracing and Photon Mapping Demos

Open Issues

- Compute mask, branching, or stream buffer?
  - Need some way to prevent execution of expensive programs over programmed subset of pixels
- Read-modify-write buffers
  - Efficient save and restore over multiple passes
- Addressing modes
  - 2D textures vs. 1D textures
- Integer computation
  - Sometimes you need a mod or a div
- Readback speed

Open Issues in Mapping Algorithms to GPUs

Acknowledgements

- Pat Hanrahan
- Ian Buck, Bill Mark
- Craig Donner, Mike Cammarano, Henrik Wann Jensen
- Funding
  - Nvidia Graduate Research Fellowship
  - ATI, DARPA, MERL, Nvidia, Sony, Sun
- ATI - James Percy, Bob Drebin
- Nvidia - Matt Papakipos, David Kirk, Kurt Akeley

Interactive Walkthroughs using Multiple GPUs

Dinesh Manocha
University of North Carolina at Chapel Hill

Interactive Visibility Culling in Complex Environments using Occlusion-Switches

Naga K. Govindaraju Avneesh Sud Sung-Eui Yoon Dinesh Manocha
University of North Carolina at Chapel Hill
{naga,sud,sungeui,dm}@cs.unc.edu
http://gamma.cs.unc.edu/switch

Abstract: *We present occlusion-switches for interactive visibility culling in complex 3D environments. An occlusion-switch consists of two GPUs (graphics processing units) and each GPU is used to either compute an occlusion representation or cull away primitives not visible from the current viewpoint. Moreover, we switch the roles of each GPU between successive frames. The visible primitives are rendered in parallel on a third GPU. We utilize frame-to-frame coherence to lower the communication overhead between different GPUs and improve the overall performance. The overall visibility culling algorithm is conservative up to image-space precision. This algorithm has been combined with levels-of-detail and implemented on three networked PCs, each consisting of a single GPU. We highlight its performance on complex environments composed of tens of millions of triangles. In practice, it is able to render these environments at interactive rates with little loss in image quality.*

CR Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling

Keywords: Interactive display, multiple GPUs, conservative occlusion culling, parallel rendering, levels-of-detail

1 Introduction

Interactive display and walkthrough of large geometric environments currently pushes the limits of graphics technology. Environments composed of tens of millions of primitives are common in applications such as simulation-based design of large man-made structures, architectural visualization, or urban simulation. In spite of the rapid progress in the performance of graphics processing units (GPUs), it is not possible to render such complex datasets at interactive rates, i.e., 20 frames a second or more, on current graphics systems.

Many rendering algorithms that attempt to minimize the number of primitives sent to the graphics processor during each frame have been developed. These are based on visibility culling, level-of-detail modeling, sample-based representations, etc. Their goal is to not render any primitives that the user will not ultimately see. These techniques have been extensively studied in computer graphics and related areas.

In this paper, we primarily deal with occlusion culling. Our goal is to cull away a subset of the primitives that are not visible from the current viewpoint. Occlusion culling has been well-studied in the literature and the current algorithms can be classified into different categories. Some are specific to certain types of models, such as architectural or urban environments. Others require extensive preprocessing of visibility, or the presence of large, easily identifiable occluders in the scene, and may not work well for complex environments. The most general algorithms use some combination of object-space hierarchies and image-space occlusion representation. These algorithms can be further classified into three categories:

1. **Specialized Architectures**: Some specialized hardware architectures have been proposed for occlusion culling [Greene et al. 1993; Greene 2001].

2. **Readbacks and Software Culling:** These algorithms read back the frame-buffer or depth-buffer, build a hierarchy, and perform occlusion culling in software [Greene et al. 1993; Zhang et al. 1997; Baxter et al. 2002]. However, readbacks can be expensive (e.g. 50 milliseconds to read back the $1K \times 1K$ depth-buffer on a Dell 530 Workstation with NVIDIA GeForce 4 card).

3. **Utilize Hardware Occlusion Queries:** Many vendors have been supporting image-space occlusion queries. However, their use can impose an additional burden on the graphics pipeline and can sometimes result in reduced throughput and frame rate [Klowoski and Silva 2001].

Overall, occlusion culling is considered quite expensive and hard to achieve in real-time for complex environments.

Main Contribution: We present a novel visibility culling algorithm based on *occlusion-switches*. An occlusion-switch consists of two graphics processing units (GPUs). During each frame, one of the GPUs renders the occluders and computes an occlusion representation, while the second GPU performs culling in parallel using an image-space occlusion query. In order to avoid any depth-buffer readbacks and perform significant occlusion culling, the two GPUs switch their roles between successive frames. The visible primitives computed by the occlusion-switch are rendered in parallel on a third GPU. The algorithm utilizes frame-to-frame coherence to compute occluders for each frame as well as lower the bandwidth or communication overhead between different GPUs. We have combined the occlusion-culling algorithm with static levels-of-detail (LODs) and used it for interactive walkthrough of complex environments. Our current implementation runs on three networked PCs, each consisting of a NVIDIA GeForce 4 graphics processor, and connected using Ethernet. We highlight the performance of our algorithm on three complex environments: a Powerplant model with more than 13 million triangles, a Double Eagle tanker with more than 82 million triangles and a part of a Boeing 777 airplane with more than 20 million triangles. Our system, SWITCH, is able to render these models at $10 - 20$ frames per second with little loss in image quality. However, our algorithm based on occlusion-switches introduces one frame of latency into the system.

As compared to earlier approaches, our overall occlusion culling and rendering algorithm offers the following advantages:

1. **Generality**: It makes no assumption about the scene and is applicable to all complex environments.

2. **Conservative Occlusion Culling:** The algorithm performs conservative occlusion up to screen-space image precision.

3. **Low Bandwidth:** The algorithm involves no depth-buffer readback from the graphics card. The bandwidth requirements between different GPUs varies as a function of the changes in the visible primitives between successive frames (e.g. a few kilobytes per frame).

4. **Significant Occlusion Culling:** As compared to earlier approaches, our algorithm culls away a higher percentage of primitives not visible from the current viewpoint.

5. **Practicality:** Our algorithm can be implemented on commodity hardware and only assumes hardware support for the occlusion query, which is becoming widely available. Furthermore, we obtain $2 - 3$ times improvement in frame rate as compared to earlier algorithms.

Organization: The rest of the paper is organized in the following manner. We give a brief overview of previous work on parallel rendering and occlusion culling in Section 2. Section 3 presents occlusion-switches and analyzes the bandwidth requirements. In Section 4, we combine our occlusion culling algorithm with pre-computed levels-of-detail and use it to render large environments. We describe its implementation and highlight its performance on three complex environments in Section 5. Finally, we highlight areas for future research in Section 6.

2 Related Work

In this section, we give a brief overview of previous work on occlusion culling and parallel rendering.

2.1 Occlusion Culling

The problem of computing portions of the scene visible from a given viewpoint has been well-studied in computer graphics and computational geometry. A recent survey of different algorithms is given in [Cohen-Or et al. 2001]. In this section, we give a brief overview of occlusion culling algorithms. These algorithms aim to cull away a subset of the primitives that are occluded by other primitives and, therefore, are not visible from the current viewpoint.

Many occlusion culling algorithms have been designed for specialized environments, including architectural models based on cells and portals [Airey et al. 1990; Teller 1992] and urban datasets composed of large occluders [Coorg and Teller 1997; Hudson et al. 1997; Schaufler et al. 2000; Wonka et al. 2000; Wonka et al. 2001]. However, they may not be able to obtain significant culling on large environments composed of a number of small occluders.

Algorithms for general environments can be broadly classified based on whether they are conservative or approximate, whether they use object space or image space hierarchies, or whether they compute visibility from a point or a region. The conservative algorithms compute the *potentially visible set* (PVS) that includes all the visible primitives, plus a small number of potentially occluded primitives [Coorg and Teller 1997; Greene et al. 1993; Hudson et al. 1997; Klowoski and Silva 2001; Zhang et al. 1997]. On the other hand, the approximate algorithms include most of the visible objects but may also cull away some of the visible objects [Bartz et al. 1999; Klowoski and Silva 2000; Zhang et al. 1997]. Object space algorithms make use of spatial partitioning or bounding volume hierarchies; however, performing "occluder fusion" on scenes composed of small occluders with object space methods is difficult. Image space algorithms including the hierarchical Z-buffer (HZB) [Greene et al. 1993; Greene 2001] or hierarchical occlusion maps (HOM) [Zhang et al. 1997] are generally more capable of capturing occluder fusion.

It is widely believed that none of the current algorithms can compute the PVS at interactive rates for complex environments on current graphics systems [El-Sana et al. 2001]. Some of the recent approaches are based on region-based visibility computation, hardware-based visibility queries and multiple graphics pipelines in parallel.

2.2 Region-based Visibility Algorithms

These algorithms pre-compute visibility for a region of space to reduce the runtime overhead [Durand et al. 2000; Schaufler et al. 2000; Wonka et al. 2000]. Most of them work well for scenes with large or convex occluders. Nevertheless, a trade-off occurs between the quality of the PVS estimation for a region and the memory overhead. These algorithms may be extremely conservative or unable to obtain significant culling on scenes composed of small occluders.

2.3 Hardware Visibility Queries

A number of image-space visibility queries have been added by manufacturers to their graphics systems to accelerate visibility computations. These include the HP occlusion culling extensions, item buffer techniques, ATI's HyperZ extensions etc. [Bartz et al. 1999; Klowoski and Silva 2001; Greene 2001; Meissner et al. 2002; Hillesl et al. 2002]. All these algorithms use the GPU to perform occlusion queries as well as render the visible geometry. As a result, only a fraction of a frame time is available for rasterizing the visible geometry and it is non-trivial to divide the time between performing occlusion queries and rendering the visible primitives. If a scene has no occluded primitives, this approach will slow down the overall performance. Moreover, the effectiveness of these queries varies based on the model and the underlying hardware.

2.4 Multiple Graphics Pipelines

The use of an additional graphics system as a visibility server has been used by [Wonka et al. 2001; Baxter et al. 2002]. The approach presented by Wonka et al. [2001] computes the PVS for a region at runtime in parallel with the main rendering pipeline and works well for urban environments. However, it uses the *occluder shrinking* algorithm [Wonka et al. 2000] to compute the region-based visibility, which works well only if the occluders are large and volumetric in nature. The method also makes assumptions about the user's motion.

Baxter et al. [2002] used a two-pipeline based occlusion culling algorithm for interactive walkthrough of complex 3D environments. The resulting system, GigaWalk, uses a variation of two-pass HZB algorithm that reads back the depth buffer and computes the hierarchy in software. GigaWalk has been implemented on a SGI Reality Monster and uses two Infinite Reality pipelines and three CPUs. In Section 5, we compare the performance of our algorithm with GigaWalk.

2.5 Parallel Rendering

A number of parallel algorithms have been proposed in the literature to render large datasets on shared-memory systems or clusters of PCs. These algorithms include techniques to assign different parts of the screen to different PCs [Samanta et al. 2000]. Other cluster-based approaches include WireGL, which allows a single serial application to drive a tiled display over a network [Humphreys et al. 2001] as well as parallel rendering with k-way replication [Samanta et al. 2001]. The performance of these algorithms varies with different environments as well as the underlying hardware. Most of these approaches are application independent and complementary to our parallel occlusion algorithm that uses a cluster of three PCs for interactive display.

Parallel algorithms have also been proposed for interactive ray-tracing of volumetric and geometric models on a shared-memory multi-processor system [Parker et al. 1999]. A fast algorithm for distributed ray-tracing of highly complex models has been described in [Wald et al. 2001].

3 Interactive Occlusion Culling

In this section, we present occlusion-switches and use them for visibility culling. The resulting algorithm uses multiple graphics processing units (GPUs) with image-space occlusion query.

3.1 Occlusion Representation and Culling

An occlusion culling algorithm has three main components. These include:

1. Compute a set of occluders that correspond to an approximation of the visible geometry.

2. Compute an occlusion representation.

3. Use the occlusion representation to cull away primitives that are not visible.

Different culling algorithms perform these steps either explicitly or implicitly. We use an image-based occlusion representation because it is able to perform "occluder fusion" on possibly disjoint occluders [Zhang et al. 1997]. Some of the well-known image-based hierarchical representations include HZB [Greene et al. 1993] and HOM [Zhang et al. 1997]. However, the current GPUs do not support these hierarchies in the hardware. Many two-pass occlusion culling algorithms rasterize the occluders, read back the frame-buffer or depth-buffer, and build the hierarchies in software [Baxter et al. 2002; Greene et al. 1993; Zhang et al. 1997].

However, reading back a high resolution frame-buffer or depth-buffer can be slow on PC architectures. Moreover, constructing the hierarchy in software incurs additional overhead.

We utilize the hardware-based occlusion queries that are becoming common on current GPUs. These queries scan-convert the specified primitives (e.g. bounding boxes) to check whether the depth of any pixels changes. Different queries vary in their functionality. Some of the well-known occlusion queries based on the OpenGL culling extension include the HP_Occlusion_Query (`http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion\_test.txt`) and the NVIDIA OpenGL extension GL_NV_occlusion_query (`http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion\_query.txt`). These queries can sometime stall the pipelines while waiting for the results. As a result, we use a specific GPU during each frame to perform only these queries.

Our algorithm uses the visible geometry from frame i as an approximation to the occluders for frame $i+1$. The occlusion representation implicitly corresponds to the depth buffer after rasterizing all these occluders. The occlusion tests are performed using hardware-based occlusion queries. The *occlusion switches* are used to compute the occlusion representation and perform these queries.

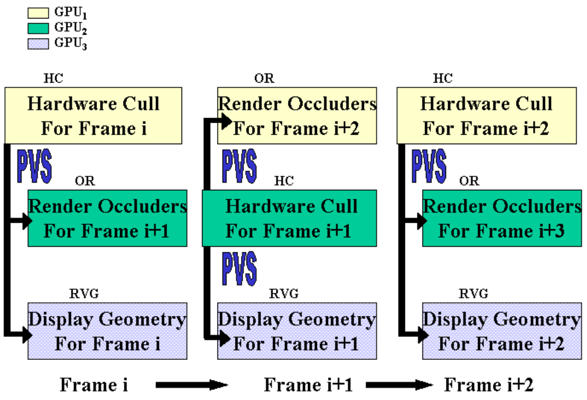

GPU$_1$
GPU$_2$
GPU$_3$
HC
Hardware Cull For Frame i
OR
Render Occluders For Frame i+2
HC
Hardware Cull For Frame i+2
PVS
OR
Render Occluders For Frame i+1
PVS
HC
Hardware Cull For Frame i+1
PVS
OR
Render Occluders For Frame i+3
PVS
RVG
Display Geometry For Frame i
RVG
Display Geometry For Frame i+1
RVG
Display Geometry For Frame i+2
Frame i
Frame i+1
Frame i+2

Figure 1: *System Architecture: Each color represents a separate GPU. Note that GPU$_1$ and GPU$_2$ switch their roles each frame with one performing hardware culling and other rendering occluders. GPU3 is used as a display client.*

3.2 Occlusion-Switch

An occlusion-switch takes the camera for frame $i+1$ as input and transmits the potential visible set and camera for frame i as the output to the renderer. The occlusion-switch is composed of two GPUs, which perform the following functions, each running on a separate GPU in parallel:

- **Compute Occlusion Representation (OR)**: Render the occluders to compute the occlusion representation. The occluders for frame $i+1$ correspond to the visible primitives from frame i.

- **Hardware Culling (HC)**: Enable the occlusion query state on the GPU and render the bounding boxes corresponding to the scene geometry. Use the image-space occlusion query to determine the visibility of each bounding box and compute the PVS. Moreover, we disable modifications to the depth buffer while performing these queries.

During a frame, each GPU in the occlusion-switch performs either OR or HC and at the end of the frame the two GPUs inter-change their function. The depth buffer computed by OR during the previous frame is used by HC to perform the occlusion queries during the current frame. Moreover, the visible nodes computed by HC correspond to the PVS. The PVS is rendered in parallel on a third GPU and is used by the OR for the next frame to compute the occlusion representation. The architecture of the overall system is shown in Fig. 1. The overall occlusion algorithm involves no depth buffer readbacks from the GPUs.

3.3 Culling Algorithm

The occlusion culling algorithm uses an occlusion-switch to compute the PVS and renders them in parallel on a separate GPU. Let GPU_1 and GPU_2 constitute the occlusion-switch and GPU_3 is used to render the visible primitives (RVG). In an occlusion-switch, the GPU performing HC requires OR for occlusion tests. We circumvent the problem of transmitting occlusion representation from the GPU generating OR to GPU performing hardware cull tests by "switching" their roles between successive frames as shown in Fig. 1. For example, GPU_1 is performing HC for frame i and sending visible nodes to GPU_2 (to be used to compute OR for frame $i+1$) and GPU_3 (to render visible geometry for frame i). For frame $i+1$, GPU_2 has previously computed OR for frame $i+1$. As a result, GPU_2 performs HC, GPU_1 generates the OR for frame $i+2$ and GPU_3 displays the visible primitives.

3.4 Incremental Transmission

The HC process in the occlusion culling algorithm computes the PVS for each frame and sends it to the OR and RVG. To minimize the communication overhead, we exploit frame-to-frame coherence in the list of visible primitives. All the GPUs keep track of the visible nodes in the previous frame and the GPU performing HC uses this list and only transmits the changes to the other two GPUs. The GPU performing HC sends the visible nodes to OR and RVG, and therefore, it has information related to the visible set on HC. Moreover, the other two processes, OR and RVG, maintain the visible set as they receive visible nodes from HC. To reduce the communication bandwidth, we transmit only the difference in the visible sets for the current and previous frames. Let V_i represent the potential visible set for frame i and $\delta_{j,k} = V_j - V_k$ be the difference of two sets. During frame i, HC transmits $\delta_{i,i-1}$ and $\delta_{i-1,i}$ to OR and RVG, respectively. We reconstruct V_i at OR and RVG based on the following formulation:

$$V_i = (V_{i-1} - \delta_{i-1,i}) \cup \delta_{i,i-1}.$$

In most interactive applications, we expect that the size of the set $\delta_{i-1,i} \cup \delta_{i,i-1}$ is much smaller than that of V_i.

3.5 Bandwidth Requirements

In this section, we discuss the bandwidth requirements of our algorithm for a distributed implementation on three different graphics systems (PCs). Each graphics system consists of a single GPU and they are connected using a network. In particular, we map each node of the scene by the same node identifier across the three different graphics systems. We transmit this integer node identifier across the network from the GPU performing HC to each of the GPUs performing OR and RVG. This procedure is more efficient than sending all the triangles that correspond to the node as it requires relatively smaller bandwidth per visible node (i.e. 4 bytes per node). So, if the number of visible nodes is n, then GPU performing HC must send $4n$ bytes per frame to each OR and RVG client. Here n refers to the number of visible objects and not the visible polygons. We can reduce the header overhead by sending multiple integers in a packet. However, this process can introduce some extra latency in the pipeline due to buffering. Moreover, the size of camera parameters is 72 bytes; consequently, the bandwidth requirement per frame is $8n + nh/b + 3(72 + h)$ bytes, where h is the size of header in bytes and buffer size b is the number of node-ids in a packet. If the frame rate is f frames per second, the total bandwidth required is $8nf + nhf/b + 216f + 3hf$. If we send visible nodes by incremental transmission, then n is equal to the size of $\delta_{i,i-1} \cup \delta_{i-1,i}$.

4 Interactive Display

In this section, we present our overall rendering algorithm for interactive display of large environments. We use the occlusion culling algorithm described above and combines it with pre-computed static levels-of-detail (LODs) to render large environments. We represent our environment using a scene graph, as described in [Erikson et al. 2001]. We describe the scene graph representation and the occlusion culling algorithm. We also highlight many optimizations used to improve the overall performance.

4.1 Scene Graph

Our rendering algorithm uses a scene graph representation along with pre-computed static LODs. Each node in the scene graph stores references to its children as well as its parent. In addition, we store the bounding box of each node in the scene graph, which is used for view frustum culling and occlusion queries. This bounding box may correspond to an axis-aligned bounding box (AABB) or an oriented bounding box (OBB). We pre-compute the LODs for each node in the scene graph along with hierarchical levels-of-detail (HLODs) for each intermediate node in the scene graph [Erikson et al. 2001]. Moreover, each LOD and HLOD is represented as a separate node in the scene graph and we associate an error deviation metric that approximately corresponds to the Hausdorff distance between the original model and the simplified object. At runtime, we project this error metric to the screen space and compute the maximum deviation in the silhouette of the original object and its corresponding LOD or HLOD. Our rendering algorithm uses an upper bound on the maximum silhouette deviation error and selects the lowest resolution LOD or HLOD that satisfies the error bound.

```
HardwareCull(Camera *cam)
queue = root of scene graph
disable color mask and depth mask
while( queue is not empty)
do
    node = pop(queue)
    visible= OcclusionTest(node)
    if(visible)
        if(error(node) < pixels of error)
            Send node to OR and RVG
        else
            push children of node to end of queue
        endif
    end if
end do
```

ALGORITHM 4.1: *Pseudo code for Hardware cull (HC). OcclusionTest renders the bounding box and returns either the number of visible pixels or a boolean depending upon the implementation of query. The function error(node) returns the screen space projection error of the node. Note that if the occlusion test returns the number of visible pixels, we could use it to compute the level at which it must be rendered.*

4.2 Culling Algorithm

At runtime, we traverse the scene graph and cull away portions of geometry that are not visible. The visibility of a node is computed by rendering its bounding box against the occlusion representation and querying if it is visible or not. Testing the visibility of a bounding box is a fast and conservative way to reject portions of the scene that are not visible. If the bounding box of the node is visible, we test whether any of the LODs or HLODs associated with that node meet the pixel-deviation error-bound. If one of the LODs or HLODs is selected, we include that node in the PVS and send it to the GPU performing OR for the next frame as well as to the GPU performing RVG for the current frame. If the node is visible and none of the HLODs associated with it satisfy the simplification error bound, we traverse down the scene graph and apply the procedure recursively on each node. On the other hand, if the bounding box of the node is not visible, we do not render that node or any node in the sub-tree rooted at the current node.

The pseudocode for the algorithm is described in Algorithm 4.1. The image-space occlusion query is used to perform view frustum culling as well as occlusion culling on the bounding volume.

4.3 Occluder Representation Generation

At runtime, if we are generating OR for frame $i+1$, we receive camera $i+1$ from RVG and set its parameters. We also clear its depth and color buffer. While OR receives nodes from GPU performing HC, we render them at the appropriate level of detail. An end-of-frame identifier is sent from HC to notify that no more nodes need to be rendered for this frame.

4.4 Occlusion-Switch Algorithm

We now describe the algorithm for the "switching" mechanism described in Section 3. The two GPU's involved in the occlusion-switch toggle or interchange their roles of performing HC and generating OR. We use the algorithms described in sections 4.2 and 4.3 to perform HC and OR, respectively. The pseudocode for the resulting algorithm is shown in Algorithm 4.2.

```
1   if GPU is generating OR
2       camera=grabLatestCam()
3    end if
4   Initialize the colormask and depth mask to true.
5    if GPU is performing HC
6       Send Camera to RVG
7    else /*GPU needs to render occluders */
8       Clear depth buffer
9    end if
10    Set the camera parameters
11    if GPU is performing HC
12        HardwareCull(camera)
13        Send end of frame to OR and RVG
14    else /* Render occluders */
15        int id= end of frame +1 ;
16        while(id!=end of frame)
17        do
18            id=receive node from HC
19            render(id, camera);
20        end do
21    end if
22    if GPU is performing HC
23        do OR for next frame
24    else
25        do HC for next frame
26    end if
```

ALGORITHM 4.2: *The main algorithm for the implementation of occlusion-switch. Note that we send the camera parameters to the RVG client at the beginning of HC (on line 6) in order to reduce latency.*

4.5 Render Visible Geometry

The display client, RVG, receives the camera for the current frame from HC. In addition, it receives the visible nodes in the scene graph and renders them at the appropriate level-of-detail. Moreover, the display client transmits the camera information to the GPU's involved in occlusion-switch based on user interaction. The colormask and depthmask are set to true during initialization.

4.6 Incremental Traversal and Front Tracking

The traversal of scene graph defines a cut that can be partitioned into a visible front and an occluded front.

- **Visible Front**: Visible front is composed of all the visible nodes in the cut. In addition, each node belonging to the visible front satisfies the screen space error metric while its parent does not.

- **Occluded Front**: Occluded front is composed of all the occluded nodes in the cut. Also, note that an occluded node may not satisfy the screen space error metric.

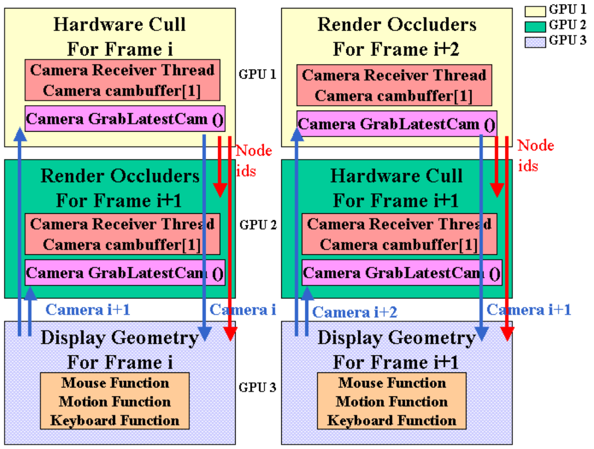


Figure 2: *System Overview: Each color represents a separate GPU with GPU_1 and GPU_2 forming a switch and GPU_3 as the display client. Each of GPU_1 and GPU_2 has a camera-receiver thread and receives camera parameters when the client transmits them due to user's motion and stores them in a camera buffer of size one. The GPU performing OR grabs the latest camera from this thread as the camera position for the next frame. Notice that in this design, the GPU performing HC doesn't have any latency in terms of receiving the camera parameters.*

We reduce the communication overhead by keeping track of the visible and occluded fronts from the previous frame at each GPU. Each node in the front is assigned one of the following states:

- **Over-refined**: Both the node and its parent satisfy the silhouette deviation metric in screen space.

- **Refined**: The node satisfies the silhouette deviation metric while the parent does not.

- **Under-refined**: The node does not satisfy the silhouette deviation metric.

Each node in the front is updated depending upon its state. If the node is *Over-refined*, we traverse up the scene graph to reach a parent node which is *Refined*. If the node is *Under-refined*, we traverse down the scene graph generating a set of *Refined* children nodes. At the beginning of each frame, both OR and RVG update the state of each node in the visible front before rendering it.

We also render each node in $\delta_{i,i-1}$ at OR and RVG. At the end of the frame, the visible nodes for the current frame are reconstructed as described in Section 3.4. The update of the state of each node is important for maintaining the conservative nature of the algorithm.

At the GPU performing HC, we also maintain the occluded front in addition to the visible front of previous frame. This enables us to compute $\delta_{i,i-1}$ efficiently by performing culling on the occluded front before the visible front. A node in the occluded front is refined only if it is in the Over-refined state. Each of the occluded fronts and visible fronts is refined before performing culling algorithm on the refined fronts. Moreover, $\delta_{i,i-1}$ is a part of the refined occluded front.

4.7 Optimizations

We use a number of optimizations to improve the performance of our algorithms, including:

- **Multiple Occlusion Tests**: Our culling algorithm performs multiple occlusion tests using GL_NV_occlusion_query; this avoids immediate readback of occlusion identifiers, which can stall the pipeline. More details on implementation are described in section 4.7.1.

- **Visibility for LOD Selection**: We utilize the number of visible pixels of geometry queried using GL_NV_occlusion_query in selecting the appropriate LOD. Details are discussed in section 4.7.2.

4.7.1 Multiple Occlusion Tests

Our rendering algorithm performs several optimizations to improve the overall performance. The GL_NV_occlusion_query on NVIDIA GeForce 3 and GeForce 4 cards allows multiple occlusion queries at a time and query the results at a later time. We traverse the scene graph in a breadth first manner and perform all possible occlusion queries for the nodes at a given level. This traversal results in an improved performance. Note that certain nodes may be occluded at a level and are not tested for visibility. After that we query the results and compute the visibility of each node. Let L_i be the list of nodes at level i which are being tested for visibility as well as pixel-deviation error. We generate the list L_{i+1} that will be tested at level $i+1$ by pushing the children of a node $n \in L_i$ only if its bounding box is visible, and it does not satisfy the pixel-deviation error criterion. We use an occlusion identifier for each node in the scene graph and exploit the parallelism available in GL_NV_occlusion_query by performing multiple occlusion queries at each level.

4.7.2 Visibility for LOD Selection

The LODs in a scene graph are associated with a screen space projection error. We traverse the scene graph until each LOD satisfies the pixels-of-error metric. However, this approach can be too conservative if the object is mostly occluded. We therefore utilize the visibility information in selecting an appropriate LOD or HLOD of the object.

The number of visible pixels for a bounding box of a node provides an upper bound on the number of visible pixels for its geometry. The GL_NV_occlusion_query also returns the number of pixels visible when the geometry is rendered. We compute the visibility of a node by rendering the bounding box of the node and the query returns the number of visible pixels corresponding to the box. If the number of visible pixels is less than the pixels-of-error specified by a bound, we do not traverse the scene graph any further at that node. This additional optimization is very useful if only a very small portion of the bounding box is visible, and the node has a very high screen space projection error associated with it.

4.8 Design Issues

Latency and reliability are two key components considered in the design of our overall rendering system. In addition to one frame of latency introduced by an occlusion-switch, our algorithm introduces additional latency due to the transfer of camera parameters and visible node identifiers across the network. We also require reliable transfer of data among different GPUs to ensure the correctness of our approach.

4.8.1 System Latency

A key component of any parallel algorithm implemented using a cluster of PCs is the network latency introduced in terms of transmitting the results from one PC to another during each frame. The performance of our system is dependent on the latency involved in receiving the camera parameters by the GPUs involved in occlusion-switch. In addition, there is latency in terms of sending the camera parameters from the GPU performing HC to the GPU performing RVG. Moreover, latency is also introduced in sending the visible nodes across the network to RVG and OR. We eliminate the latency problem in receiving the camera parameters by the GPU performing HC using the switching mechanism.

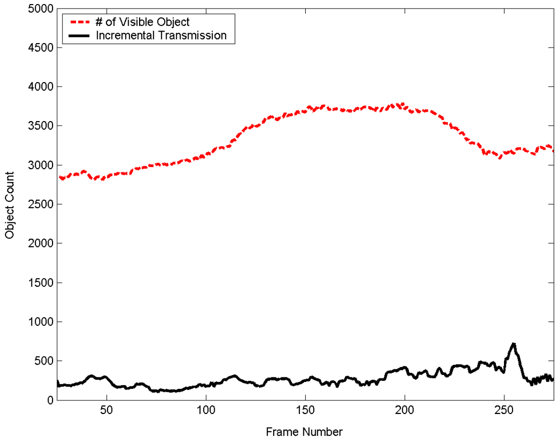


Figure 3: *Comparison of number of nodes transmitted with and without incremental transmission (described in section 4.6) for a sample path on Double Eagle Tanker model. Using incremental transmission, we observe an average reduction of 93% in the number of nodes transmitted between the GPUs.*

Let GPU_1 and GPU_2 constitute an occlusion-switch. GPU_1 performs HC for frame i and GPU_2 generates OR for frame $i+1$. For frame $i+1$, GPU_1 generates OR for frame $i+2$, and GPU_2 performs HC for frame $i+1$. Given that GPU_2 has already rendered the occluders for frame $i+1$, it already has the correct camera parameters for performing HC for frame $i+1$. As a result, no additional latency occurs in terms of HC receiving the camera parameters. However, the GPU performing OR requires the camera-parameters from the GPU performing RVG. This introduces latency in terms of receiving the camera parameters. Because HC takes some time to perform hardware cull tests before transmitting the first visible node to GPU performing OR, this latency is usually hidden. We reduce the latency in transmitting camera parameters from HC to RVG by sending them in the beginning of each frame. Figure 2 illustrates the basic protocol for transferring the camera parameters among the three GPU's. We enumerate other sources of network latency in Section 5.2.

4.8.2 Reliability

The correctness and conservativity of our algorithm depends on the reliable transmission of camera parameters and the visible nodes between the GPUs. Our system is synchronized based on transmission of an end of frame (EOF) packet. This protocol requires us to have reliable transmission of camera parameters from GPU performing HC to GPU performing RVG. Also, we require reliable transmission of node-ids and EOF from GPU performing HC to each GPU performing OR and RVG. We used reliable transfer protocols (TCP/IP) to transfer the data across the network.

5 Implementation and Performance

We have implemented our parallel occlusion culling algorithm on a cluster of three 2.2 GHz Pentium-4 PCs, each having 4 GB of RAM (on an Intel 860 chipset) and a GeForce 4 Ti 4600 graphics card. Each runs Linux 2.4, with bigmem option enabled giving 3.0 GB user addressable memory. The

Model	Pixels of Error	Average FPS		
		SWITCH	Distributed GigaWalk	GigaWalk
PP	5	14.17	6.2	5.6
DE	20	10.31	4.85	3.50
B-777	15	13.01	5.82	

Table 1: *Average frame rates obtained by different acceleration techniques over the sample path.* **FPS** *= Frames Per Second,* **DE** *= Double Eagle Tanker model,* **PP** *= Power Plant model,* **B-777** *= Boeing 777 model*

Model	Pixels of Error	Number of Polygons		
		SWITCH	GigaWalk	Exact Visibility
PP	5	91550	119240	7500
DE	20	141630	173350	10890

Table 2: *Comparison of number of polygons rendered to the actual number of visible polygons by the two implementations.* **DE** *= Double Eagle Tanker model,* **PP** *= Power Plant model*

PCs are connected via 100 Mb/s Ethernet. We typically obtain a throughput of $1-2$ million triangles per second in immediate mode using triangle strips on these graphics cards. Using NVIDIA OpenGL extension GL_NV_occlusion_query, we perform an average of around $50,000$ queries per second.

The scene database is replicated on each PC. Communication of camera parameters and visible node ids between each pair of PCs is handled by a separate TCP/IP stream socket over Ethernet. Synchronization between the PCs is maintained by sending a sentinel node over the node sockets to mark an end of frame(EOF).

We compare the performance of the implementation of our algorithm (called SWITCH) with the following algorithms and implementations:

- **GigaWalk**: A fast parallel occlusion culling system which uses two IR2 graphics pipelines and three CPUs [Baxter et al. 2002]. OR and RVG are performed in parallel on two separate graphics pipelines while occlusion culling is performed in parallel using a software based hierarchical Z-buffer. All the interprocess communication is handled using the shared memory.

- **Distributed GigaWalk**: We have implemented a distributed version of GigaWalk on two PCs with NVIDIA GeForce 4 GPUs. One of the PCs serves as the occlusion server implementing OR and occlusion culling in parallel. The other PC is used as a display client. The occlusion culling is performed in software similar to GigaWalk. Interprocess communication between PCs is based on TCP/IP stream sockets.

We compared the performance of the three systems on three complex environments: a coal fired Power Plant (shown in the color plate) composed of 13 million polygons and 1200 objects, a Double Eagle Tanker (shown in the color plate) composed of 82 million polygons and $127K$ objects, and part of a Boeing 777 (shown in the color plate) composed of 20 million triangles and $52K$ objects. Figures 4, 5(a) and 5(b) illustrate the performance of SWITCH on a complex path in the Boeing 777, Double Eagle and Powerplant models, respectively (as shown in the video). Notice that we are able to obtain $2-3$ times speedups over earlier systems.

We have also compared the performance of occlusion culling algorithm in terms of the number of objects and polygons rendered as compared to the number of objects and polygons exactly visible. *Exact visibility* is defined as the number of primitives actually visible up to the screen-space and depth-buffer resolution from a given viewpoint. The exact visibility is computed by drawing each primitive in a different color to an "itembuffer" and counting the number of colors visible. Figures 6(a) and 6(b) show the culling performance of our algorithm on the Double Eagle Tanker model.

Model	Pixels of Error	Number of Objects		
		SWITCH	GigaWalk	Exact Visibility
PP	5	1557	2727	850
DE	20	3313	4036	1833

Table 3: *Comparison of number of objects rendered to the actual number of visible objects by the two implementations.* **DE** *= Double Eagle Tanker model,* **PP** *= Power Plant model*

The average speedup in frame rate for the sample paths is shown in Table 1. Tables 2 and 3 summarize the comparison of the primitives rendered by SWITCH and GigaWalk with the exact visibility for polygons and objects respectively. As the scene graph of the model is organized in terms of objects and we perform visibility tests at an object level and not at the polygon level. Consequently, we observe a discrepancy in the ratios of number of primitives rendered to the exact visibility for objects and polygons.

5.1 Bandwidth Estimates

In our experiments, we have observed that the number of visible objects n typically ranges in the order of 100 to 4000 depending upon scene complexity and the viewpoint. If we render at most 30 frames per second (fps), header size h (for TCP, IP and ethernet frame) is 50 bytes and buffer size b is 100 nodes per packet, then we require a maximum bandwidth of 8.3 Mbps. Hence, our system is not limited by the available bandwidth on fast ethernet. However, the variable window size buffering in TCP/IP [Jacobson 1988], introduces network latency. The incremental transmission algorithm greatly lowers the communication overhead between different GPUs. Figure 3 shows the number of node identifiers transmitted with and without incremental transmission for a sample path in the Double Eagle Tanker model. We observe a very high frame-to-frame coherence and an average reduction of 93% in the bandwidth requirements. During each frame, the GPUs need to transmit pointers to a few hundred nodes, which adds up to a few kilobytes. The overall bandwidth requirement is typically a few megabytes per second.

5.2 Performance Analysis

In this section, we analyze different factors that affect the performance of occlusion-switch based culling algorithm. One of the key issues in the design of any distributed rendering algorithm is system latency. In our architecture, we may experience latency due to one of the following reasons:

1. **Network** : Network latencies mainly depend upon the implementation of transport protocol used to communicate between the PCs. The effective bandwidth varies depending on the packet size. Implementations like TCP/IP inherently buffer the data and may introduce latencies. Transmission of a large number of small size packets per second can cause packet loss and re-transmission introduces further delays. Buffering of node ids reduces loss but increases network latency.

2. **Hardware Cull** : The occlusion query can use only a limited number of identifiers before the results of pixel count are queried. Moreover, rendering a bounding box usually requires more resources in terms of fill-rate as compared to rasterizing the original primitives. If the application is fill-limited, HC can become a bottleneck

in the system. In our current implementation, we have observed that the latency in HC is smaller as compared to the network latency. Using a front based ordered culling, as described in section 4.6, reduces the fill-requirement involved in performing the queries and results in a better performance.

3. **OR and RVG** : OR and RVG can become bottlenecks when the number of visible primitives in a given frame is very high. In our current implementation, HC performs culling at the object level. As a result, the total number of polygons rendered by OR or RVG can be quite high depending upon the complexity of the model, the LOD error threshold and the position of the viewer. We can reduce this number by selecting a higher threshold for the LOD error.

The overall performance of algorithm is governed by two factors: culling efficiency for occlusion culling and the overall frame-rates achieved by the rendering algorithm.

- **Culling Efficiency**: Culling efficiency is measured in terms of the ratio of number of primitives in the potential visible set to the number of primitives visible. The culling efficiency of occlusion-switch depends upon the occlusion-representation used to perform culling. A good selection of occluders is crucial to the performance of HC. The choice of bounding geometric representation used to determine the visibility of an object affects the culling efficiency of HC. In our current implementation, we have used rectangular bounding box as the bounding volume because of its simplicity. As HC is completely GPU-based, we can use any other bounding volume (e.g. a convex polytope, k-dop) and the performance of the query will depend on the number of triangles used to represent the boundary of the bounding volume.

- **Frame Rate**: Frame rate depends on the culling efficiency, load balancing between different GPUs and the network latency. Higher culling efficiency results in OR and RVG rendering fewer number of primitives. A good load balance between the occlusion-switch and the RVG would result in maximum system throughput. The order and the rate at which occlusion tests are performed affects the load balance across the GPUs. Moreover, the network latency also affects the overall frame rate. The frame rate also varies based on the LOD selection parameter.

With faster GPUs, we would expect higher culling efficiency as well as improved frame rates.

5.3 Comparison with Earlier Approaches

We compare the performance of our approach with two other well-known occlusion culling algorithms: HZB [Greene et al. 1993] and HOM [Zhang et al. 1997]. Both of these approaches use a combination of object-space and image-space hierarchies and are conservative to the image precision. Their current implementations are based on frame-buffer readbacks and performing the occlusion tests in software. The software implementation incurs additional overhead in terms of hierarchy construction. Moreover, they project the object's bounding volume to the screen space and compute a 2D screen space bounding rectangle to perform the occlusion test. As a result, these approaches are more conservative as compared to occlusion-switch based culling algorithm. Further, the frame-buffer or depth-buffer readbacks can be expensive as compared to the occlusion queries, especially on current PC systems. In practice, we obtained almost three times speed-up over an implementation of HZB on two PCs (Distributed GigaWalk).

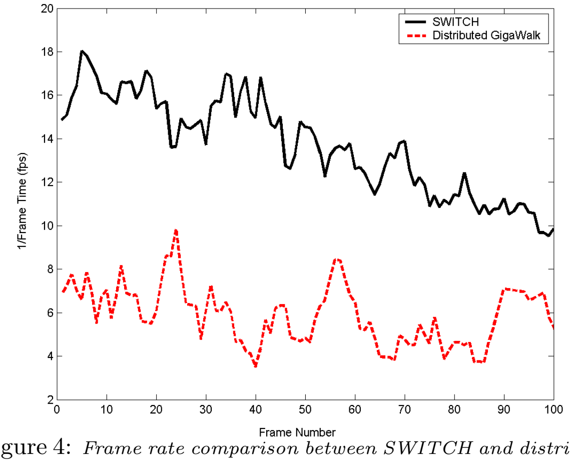


Figure 4: *Frame rate comparison between SWITCH and distributed Gigawalk at* 1024 × 1024 *screen resolution and* 15 *pixels of error on Boeing model.*

Our algorithm also utilizes the number of visible pixels parameter returned by GL_NV_occlusion_query for LOD selection. This bound makes our rendering algorithm less conservative as compared to earlier LOD-based rendering algorithms, which only compute a screen space bound from the object space deviation error.

5.4 Limitations

Occlusion-switch based culling introduces an extra frame of latency in addition to double-buffering. The additional latency does not decrease the frame rate as the second pass is performed in parallel. However, it introduces additional latency into the system; the overall algorithm is best suited for latency-tolerant applications. In addition, a distributed implementation of the algorithm may suffer from network delays, depending upon the implementation of network transmission protocol used. Our overall approach is general and independent of the underlying networking protocol.

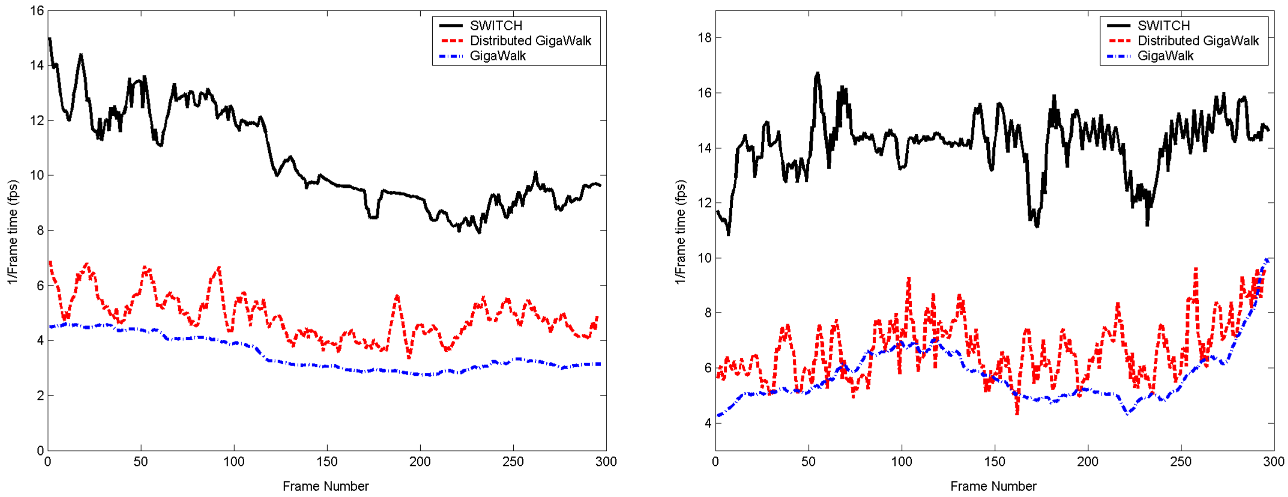
Our occlusion culling algorithm also assumes high spatial coherence between successive frames. If the camera position changes significantly from one frame to the next, the visible primitives from the previous frame may not be a good approximation to the occluder set for the current frame. As a result, the culling efficiency may not be high.

Our algorithm performs culling at an object level and does not check the visibility of each triangle. As a result, its performance can vary based on how the objects are defined and represented in the scene graph.

6 Summary and Future Work

We have presented a new occlusion culling algorithm based on occlusion-switches and used it to render massive models at interactive rates. The occlusion-switches, composed of two GPUs, make use of the hardware occlusion query that is becoming widely available on commodity GPUs. We have combined the algorithm with pre-computed levels-of-detail and highlighted its performance on three complex environments. We have observed $2-3$ times improvement in frame rate over earlier approaches. The culling performance of the algorithm is further improved by using a sub-object hierarchy and it is used for interactive shadow generation [Govindaraju et al. 2003].

Many avenues for future work lie ahead. A low latency network implementation is highly desirable to maximize the performance achieved by our parallel occlusion culling algorithm. One possibility is to use raw GM sockets over



(a) Double Eagle Tanker model at 20 pixels of error

(b) Powerplant model at 5 pixels of error

Figure 5: *Frame rate comparison between SWITCH, GigaWalk and Distributed GigaWalk at* 1024×1024 *screen resolution. We obtain* $2-3$ *times improvement in the frame rate as compared to Distributed GigaWalk and GigaWalk.*

Myrinet. We are also exploring the use of a reliable protocol over UDP/IP. Our current implementation loads the entire scene graph and object LODs on each PC. Due to limitations on the main memory, we would like to use out-of-core techniques that use a limited memory footprint. Moreover, the use of static LODs and HLODs can lead to popping artifacts as the rendering algorithm switches between different approximations. One possibility is to use view-dependent simplification techniques to alleviate these artifacts. Finally, we would like to apply our algorithm to other complex environments.

Acknowledgments

Our work was supported in part by ARO Contract DAAD19-99-1-0162, NSF award ACI 9876914, ONR Young Investigator Award (N00014-97-1-0631), a DOE ASCI grant, and by Intel Corporation.

The Double Eagle model is courtesy of Rob Lisle, Bryan Marz, and Jack Kanakaris at NNS. The Power Plant environment is courtesy of an anonymous donor. The Boeing 777 model is courtesy of Tom Currie, Bill McGarry, Marie Murray, Nik Prazak, and Ty Runnels at the Boeing Company. We would like to thank David McAllister, Carl Erikson, Brian Salomon and other members of UNC Walkthrough group for useful discussions and support.

References

Airey, J., Rohlf, J., and Brooks, F. 1990. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, 41–50.

Bartz, D., Meibner, M., and Huttner, T. 1999. Opengl assisted occlusion culling for large polygonal models. *Computer and Graphics 23*, 3, 667–679.

Baxter, B., Sud, A., Govindaraju, N., and Manocha, D. 2002. Gigawalk: Interactive walkthrough of complex 3d environments. *Proc. of Eurographics Workshop on Rendering*.

Cohen-Or, D., Chrysanthou, Y., and Silva, C. 2001. A survey of visibility for walkthrough applications. *SIGGRAPH Course Notes # 30*.

Coorg, S., and Teller, S. 1997. Real-time occlusion culling for models with large occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*.

Durand, F., Drettakis, G., Thollot, J., and Puech, C. 2000. Conservative visibility preprocessing using extended projections. *Proc. of ACM SIGGRAPH*, 239–248.

El-Sana, J., Sokolovsky, N., and Silva, C. 2001. Integrating occlusion culling with view-dependent rendering. *Proc. of IEEE Visualization*.

Erikson, C., Manocha, D., and Baxter, B. 2001. Hlods for fast display of large static and dynmaic environments. *Proc. of ACM Symposium on Interactive 3D Graphics*.

Govindaraju, N., Lloyd, B., Yoon, S., Sud, A., and Manocha, D. 2003. Interactive shadow generation in complex environments. Tech. rep., Department of Computer Science, University of North Carolina.

Greene, N., Kass, M., and Miller, G. 1993. Hierarchical z-buffer visibility. In *Proc. of ACM SIGGRAPH*, 231–238.

Greene, N. 2001. Occlusion culling with optimized hierarchical z-buffering. In *ACM SIGGRAPH COURSE NOTES ON VISIBILITY, # 30*.

Hillesl, K., Salomon, B., Lastra, A., and Manocha, D. 2002. Fast and simple occlusion culling using hardware-based depth queries. Tech. Rep. TR02-039, Department of Computer Science, University of North Carolina.

Hudson, T., Manocha, D., Cohen, J., Lin, M., Hoff, K., and Zhang, H. 1997. Accelerated occlusion culling using shadow frusta. In *Proc. of ACM Symposium on Computational Geometry*, 1–10.

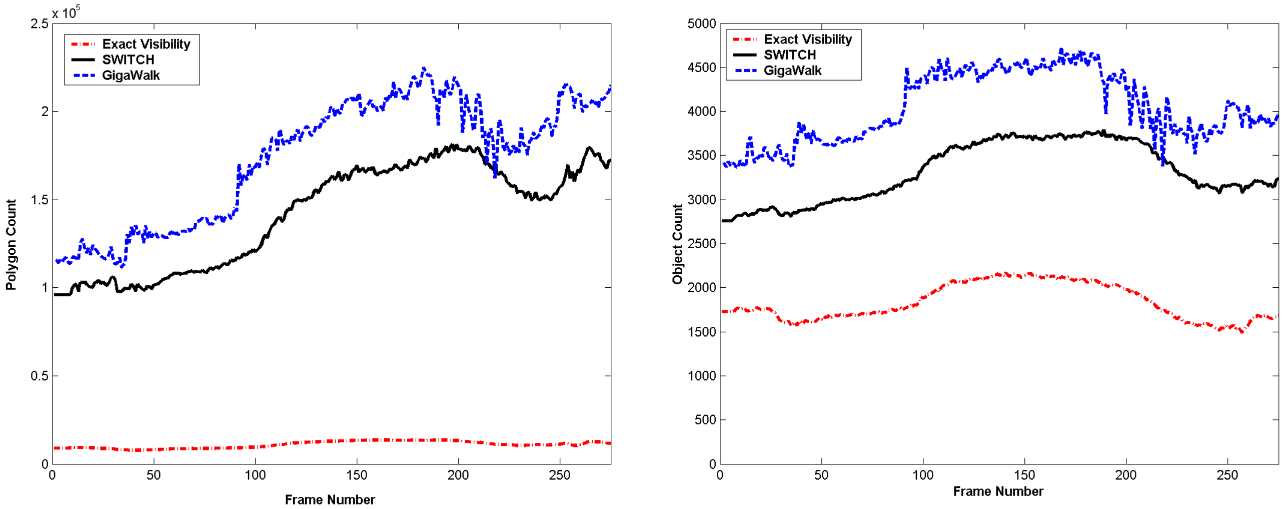
Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., and Hanrahan, P. 2001. Wiregl: A scalable graphics system for clusters. *Proc. of ACM SIGGRAPH*.

Jacobson, V. 1988. Congestion avoidance and control. *Proc. of ACM SIGCOMM*, 314–329.

Klowoski, J., and Silva, C. 2000. The prioritized-layered projection algorithm for visible set estimation. *IEEE Trans. on Visualization and Computer Graphics 6*, 2, 108–123.

Klowoski, J., and Silva, C. 2001. Efficient conservative visiblity culling using the prioritized-layered projection algorithm. *IEEE Trans. on Visualization and Computer Graphics 7*, 4, 365–379.

Meissner, M., Bartz, D., Huttner, T., Muller, G., and Einighammer, J. 2002. Generation of subdivision hierarchies for efficient occlusion culling of large polygonal models. *Computer and Graphics*.



(a) At polygon level

(b) At object level

Figure 6: *Double Eagle Tanker: Comparison of exact visibility computation with SWITCH and GigaWalk at* 20 *pixels of error at* 1024 × 1024 *screen resolution. SWITCH is able to perform more culling as compared to GigaWalk. However, it renders one order of magnitude more triangles or twice the number of objects as compared to exact visibility.*

Parker, S., Martic, W., Sloan, P., Shirley, P., Smits, B., and Hansen, C. 1999. Interactive ray tracing. *Symposium on Interactive 3D Graphics*.

Samanta, R., Funkhouser, T., Li, K., and Singh, J. P. 2000. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. *Eurographics/SIGGRAPH workshop on Graphics Hardware*, 99–108.

Samanta, R., Funkhouser, T., and Li, K. 2001. Parallel rendering with k-way replication. *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*.

Schaufler, G., Dorsey, J., Decoret, X., and Sillion, F. 2000. Conservative volumetric visibility with occluder fusion. *Proc. of ACM SIGGRAPH*, 229–238.

Teller, S. J. 1992. *Visibility Computations in Densely Occluded Polyheral Environments*. PhD thesis, CS Division, UC Berkeley.

Wald, I., Slusallek, P., and Benthin, C. 2001. Interactive distributed ray-tracing of highly complex models. In *Rendering Techniques*, 274–285.

Wonka, P., Wimmer, M., and Schmalstieg, D. 2000. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques*, 71–82.

Wonka, P., Wimmer, M., and Sillion, F. 2001. Instant visibility. In *Proc. of Eurographics*.

Zhang, H., Manocha, D., Hudson, T., and Hoff, K. 1997. Visibility culling using hierarchical occlusion maps. *Proc. of ACM SIGGRAPH*.

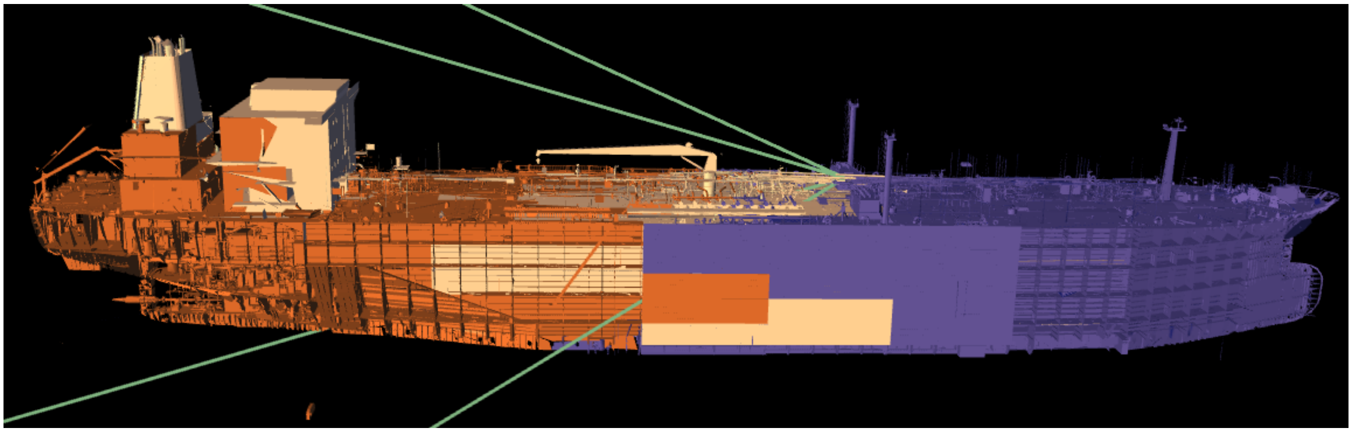
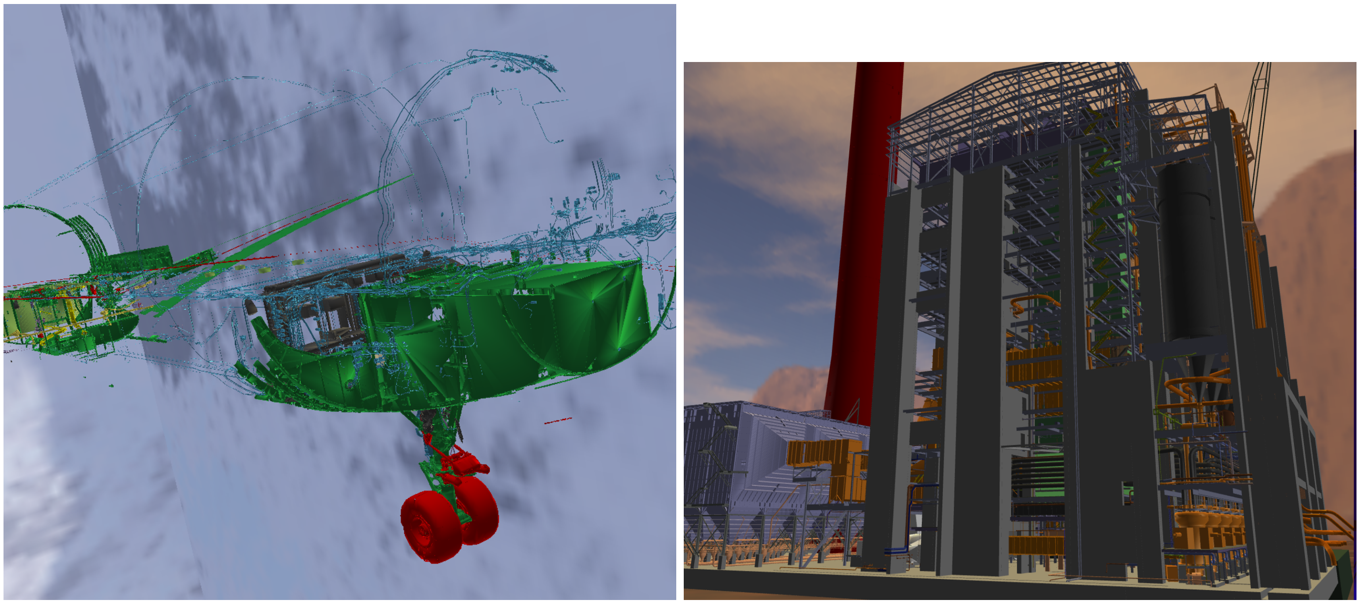


Figure 1: *Performance of occlusion-switch algorithm on the DoubleEagle Tanker model: This environment consists of more than* 82 *million triangles and our algorithm renders it a t* $9 - 15$ *fps on a cluster of 3 PCs, each consisting of an NVIDIA GeForce 4 GPU. Occlusion-switch culls away most occluded portions of the model and renders around* 200K *polygons in the view shown. Objects are rendered in following colors - visible: yellow; view-frustum culled: violet; and occlusion-culled: orange.*



(a) *Portion of a Boeing* 777 *model rendered at* 15 *pixels of error. Our system, SWITCH, is able to render it at* $11 - 18$ *frames per second on a 3-PC cluster.*

(b) *Powerplant model composed of more than* 12.7 *million triangles. SWITCH can render it at* $11 - 19$ *frames per second using* 5 *pixels of deviation error.*

Figure 2: *Performance of Occlusion-switch on complex CAD models: Both the models are rendered at* 1024×1024 *screen resolution using NVIDIA GeForce 4 cards.*

Thirteenth Eurographics Workshop on Rendering (2002)
P. Debevec and S. Gibson (Editors)

GigaWalk: Interactive Walkthrough of Complex Environments

William V. Baxter III Avneesh Sud Naga K. Govindaraju Dinesh Manocha

University of North Carolina at Chapel Hill
{baxter,sud,naga,dm}@cs.unc.edu
http://gamma.cs.unc.edu/GigaWalk

Abstract

We present a new parallel algorithm and a system, GigaWalk, for interactive walkthrough of complex, gigabyte-sized environments. Our approach combines occlusion culling and levels-of-detail and uses two graphics pipelines with one or more processors. GigaWalk uses a unified scene graph representation for multiple acceleration techniques, and performs spatial clustering of geometry, conservative occlusion culling, and load-balancing between graphics pipelines and processors. GigaWalk has been used to render CAD environments composed of tens of millions of polygons at interactive rates on systems consisting of two graphics pipelines. Overall, our system's combination of levels-of-detail and occlusion culling techniques results in significant improvements in frame-rate over view-frustum culling or either single technique alone.

Keywords: Interactive display systems, parallel rendering, occlusion culling, levels-of-detail, Engineering Visualization.

1. Introduction

Users of computer-aided design and virtual reality applications often create and use geometric models of large, complex 3D environments. Gigabyte-sized datasets representing power plants, ships, airplanes, submarines and urban scenes are not uncommon. Simulation-based design and design review of such datasets benefits significantly from the ability to generate user-steered interactive displays or *walkthroughs* of these environments. Yet, rendering these environments at interactive rates and with high fidelity has been a major challenge.

Many acceleration techniques for interactive display of complex datasets have been developed. These include visibility culling, object simplification and the use of image-based or sampled representations. They have been successfully combined to render certain specific types of datasets at interactive rates, including architectural models [15], terrain datasets [25], scanned models [33] and urban environments [42]. However, there has been less success in displaying more general complex datasets due to several challenges facing existing techniques:

Occlusion Culling: While possible for certain environments, performing exact visibility computations on large, general datasets is difficult to achieve in real time on current graphics systems [11]. Furthermore, occlusion culling alone will not sufficiently reduce the load on the graphics pipeline when many primitives are actually visible.

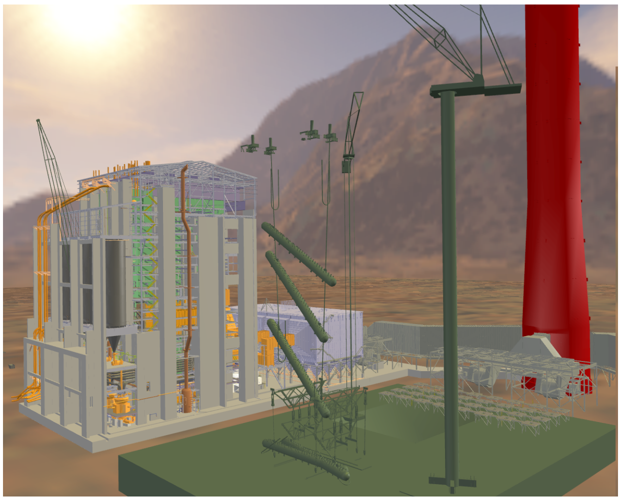


Figure 1: Coal-Fired Power plant: This 1.7 gigabyte environment consists of over 13 million triangles and 1200 objects. GigaWalk can display it 12-37 frames per second on an SGI Onyx workstation using two IR2 graphics pipelines and three 300MHz R12000 CPUs.

Object Simplification: Object simplification techniques alone have difficulty with high-depth-complexity scenes, as they do not address the problems of overdraw and fill load on the graphics pipeline.

Image-based Representations: There are some promising image-based algorithms, but generating complete samplings of large complex environments automatically and efficiently remains a difficult problem. The use of image-based methods can also lead to popping and aliasing artifacts.

1.1. Main Results

We present a parallel architecture that enables interactive rendering of complex environments comprised of many tens of millions of polygons. Initially, we precompute geometric levels-of-detail (LODs) and represent the dataset using a scene graph. Then at runtime we compute a *potentially visible set* (PVS) of geometry for each frame using a combination of view frustum culling and a two-pass hierarchical Z-buffer occlusion culling algorithm [19] in conjunction with the pre-computed LODs. The system runs on two graphics rasterization pipelines and one or more CPU processors. Key features of our approach include:

1. A parallel rendering algorithm that is general and automatic, makes few assumptions about the model, and places no restrictions on user motion through the scene.
2. A unified scene graph hierarchy that is used for both geometric simplification and occlusion culling.
3. A parallel, image-precision occlusion culling algorithm based on the hierarchical Z-buffer [19, 20]. It uses *hierarchical occluders* and can perform conservative as well as approximate occlusion culling.
4. A parallel rendering algorithm that balances the computational load between two rendering pipelines and one or more processors.
5. An interactive system, GigaWalk, to render large, complex environments with good fidelity on two-pipeline graphics systems. The graphics pipelines themselves require only standard rasterization capabilities.

We demonstrate the performance of our system on two complex CAD environments: a coal-fired power plant (Fig. 1) composed of 13 million triangles, and a Double Eagle Tanker (Plate 1) composed of over 82 million triangles. GigaWalk is able to render models such as these at $11-50$ frames a second with little loss in image quality on an SGI Onyx workstation using two IR2 pipelines. The end-to-end latency of this implementation is typically $50-150$ milliseconds. We have also developed a preliminary implementation of GigaWalk on a pair of networked PCs.

1.2. Organization

The rest of the paper is organized as follows. We give a brief survey of previous work in Section 2. Section 3 gives an overview of our approach. In Section 4 we describe the scene representation and preprocessing steps. Section 5 presents the parallel algorithm for interactive display. We describe the system implementation and highlight its performance on complex models in Section 6.

2. Prior Work

In this section, we present a brief overview of previous research on interactive rendering of large datasets, including geometric simplification and occlusion culling algorithms, and other systems that have combined multiple rendering acceleration techniques.

2.1. Geometric Simplification

Simplification algorithms compute a reduced-polygon approximation of a model while attempting to retain the shape of the original. A recent survey of simplification algorithms is presented in [30].

Algorithms for simplifying large environments can be classified as either static (view-independent) or dynamic (view-dependent). Static approaches pre-compute a discrete series of levels-of-detail (LODs) in a view-independent manner [9, 17, 32, 35]. Erikson et al. [10] presented an approach to large model rendering based on the hierarchical use of static LODs, or HLODs. We also use LODs and HLODs in our system.

At run-time, rendering algorithms for static LODs choose an appropriate LOD to represent each object based on the viewpoint. Selecting the LODs requires little run-time computation, and rendering static LODs on contemporary graphics hardware is also efficient.

View-dependent, dynamic algorithms pre-compute a data structure that encodes a continuous range of detail. Examples include progressive meshes [23, 24, 44] and hierarchies of decimation operations [29, 12]. Selection of the appropriate LOD is based on view-parameters such as illumination and viewing position. Overall, view-dependent LODs can provide better fidelity than static LODs and work well for large connected datasets such as terrain and spatially large objects. However, the run-time overhead is higher compared to static LODs, since all level-of-detail selection is done at the individual feature level (vertex, edge, polygon), rather than the object level.

2.2. Occlusion Culling

Occlusion culling methods attempt to quickly determine a PVS for a viewpoint by excluding geometry that is occluded. A recent survey of different algorithms is presented in [6].

Several effective algorithms have been developed for specific environments. Examples include cells and portals for architectural models [2, 39] and algorithms for urban datasets or scenes with large, convex occluders [7, 22, 36, 41, 42]. In this section, we restrict the discussion to occlusion culling algorithms for general environments.

Algorithms for occlusion culling can be broadly classified based on whether they are conservative or approximate, whether they use object space or image space hierarchies, and whether they compute visibility from a point (*from-point*) or a region (*from-region*). Conservative algorithms compute a PVS that includes all the visible primitives, plus a small number of potentially occluded primitives [7, 19, 22, 28, 45]. The approximate algorithms identify most of the visible objects but may incorrectly cull some objects [5, 27, 45].

Object space algorithms can perform culling efficiently and accurately given a small set of large occluders, but it is difficult to perform the “occluder fusion" necessary to effectively cull in scenes composed of many small occluders. For these types of scenes, the image space algorithms typified by the hierarchical Z-buffer (HZB) [19, 20] or hierarchical occlusion maps (HOM) [45] are more effective.

From-region algorithms pre-compute a PVS for each region of space to reduce the run-time overhead [8, 36, 41]. This works well for scenes with large occluders, but the amount of geometry culled by a given occluder diminishes as the region sizes are increased. Thus there is a trade-off between the quality of the PVS estimation for each region and the memory overhead. These algorithms may be overly conservative and have difficulty obtaining significant culling in scenes including only small occluders. In contrast, from-point algorithms generally provide more accurate culling, but they have a higher run-time cost.

2.3. Parallel Approaches

A number of parallel approaches based on multiple graphics pipelines have been proposed. These can provide scalable rendering on shared-memory systems or clusters of PCs. These approaches can by classified mainly as either object-parallel, screen-space-parallel, or frame-parallel [21, 37]. Specific examples include distributing primitives to different pipelines by the screen region into which they fall (screen-space-parallel), or rendering only every Nth frame on each pipeline (frame-parallel).

Another parallel approach to large model rendering that shows promise is interactive ray tracing [4, 40]. The algorithm described in [40] is able to render the Power Plant model at 4-5 frames a second with 640×480 pixel resolution on a cluster of seven dual processor PCs.

Garlick et al. [16] presented a system for performing view-frustum culling on multiple CPUs in parallel with the rendering process. Their observation that culling can be performed in parallel to improve overall system performance is the fundamental concept behind our approach as well.

Wonka et al. [42] presented a “visibility server” that performed occlusion culling to compute a PVS at run-time in parallel on a separate machine. Their system works well for urban environments; however, it relies on the *occluder shrinking* algorithm [41] to compute the region-based visibility. This approach is effective only if the occluders are large and volumetric in nature.

2.4. Hybrid Approaches

The literature reports several systems that combine multiple techniques to accelerate the rendering of large models. For example, The BRUSH system [34] used LODs with hierarchical representation for large mechanical and architectural models. The UC Berkeley Architecture Walkthrough system [15] combined hierarchical algorithms with object-space visibility computations [39] and LODs for architectural models.

More recently, Anjudar et al. [3] presented a framework that integrates occlusion culling and LODs. The crux of the approach is to estimate the degree of visibility of each object in the PVS and use that value both to select appropriate LODs and to cull. The method relies on decomposing scene objects into overlapping convex pieces (axis-aligned boxes) that then serve as individual “synthetic occluders”. Thus the effective maximum occluder size depends on the largest axis-aligned box that will fit inside each object.

Another recent integrated approach uses a prioritized-layered projection visibility approximation algorithm with view-dependent rendering [11]. The resulting rendering algorithm seems a promising approach when approximate (non-conservative) visibility is acceptable.

The UNC Massive Model Rendering (MMR) system [1] combined LODs with image-based impostors and occlusion culling to deliver interactive walkthroughs of complex models. A more detailed comparison with this system will be made later in Section 6.5.

Various proprietary systems exist as well, such as the one Boeing created in the 1990’s to visualize models of large passenger jets. However, to the best of our knowledge, no detailed descriptions of this system are available, so it is difficult to make comparisons.

3. Overview

In this section, we give a brief overview of the main components of our approach. These components are simplification, occlusion culling, and a parallel architecture.

3.1. Model Simplification

Given a large environment, we generate a scene graph by clustering small objects, and partitioning large objects to create a spatially coherent, axis-aligned bounding box (AABB) hierarchy. The hierarchy construction will be discussed in more detail in Section 4.

3.2. Parallel Occlusion Culling

At run-time, our algorithm performs occlusion culling, in addition to view frustum culling, based on the pre-computed AABB scene graph hierarchy. We use a two-pass version of the hierarchical Z-buffer algorithm [19] with a two-graphics-pipeline parallel architecture. In this architecture, occluders are rendered on one pipeline while the final interactive rendering of visible primitives takes place on the second pipeline. A separate software thread performs the actual culling using the Z-buffer that results from the occluder rendering. The architecture will be presented in detail in Section 5.

We chose to use the hierarchical Z-buffer (HZB) because of its good culling performance, minimal restrictions on the type of occluders, and for its ability to perform occluder fusion. Moreover, it can be made to work well without extra preprocessing or storage overhead by exploiting temporal coherence. The preprocessing and storage cost of GigaWalk is thus the same as that of an LOD-only system.

Occluder Selection: A key component of any occlusion culling algorithm is occluder selection, which can be accomplished in a number of ways. A typical approach uses solid angles and spatial distributions of objects to estimate a small set of good occluders [45, 27]. However, occluders selected according to such heuristics are not necessarily optimal in terms of the number of other objects they actually occlude. The likelihood of obtaining good occlusion can be increased by making the occluder set larger, but computational costs usually demand the set be as small as possible.

Our parallel approach, on the other hand, allows us to take advantage of the temporal coherence based occluder selection algorithm presented by Greene et al.[19], which treats *all* the visible geometry from the previous frame as occluders for the current frame. This method makes use of frame-to-frame coherence and provides a good approximation to the foreground occluders for the current frame.

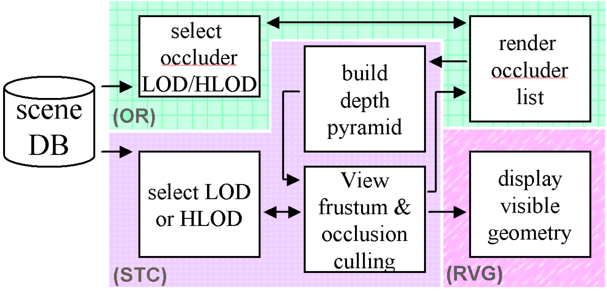


Figure 2: System Architecture: Each shaded region represents a separate process. The OR and RVG processes are associated with separate graphics pipelines, whereas the STC uses one or more processors.

3.3. GigaWalk Architecture

Fig. 2 presents the overall architecture of our run-time system. It shows the three processes that run in parallel:

1. **Occluder Rendering (OR):** Using all the visible geometry from a previous frame as the occluder set, this process renders that set into a depth buffer. It runs on the first graphics pipeline.
2. **Scene Traversal, Culling and LOD Selection (STC):** This process computes the HZB using the depth buffer computed by OR. It traverses the scene graph, computes the visible geometry and selects appropriate LODs based on the user-specified error tolerance. The visible geometry is used by RVG for the current frame and OR for the next frame. It runs on one or more processors.
3. **Rendering Visible Scene Geometry (RVG):** This process renders the visible scene geometry computed by STC. It uses the second graphics pipeline.

More details of the run-time system are given in Sections 5 and 6.

4. Scene Representation

In this section, we give an overview of our pre-processing algorithm used to compute a scene graph representation of the geometric environment.

CAD datasets often consist of a large number of objects which are organized according to a functional, rather than spatial, hierarchy. By “object” we mean simply the lowest level of organization in a model or model data structure above the primitive level. The size of objects can vary dramatically in CAD datasets. For example, in the Power Plant model a large pipe structure, which spans the entire model and consists of more than 6 million polygons, is one object. Similarly, a relatively small bolt with 20 polygons is another object. Our rendering algorithm computes LODs, selects them, and performs occlusion culling at the object level; therefore, the criteria used for organizing primitives into objects has a serious impact on the performance of the system. Our first step, then, is to redefine objects in a dataset based on criteria that will improve performance.

4.1. Unified Scene Hierarchy

Our rendering algorithm performs occlusion culling in two rendering passes: Pass 1 renders occluders to create a hierarchical Z-buffer to use for culling, Pass 2 renders the objects that are deemed visible by the HZB culling test. Given this two-pass approach, we could consider using a separate representation for occluders in Pass 1 than for displayed objects in Pass 2 [22, 45]. Using different representations has the advantage of allowing different criteria for partitioning and clustering of each hierarchy. Moreover, it gives us the flexibility to use an alternate error metric for creating simplified occluders, one optimized to preserve occlusion properties rather than visual fidelity.

Despite these potential advantages, we use a single, unified hierarchy for occlusion culling and LOD-based rendering. A single hierarchy offers several benefits. First, using the same representation decreases the storage overhead and the overall preprocessing cost. Second, it leads to a conservative occlusion culling algorithm. Our rendering algorithm treats the visible geometry from the previous frame as the occluder set for the current frame. In order to guarantee conservative occlusion culling, it is sufficient to ensure that exactly the same set of nodes and LODs in the unified scene graph are used by each process.

4.1.1. Criteria for Hierarchy

A good hierarchical representation of the scene graph is crucial for the performance of occlusion culling and the overall rendering algorithm. We use the same hierarchy for view frustum culling, occluder selection, occlusion tests on potential occludees, hierarchical simplification, and LOD selection. Though there has been considerable work on spatial partitioning and bounding volume hierarchies, including top-down and bottom-up strategies and spatial clustering, none of them seem to have addressed all the characteristics desired by our rendering algorithm. These include good spatial localization, object size, balance of the hierarchy, and minimal overlap between the bounding boxes of sibling nodes in the tree.

Bottom-up hierarchies lead to better localization and higher fidelity LODs. However, it is harder to use bottom-up techniques to compute hierarchies that are both balanced and have minimal spatial overlap between nodes. On the other hand, top-down schemes are better at ensuring balanced hierarchies and bounding boxes with little or no overlap between sibling nodes. Given their respective benefits, we use a hybrid approach that combines both top-down partitioning and hierarchy construction with bottom-up clustering.

4.2. Hierarchy Generation

In order to generate uniformly-sized objects, our preprocessing algorithm first redefines the objects using a combination of partitioning and clustering algorithms. The partitioning algorithm takes large objects and splits them into multiple objects. The clustering step groups objects with low polygon counts based on their spatial proximity. The combination of these steps seems to result in a redistribution of geometry with good localization and emulates some of the benefits of pure bottom-up hierarchy generation. The overall algorithm proceeds as follows:

1. Partition large objects into sub-objects in the initial database (top-down)
2. Organize disjoint objects and sub-objects into clusters (bottom-up)
3. Partition again to eliminate any uneven spatial clusters (top-down)
4. Compute an AABB bounding volume hierarchy on the final redefined set of objects (top-down).

The partitioning (stages 2 and 4) uses standard top-down techniques that group polygons based on an object's center or center-of-mass, along with several heuristics for selecting the split axis. The clustering algorithm (stage 3) was adapted from a computer vision technique for image segmentation [14]. The algorithm uses minimum spanning trees (MST) to represent clusters and is similar to *Kruskal's* algorithm [26]. Plate 2 shows the results of clustering and partitioning on the Power Plant and Double Eagle models. More details on the partitioning and clustering algorithm as well as hierarchy computation are given in [38].

4.3. HLOD Generation

Given the AABB-based scene graph representation, the algorithm computes a series of LODs for each node. The HLODs are computed in a bottom-up manner. The HLODs of the leaf nodes are the same as static LODs, while the HLODs of intermediate nodes are computed by combining the LODs of the nodes with the HLODs of node's children[10]. We use the GAPS [9] simplification algorithm, which can merge disjoint objects.

The majority of the pre-computation time is spent in LOD and HLOD generation. The HLODs of an internal node depend only on the LODs of the children, so by keeping only the LODs of the current node and its children in main memory, HLOD generation is accomplished within a small memory footprint. Specifically, the memory usage is given by

$$\begin{aligned} main\_memory\_footprint \leq\ & sizeof(AABBHierarchy) \\ & + \max_{N_i \in SG} (sizeof(N_i) + \\ & \sum_{C_j \in Child(N_i)} sizeof(C_j)), \end{aligned}$$

where SG corresponds to the scene graph.

4.4. HLODs as Hierarchical Occluders

Our occlusion culling algorithm uses LODs and HLODs of nodes as occluders to compute the HZB. They are selected based on the maximum screen-space pixel deviation error on object silhouettes.

The HLODs used by the rendering algorithm can also be regarded as "hierarchical occluders". A hierarchical occluder associated with a node N_i is an approximation of a group of occluders contained in the subtree rooted at N_i. The approximation provides a lower polygon count representation of a collection of object-space occluders. It can also be regarded as object-space occluder fusion.

5. Interactive Display

In this section, we present our parallel rendering architecture for interactive display of complex environments. Here we describe in detail the operations performed by each of the two graphics pipelines and each of the three processes: occluder rendering (OR), scene traversal and culling (STC) and rendering visible geometry (RVG), which run synchronously in parallel (as shown in Fig. 3).

5.1. Run-time Architecture

The relationship between different processes and the tasks performed by them is shown in Fig. 2.

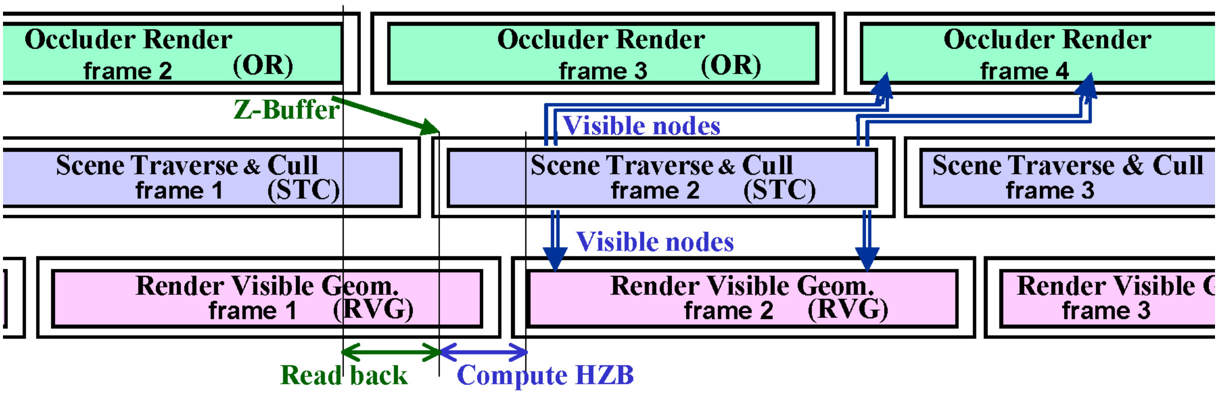


Figure 3: Timing relationship between different processes. The arrows indicate data passed between processes during the computation of frame 2. Along with the other data indicated, viewpoints also travel through the pipeline according to the frame numbers. This diagram demonstrates the use of occluders from frame $i-2$ rather than $i-1$ (see Section 5.2).

5.1.1. Occluder Rendering

The first stage for a given frame is to render the occluders. The occluders are simply the visible geometry from a previous frame. By using this temporal coherence strategy, the load on the two graphics pipelines is essentially balanced, since they render the exact same set of primitives, just shifted in time. The culling and LOD selection performed for displaying frame i results in an occluder set for frame $i+1$ that has manageable size. A brief pseudo-code description is given in Algorithm 5.1.

```
Occluder_Render(δ,frame i)
   • get current instantaneous camera position
     (camera i)
   • while (more nodes on node queue from STC (i-1))
       * pop next node off the queue
       * select LOD/HLOD for the node according to
           error tolerance, δ, using camera i
       * render that LOD/HLOD into Z-buffer i
   • read back Z-buffer i from graphics hardware
   • push Z-buffer i onto queue for STC i
   • push camera i onto queue for STC i
```

ALGORITHM 5.1: Occluder_Render.

Since the list of visible geometry for rendering comes from the culling stage (STC), and STC gets its input from this process (OR), a start-up procedure is required to initialize the pipeline and resolve this cyclic dependency. During startup, the OR stage is bypassed on the first frame, and STC generates its initial list of visible geometry without occlusion culling.

5.1.2. Scene Traversal, Culling and LOD Selection

The STC process first computes the HZB from the depth buffer output from OR. It then traverses the scene graph, performing view-frustum culling, occlusion culling and LOD error-based selection in a recursive manner. The LOD selection proceeds exactly as in [10, 43]: recursion terminates at nodes that are either culled, or which meet the user-specified pixel-error tolerance. A pseudo-code description is given in Algorithm 5.2. The occlusion culling is performed by comparing the bounding box of the node with the HZB. It can be performed in software or can make use of hardware-based queries as more culling extensions become available.

```
Scene_Traversal_Cull(ε, frame i)
   • get Z-buffer i from OR i via Z-buffer queue
   • build HZB i
   • get camera i from OR i queue
   • push copy of camera i onto queue for RVG i
   • set NodeList[i] = Root(SceneGraph)
   • while (NotEmpty(NodeList[i]))
       * node = First(NodeList[i])
       * set NodeList[i] = Delete(NodeList[i],node)
       * if (View_Frustum_culled(node)) then next;
       * if (Occlusion_Culled(node)) then next;
       * if HLOD_Error_Acceptable(ε,node) then
           – push node onto queue for OR $i+1$;
           – push node onto display queue for RVG i;
       * else
         set NodeList[i] = Add(NodeList[i],
                                   Children(node));
```

ALGORITHM 5.2: Scene_Traversal_Cull.

5.1.3. Rendering Visible Scene Geometry

All the culling is performed by STC, so the final render loop has only to rasterize the nodes from STC as they are placed in the queue. See Algorithm 5.3.

5.2. Occluder Selection

Ideally, the algorithm uses the visible geometry from the previous frame $(i-1)$ as the occluders for the current frame to get the best approximation to the current foreground geometry. However, using the previous frame's geometry can lead

```
Render_Visible_Scene_Geometry(frame i)
    • get camera i from STC i queue
    • while (more nodes on queue from STC i)
        • pop node off queue
        • render node
```

ALGORITHM 5.3: Render_Visible_Scene_Geometry.

to bubbles in the pipeline, because of the dependency between the OR and STC stages: STC must wait for OR to finish rendering the occluders before it can begin traversing the scene graph and culling. Fortunately, using the visible geometry from two frames previous can eliminate that dependency, and still provides a good approximation to the visible geometry for most interactive applications.

5.3. Trading Fidelity for Performance

The user can trade off fidelity for better performance in a number of ways. The primary control for achieving higher frame rates is the allowable LOD pixel error (see Plate 3).

Our system has been designed primarily to offer conservative occlusion culling, and we report all of our results based on this mode of operation. Our system can guarantee conservative culling results for two reasons: 1) the underlying HZB algorithm used is itself conservative, and 2) for a given frame *i* we choose the exact same set of LODs for both OR and STC stages. By choosing the same LODs, we ensure that the Z-buffer used for culling is consistent with the geometry it is used to cull. Without this selection algorithm, conservativity is not guaranteed.

We have also modified our run-time pipeline in a number of ways to optionally increase frame rate or decrease latency, by allowing the user to relax the restriction that occlusion culling be performed conservatively:

• **Asynchronous rendering pipeline:** Rather than waiting for the next list of visible geometry from the culling stage (STC) to render frame $i+1$, the render stage (RVG) can instead proceed to render another frame, still using the geometry from frame *i*, but using the most recent camera position, corresponding to the user's most up-to-date position. This modification eliminates the extra frame of latency introduced by our method. The main drawback is that it may introduce occlusion errors that, while typically brief, are potentially unbounded when the user moves drastically.

• **Nth Farthest Z Buffer Values:** The occlusion culling can be modified to use not the farthest Z values in building the depth pyramid, but the Nth farthest [20], thereby allowing for approximate "aggressive" culling.

• **Lower HZB resolution for occluder rendering:** The pixel resolution of the OR stage can be set smaller than that of the RVG stage. If readback from the depth buffer or HZB computation is relatively slow, this can improve the performance. However, using a lower resolution source for HZB allows for the possibility of depth buffer aliasing artifacts that can manifest themselves as small occlusion errors. In practice, however, we have not been able to visually detect any such errors when using OR depth buffers with as little as half the RVG resolution.

6. Implementation and Performance

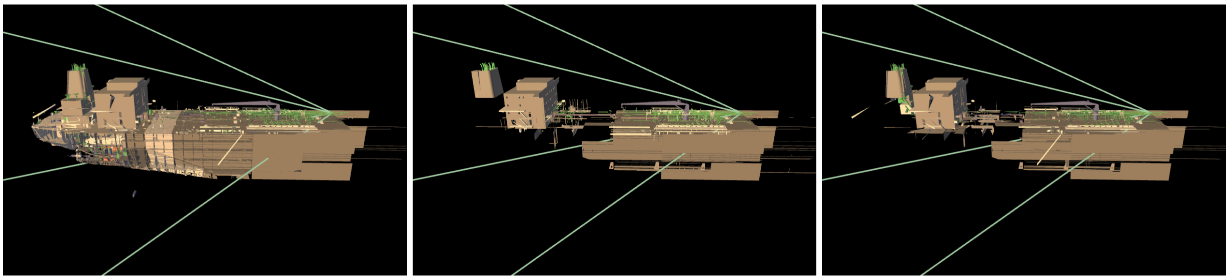
We have implemented our parallel rendering algorithm on two hardware systems. The first is a shared-memory multiprocessor machine with dual graphics rasterization pipelines: an SGI Onyx workstation with 300MHz R12000 MIPS processors, Infinite Reality (IR2e) graphics boards, and 16GB of main memory. Our algorithm uses three CPUs and two graphics pipelines of this machine. We have also made a preliminary port of the system to a pair of networked, dual processor PCs: both are Dell Precision Workstations with GeForce 4 graphics cards, 2GHz processors, and 2GB of main memory.

All of the inter-process communication is implemented using a templated producer-consumer queue data structure. For the SGI implementation, this uses shared memory to pass data between processes. On the PC, the queue class was re-implemented to pass data over TCP/IP sockets. Each stage (OR,STC,RVG) is connected with the others using one or more instances of this queue data structure. Synchronization between the processes is accomplished by pushing sentinel nodes onto the shared queues to delimit the data at the end of a frame. The scene graph resides in shared memory in the SGI version, and is simply replicated on both PCs for the PC version. The overall run-time system is about $6,000$ lines of C++ code.

We have tested the performance of GigaWalk on two complex environments, a coal-fired Power Plant (shown in Fig. 1) and a Double Eagle Tanker (shown in Plate 1). The details about these environments are shown in Table 1. In addition to the model complexity, the table also lists the object counts after the clustering and partitioning steps. Unless otherwise noted, performance results from this point on will refer to the SGI implementation.

6.1. Improvement in Frame Rate

GigaWalk is able to render our two example complex environments at interactive rates from most viewpoints. The frame rate varies from 11 to 50 FPS. It is more than 20 frames a second from most viewpoints in the scene. We have recorded and analyzed some example paths through these models, as shown on the video available at the WWW site: *http://gamma.cs.unc.edu/GigaWalk*. In Fig. 5, we show the improvement in frame rate for each environment. The graphs compare the frame rate for each individual rendering acceleration technique alone and for the combination. Table 2 shows the average speed-ups obtained by each technique over the same path. The comparison between the techniques for a given viewpoint is shown in Fig. 4.



(a) Polygon Count = 202666 (b) Polygon Count = 3578485 (c) Polygon Count = 61771

Figure 4: Comparison between different acceleration techniques from the same viewpoint. (a) Rendered with only HLODs. (b) Rendered with only HZB occlusion culling. (c) Rendered with GigaWalk using HLODs and HZB occlusion culling.

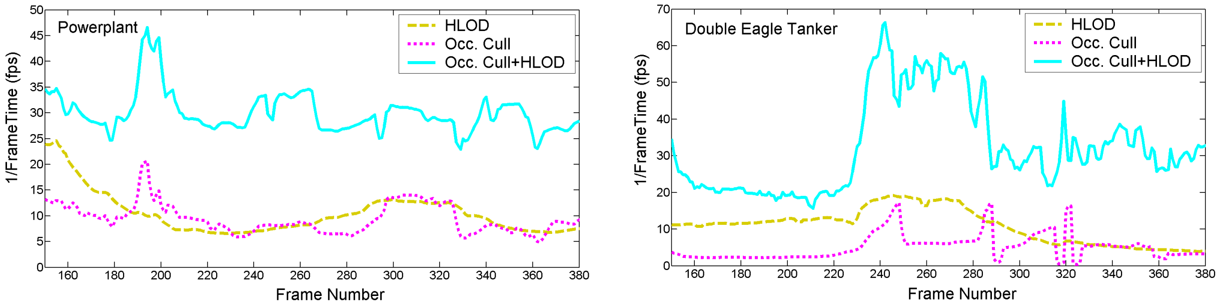


Figure 5: Comparison of acceleration techniques on a path in the Power Plant model at 10 pixels of error (left) and Double Eagle at 30 pixels of error (right), on the SGI Workstation using two IR pipelines. The Y-axis shows the instantaneous frame rate. The combination of HLODs + occlusion culling results in 2-5 times improvement over using only one of them. Display resolution was 640 × 480.

The networked PC implementation achieved an average frame rate of 10 frames per second on the Powerplant model at 1024 × 1024 resolution with at most 10 pixels of screen-space error, and about 11.5 frames per second on the Double Eagle tanker model rendered at 1024 × 1024 resolution with at most 20 pixels of screen-space error. It compares favorably with the shared memory implementation, but has much higher variance. This increase in variance is due to latency incurred from TCP/IP network buffering.

6.2. Culling Performance

Figure 6 shows the number of objects and polygons rendered for each frame on a path through the Power Plant and a path through the Double Eagle. It is clear from the left graphs that most of the reduction in object count comes from occlusion culling. The differences between the exact visibility counts and GigaWalk's are explained by GigaWalk's HZB occlusion algorithm, which culls based on objects' axis-aligned screen-space bounding rectangles rather than actual object polygons. On average for these paths, GigaWalk draws about twice many objects as a perfect object-level visibility algorithm, and about ten times as many polygons as a perfect polygon-level visibility algorithm.

		Object Count			
Env	**Poly**	**Init**	**Part**^{1}	**Clust**	**Part**^{2}
	$\times 10^6$	$\times 10^4$	$\times 10^4$	$\times 10^3$	$\times 10^5$
PP	12.2	0.12	6.95	3.33	0.38
DE	82.4	12.7	2.21	2.31	1.2

Table 1: A breakdown of the complexity of each environment. **Poly** is the polygon count. **Init** is the number of objects in the original dataset. The algorithm first partitions (**Part**^{1}) objects into sub-objects, then generates clusters (**Clust**), and finally partitions large uneven spatial clusters **Part**^{2}. The table shows the object count after each step.

6.3. System Latency

Our algorithm introduces a frame of latency to rendering times. Latency can be a serious issue for many interactive applications like augmented reality. Our approach is best suited for latency-tolerant applications, namely walkthroughs of large synthetic environments on desktop or projection displays. The end-to-end latency in the shared-memory imple-

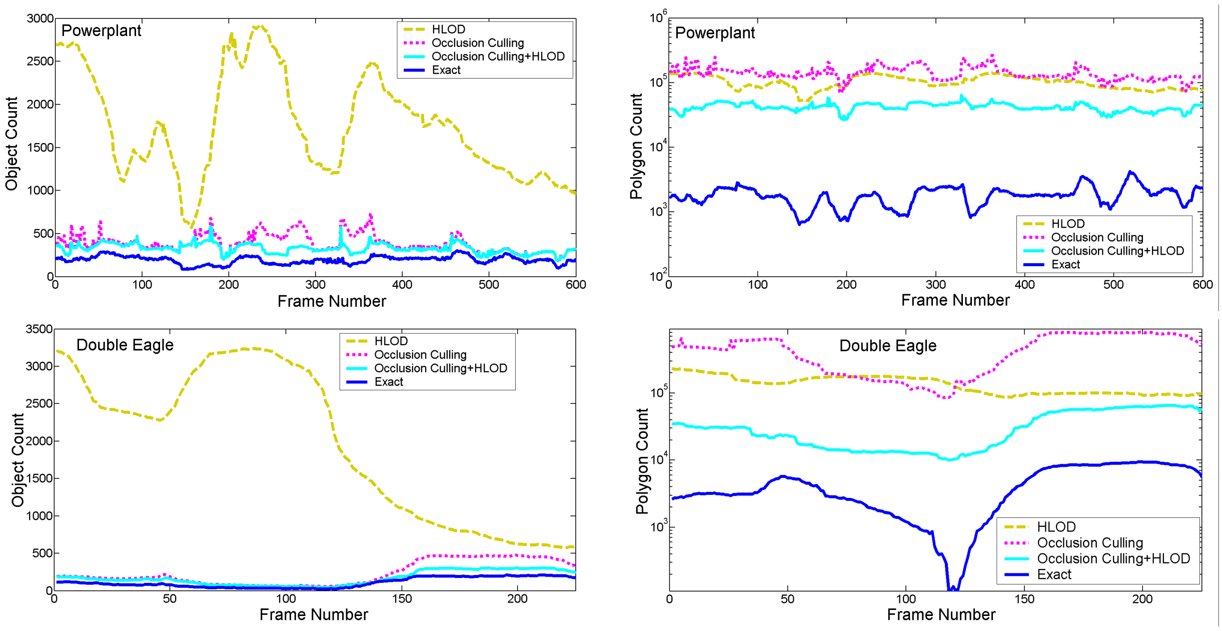


Figure 6: Comparison of object counts (left column) and polygon counts (right column) for different acceleration techniques. Top row is a path on the Powerplant model at 10 pixels of error. Bottom row is a path on the Double Eagle model at 30 pixels of error. The Y-axis shows number of objects or polygons drawn. "Exact" indicates what would be drawn by a perfect visibility algorithm using HLODs. Display resolution was 640x480.

Model	Average FPS			
	OCH	HLOD+VFC	OCC+VFC	VFC
PP	30.67	9.55	9.48	1.15
DE	29.43	9.76	3.27	0.02

Table 2: Average frame rates obtained by different acceleration techniques over the sample path. **FPS** = Frames Per Second, **HLOD** = Hierarchical levels of detail, **OCH** = Occlusion culling with HLOD, **OCC** = Occlusion Culling, **VFC** = View Frustum Culling

mentation varies with the frame rate. It is typically in the range 50 – 150 ms. The high end of this range is achieved when the frame rate dips close to 10 frames a second. This latency is within the range that most users can easily adapt to (less than 300 ms) without changing their interaction mode with the model [13].

In many interactive applications, the dominant component of latency is the frame rendering time [13]. Through the use of our two-pass occlusion culling technique, our rendering algorithm improves the frame rate by a factor of 3-4. As a result, the overall system latency is decreased, in contrast to an algorithm that does not use occlusion culling.

6.4. Preprocessing

This section reports the amount of time and memory used by our preprocessing.

6.4.1. Time and Space Requirements

The preprocessing was done on a single-processor 2GHz Pentium 4 PC with 2GB RAM. The preprocessing times for the Double Eagle model were: 177 min for hierarchy generation (partitioning/clustering), and 32.5 hours for out of core HLOD generation. The size of the final HLOD scene graph representation is 7.6GB which is less than 2 times the original data size. The AABB hierarchy skeleton occupies 7MB of space, though this could easily be further reduced.

The main memory requirement for partitioning and clustering is bounded by the size of the largest object/cluster. For the Double Eagle it was less than 200MB for partitioning, 1GB for clustering and 300MB for out of core HLOD generation.

6.5. Comparison with Earlier Approaches

A number of algorithms and systems have been proposed for interactive display of complex environments. These include specialized approaches for architectural, terrain and urban environments, as highlighted in Section 2. Given low depth complexity scenes, or scenes composed of large or convex occluders (e.g, architectural or urban models), our general approach is not likely to perform any better than special-purpose algorithms designed specifically to exploit such features.

Of the previous systems which do handle general environments, however, few have been able to reduce both depth complexity (e.g. by using occlusion culling) and screen-space complexity (e.g. by using LODs). It is worth making the comparison with one previous system which was designed to do both, the MMR[1].

The MMR system combined LODs and occlusion culling for near-field geometry with image-based textured meshes to approximate the far-field, in a cell-based framework. While the combination of techniques proved capable of achieving interactive frame rates, the system had some drawbacks. First, the creation of cells required user intervention. For instance, some hand-selected model-dependent features were used in the Powerplant to define viewpoint cells. In contrast, the preprocessing and scene graph computation in GigaWalk is fully automatic. Second, the image-based far-field representations used in the MMR system resulted in dramatic popping and distortion when switching between different cells. Third, the memory overhead and preprocessing cost of creating six meshes and textures per cell was quite quite high. Finally, since the MMR used just a single rendering pipeline, it could only afford to use a few objects as occluders, rather than all the visible objects from the previous frame. The occluders had to be pre-selected offline using a heuristic which could not always find good candidates.

In MMR's favor, however, the cell based spatial decomposition allowed for a simple out-of-core prefetching and rendering algorithm. In contrast, GigaWalk currently assumes that the entire scene graph and the LODs and HLODs are loaded into main memory.

7. Conclusions and Lessons Learned

We have presented an approach to rendering interactive walkthroughs of complex 3D environments. The algorithm features a novel integration of conservative occlusion culling and levels-of-detail using a parallel algorithm. We have demonstrated a new parallel rendering architecture that integrates these acceleration techniques on two graphics pipelines, and highlighted its performance on two complex CAD environments. To the best of our knowledge, GigaWalk is the first system that can render such complex environments at interactive rates with this level of fidelity.

There are many complex issues with respect to the design and performance of systems for interactive display of complex environments. These include load balancing, extent of parallelism and scalability of the resulting approach, the effectiveness of occlusion culling and issues related to loading and managing large datasets.

7.1. Load Balancing

There is a trade-off between the depth of scene graph, which is controlled by the choice of minimum cluster size, and the culling efficiency. Smaller bounding boxes lead to better culling since more boxes can be rejected, so less geometry is sent to RVG. On the other hand, more boxes increases the cost of scene graph traversal and culling in STC. In our system, scene traversal and object culling (STC) operate in parallel with rendering (OR and RVG). If our performance bottleneck is the rendering processes (RVG and OR), we can shift the load back to the culling (STC) process by creating a finer partitioning. Conversely, we can use a coarser partitioning to move the load back to RVG and OR. Thus, the system can achieve load balancing between different processes running on the CPUs or graphics pipelines by changing the granularity of partitioning.

7.2. Parallelism

Parallel graphics hardware is increasingly being used to improve the rendering performance of walkthrough systems. Generally, though, the speed-up obtained from using N pipelines is no more than a factor of N. Using a second pipeline for occlusion culling (i.e. $N = 2$), however, enables GigaWalk to achieve more than two times speed-up for scenes with high depth complexity. For low depth complexity scenes there is little or no speed-up, but there is no loss in frame rate as the occlusion culling is performed using a separate pipeline. However, our parallel algorithm does introduce a frame of latency.

Note also that other parallel approaches [21, 37] are fundamentally orthogonal to our approach, and could potentially be used in conjunction with our architecture as black-box replacements for the OR and RVG rendering pipelines.

7.3. Load Times

One of the considerations in developing a walkthrough system to render gigabyte datasets is the time taken to load gigabytes of data from secondary storage, which can be many hours. To speed up the system we have implemented an on-demand loading system. Initially the system takes a few seconds to load the skeletal representation of the scene graph with just bounding boxes. Once loaded, the user commences the walkthrough while a fourth, asynchronous background process automatically loads the geometry for the nodes in the scene graph that are visible. We have found that adding such a feature is very useful in terms of system development and testing its performance on new complex environments.

8. Limitations and Future Work

Our current implementation of GigaWalk has many limitations. The current system works only for static environments, and it would be desirable to extend it to dynamic environments as well, perhaps with a strategy similar to that proposed in Erikson et al.[10].

The memory overhead of GigaWalk can be high. In the current implementation, viewing an entire model requires loading the scene graph and HLODs. Vardahan and Manocha[43] have recently developed an out-of-core algorithm that renders massive datasets using view-frustum culling and LOD/HLOD based selection which we may be able to benefit from.

The preprocessing time for our largest dataset, the Double Eagle, was also higher than desired. Since most nodes in the scene graph are non-overlapping, the LODs can be generated independently. Thus the algorithm could compute the LODs and HLODs in parallel, using multiple threads. This could improve the preprocessing performance considerably, reducing the 32.5 hours spent on the Double Eagle to a few hours.

The algorithm described in this paper guarantees image quality in terms of a bound on screen-space LOD error. First, we recognize that this is far from an ideal image-quality metric, and better metrics which are suitable for interactive display are desired. Second, our system gives no guarantees on the frame rate. The current system would be improved by the addition of a target-frame-rate rendering mode. Furthermore, the current system's use of static LODs and HLODs leads to some popping when switching between different levels. We would like to explore view-dependent or hybrid view-dependent/static LOD-based simplification approaches that can improve the fidelity of our geometric approximations without increasing the polygon count.

Finally, while the current PC implementation shows promise, we need to lower the networking latency in the system. Our implementation indicates that the bandwidth is sufficient with commodity TCP/IP over Ethernet, but to reduce latency it may be necessary to move to a lightweight protocol like UDP or even use specialized low-latency network hardware like Myrinet. We are also interested in using new hardware occlusion culling extensions on PC graphics cards to accelerate GigaWalk. Govindaraju et al.[18] recently devised one approach which uses three PCs and three GPUs.

Acknowledgments

Our work was supported in part by ARO Contract DAAD19-99-1-0162, NSF award ACI 9876914, ONR Young Investigator Award (N00014-97-1-0631), a DOE ASCI grant, and by Intel Corporation.

The Double Eagle model is courtesy of Rob Lisle, Bryan Marz, and Jack Kanakaris at NNS. The Power Plant environment is courtesy of an anonymous donor. We would like to thank Carl Erikson, Brian Salomon and other members of UNC Walkthrough group for their useful discussions and support. Special thanks to Sungeui Yoon for porting GigaWalk to the PC, and to Dorian Miller for help measuring end-to-end latencies with his latency meter [31].

References

1. D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An integrated massive model rendering system using geometric and image-based acceleration. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1999.
2. J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, pages 41–50, 1990.
3. C. Andujar, C. Saona-Vazquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of detail through hardly-visibly sets. In *Proceedings of Eurographics*, 2000.
4. J. Alex and S. Teller. Immediate-mode ray-casting. Technical report, MIT LCS Technical Report 784, 1999.
5. D. Bartz, M. Meibner, and T. Huttner. OpenGL assisted occlusion culling for large polygonal models. *Computer and Graphics*, 23(3):667–679, 1999.
6. D. Cohen-Or, Y. Chrysanthou, and C. Silva. A survey of visibility for walkthrough applications. *SIGGRAPH Course Notes # 30*, 2001.
7. S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1997.
8. F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. *Proc. of ACM SIGGRAPH*, pages 239–248, 2000.
9. C. Erikson and D. Manocha. GAPS: General and automatic polygon simplification. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1999.
10. C. Erikson, D. Manocha, and W. Baxter. HLODs for fast display of large static and dynamic environments. *Proc. of ACM Symposium on Interactive 3D Graphics*, 2001.
11. J. El-Sana, N. Sokolovsky, and C. Silva. Integrating occlusion culling with view-dependent rendering. *Proc. of IEEE Visualization*, 2001.
12. J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, pages C83–C94, 1999.
13. S. Ellis, M. Young, B. Adelstein, and S. Ehrlich. Discrimination of changes of latency during voluntary hand movement of virtual objects. In *Proc. of the Human Factors and Ergonomics Society*, 1999.
14. P. Felzenszwalb and D. Huttenlocher. Efficiently computing a good segmentation. In *Proceedings of IEEE CVPR*, pages 98–104, 1998.
15. T.A. Funkhouser, D. Khorramabadi, C.H. Sequin, and S. Teller. The UCB system for interactive visualization of large architectural models. *Presence*, 5(1):13–44, 1996.

16. B. Garlick, D. Baum, and J. Winget. Interactive Viewing of Large Geometric Databases Using Multiprocessor Graphics Workstations. In *SIGGRAPH '90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation), volume 28*, 1990.
17. M. Garland and P. Heckbert. Surface simplification using quadric error bounds. *Proc. of ACM SIGGRAPH*, pages 209–216, 1997.
18. N. Govindaraju, A. Sud, S. Yoon, and D. Manocha. Parallel Occlusion Culling for Interactive Walkthroughs using Multiple GPUs TR02-27, Dept. of Computer Science, UNC-Chapel Hill, 2002.
19. N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238, 1993.
20. N. Greene. Occlusion culling with optimized hierarchical Z-buffering. In *ACM SIGGRAPH COURSE NOTES ON VISIBILITY, # 30*, 2001.
21. G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A scalable graphics system for clusters. *Proc. of ACM SIGGRAPH*, 2001.
22. T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Proc. of ACM Symposium on Computational Geometry*, pages 1–10, 1997.
23. H. Hoppe. Progressive meshes. In *Proc. of ACM SIGGRAPH*, pages 99–108, 1996.
24. H. Hoppe. View dependent refinement of progressive meshes. In *ACM SIGGRAPH Conference Proceedings*, pages 189–198. ACM SIGGRAPH, 1997.
25. H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization Conference Proceedings*, pages 35–42, 1998.
26. J.B. Kruskal On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of American Mathematical Society*, 7:48–50, 1956.
27. J. Klowoski and C. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Trans. on Visualization and Computer Graphics*, 6(2):108–123, 2000.
28. J. Klowoski and C. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Trans. on Visualization and Computer Graphics*, 7(4):365–379, 2001.
29. D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygon environments. In *Proc. of ACM SIGGRAPH*, 1997.
30. D. Luebke. A developer's survey of polygon simplification algorithms. *IEEE CG & A*, pages 24–35, May 2001.
31. D. Miller and G. Bishop. Latency meter: A device for easily monitoring VE delay. In *Proceedings of SPIE*, Vol. #4660 Stereoscopic Displays and Virtual Reality Systems IX, San Jose, CA, January 2002.
32. J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993.
33. S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. *Proc. of ACM SIGGRAPH*, 2000.
34. B. Schneider, P. Borrel, J. Menon, J. Mittleman, and J. Rossignac. BRUSH as a walkthrough system for architectural models. In *Fifth Eurographics Workshop on Rendering*, pages 389–399, July 1994.
35. W. Schroeder. A topology modifying progressive decimation algorithm. In *Proceedings of Visualization'97*, pages 205–212, 1997.
36. G. Schaufler, J. Dorsey, X. Decoret, and F. Sillion. Conservative volumetric visibility with occluder fusion. *Proc. of ACM SIGGRAPH*, pages 229–238, 2000.
37. R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. *Eurographics/SIGGRAPH workshop on Graphics Hardware*, pages 99–108, 2000.
38. A. Sud, N. Govindaraju, and D. Manocha. Partitioning and Clustering Large Environments for Interactive Walkthroughs TR02-29, Dept. of Computer Science, UNC-Chapel Hill, 2002.
39. S. J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, CS Division, UC Berkeley, 1992.
40. I. Wald, P. Slusallek, and C. Benthin. Interactive distributed ray-tracing of highly complex models. In *Rendering Techniques*, pages 274–285, 2001.
41. P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques*, pages 71–82, 2000.
42. P. Wonka, M. Wimmer, and F. Sillion. Instant visibility. In *Proc. of Eurographics*, 2001.
43. G. Varadhan and D. Manocha. Out-of-Core Rendering of Massive Geometric Environments TR02-28, Dept. of Computer Science, UNC-Chapel Hill, 2002. To appear in Proc. of IEEE Visualization, 2002.
44. J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, June 1997.
45. H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. *Proc. of ACM SIGGRAPH'97*, 1997.

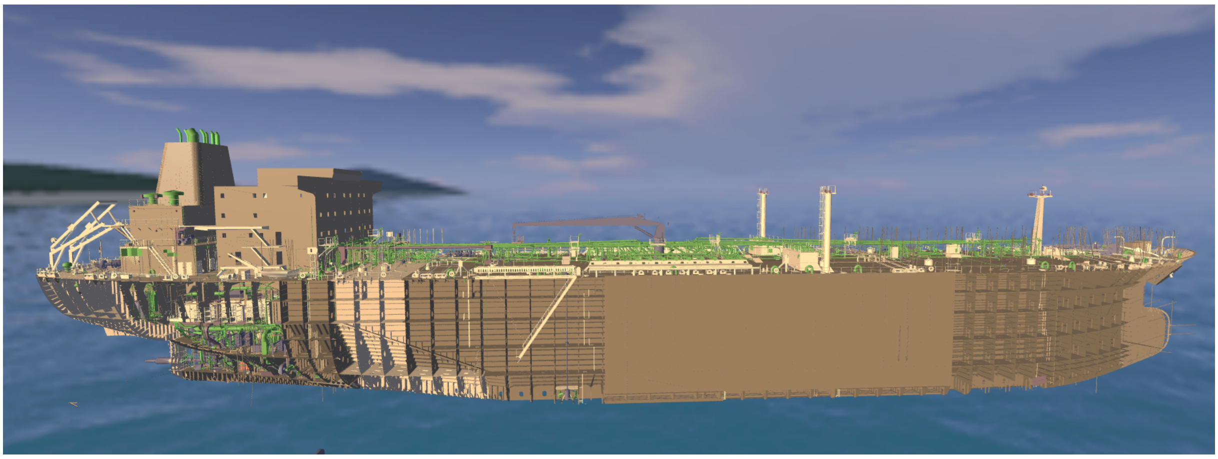
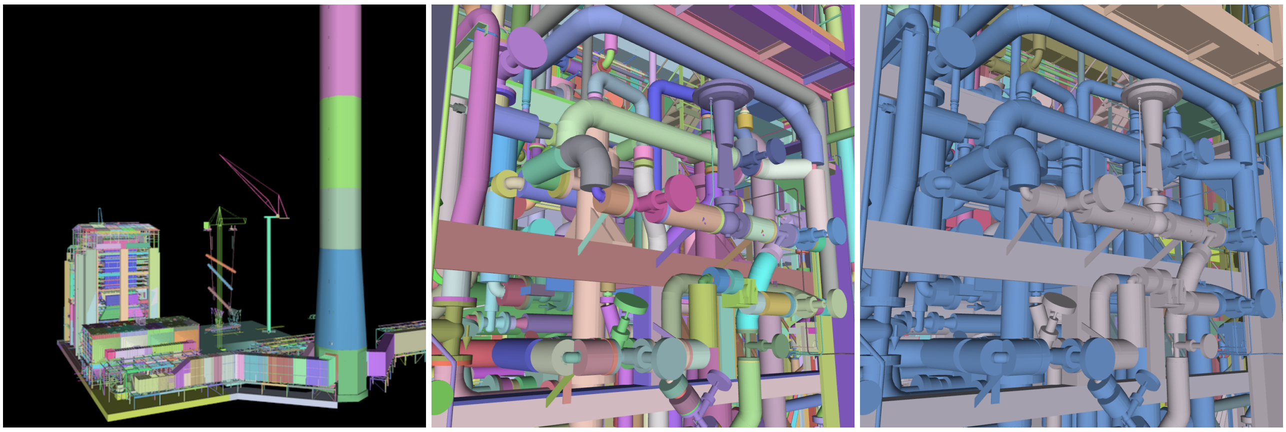


***Plate 1**: Double Eagle Tanker: This 4 gigabyte environment consists of more than 82 million triangles and 127 thousand objects. Our algorithm can render it 11-50 frames per second on an SGI system with two IR2 graphics pipelines and three 300MHz R12000 CPUs.*

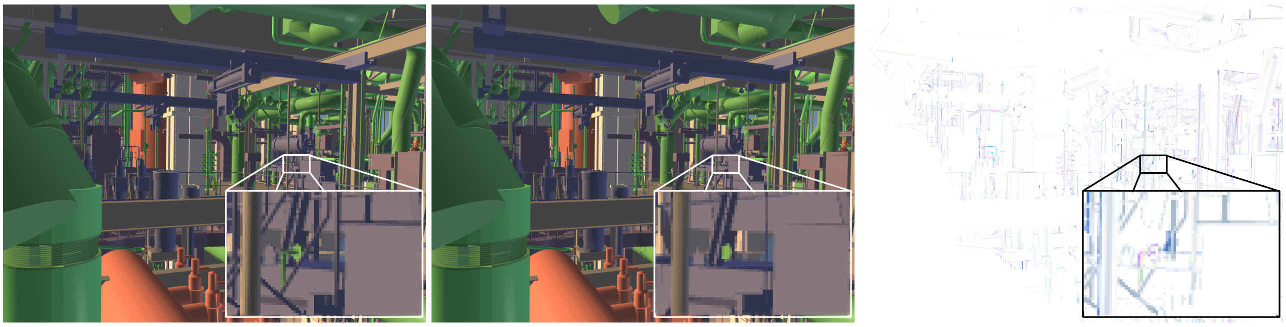


(a) Partitioning & Clustering on Power Plant

(b) Original Objects in Double Eagle

(c) Partitioning & Clustering on Double Eagle

***Plate 2**: The image on the left shows the application of the partitioning and clustering algorithm to the Power Plant model. The middle image shows the original objects in the Double Eagle tanker model with different colors. The right image shows the application of the clustering algorithm on the same model. Each cluster is shown with a different color.*



(a) Pixel Error = 0

(b) Pixel Error = 20

(c) Difference Image

***Plate 3**: The Engine Room in the Double Eagle Tanker displayed without and with HLODs. The inset shows a magnification of one region. Original resolution 1280×960.*