Course Notes: State-of-the-art in Massive Model Visualization

David Kasik[1]
The Boeing Company

Beat Bruderlin
Technical University of Ilmenau
3DInteractive GmbH

Wagner Correa
IBM

Andreas Dietrich
Saarland University
inTrace GmbH

Enrico Gobbetti
CRS4

Fabio Marton
CRS4

Dinesh Manocha
University of North Carolina

Inigo Quilez
VRContext

Philipp Slusallek
Saarland University
inTrace GmbH

Abe Stephens[2]
SCI Institute
University of Utah

Sung-Eui Yoon
Lawrence Livermore National Laboratories

[1]Course organizer.
[2]Course notes editor.

Contents

1 Course Description **2**
1.1 Prerequisites . . . 2
1.2 Speaker Information . . . 3

2 Syllabus **5**
2.1 Motivation and Challenges . . . 5
2.2 Overview Technical Context . . . 5
2.3 Efficient Data Reduction and Cache-Coherent Techniques toward Real-time Performance . . . 5
2.4 SPVN: A New Framework for Distributed Interactive Visualization of Large Datasets 6
2.5 Visibility-guided Rendering to Accelerate 3D Graphics Hardware Performance . . 6
2.6 GPU-friendly Accelerated Mesh-based and Mesh-less Techniques for the Output-sensitive Rendering of Huge Complex 3D Models . . . 7
2.7 Ray Tracing with Multi-core/Shared Memory Systems . . . 8
2.8 Massive Model Visualization Using Realtime Ray Tracing . . . 8
2.9 Putting Theory into Practice . . . 9
2.10 Closing Panel . . . 9

3 Motivation and Challenges **10**

4 Overview Technical Context **39**

5 Efficient Data Reduction and Cache-Coherent Techniques toward Real-time Performance **43**

6 SPVN: A New Framework for Distributed Interactive Visualization of Large Datasets **101**

7 Visibility-guided Rendering to Accelerate 3D Graphics Hardware Performance **108**

8 GPU-friendly Accelerated Mesh-based and Mesh-less Techniques for the Output-sensitive Rendering of Huge Complex 3D Models **133**

9 Ray Tracing with Multi-core/Shared Memory Systems **166**

10 Massive Model Visualization Using Realtime Ray Tracing **170**

11 Putting Theory into Practice **197**

1 Course Description

This course instructs students in the software and hardware strategies needed for real-time visualization and interaction with massive models. Eight international researchers and practitioners are the instructors. The general form of the course will be lecture with live demos.

Even with the advent of higher performance computing systems, the game and entertainment community has developed a set of techniques to limit model size during a real-time visualization and interaction session. However, polygon decimation, texture maps, etc., don't readily apply to domains where high levels of visual accuracy are essential. Such models can contain a billion polygons or voxels and millions of individually selectable objects.

While we include course sections that address ray-tracing and GPU rendering, our objective is to explore a systems approach. We focus on system integration and optimization approaches that let us extract higher performance. We consider rendering methods to be one of many areas of improvement. Therefore, our course complements more detailed offerings dedicated to specific rendering methods. Our overall systems approach to real-time visualization includes:

- Software techniques to overcome performance and memory size limitations (kd-trees, occlusion culling, multi-threaded programming, parallel processor transaction management, memory-mapped files, display lists, cache coherence, temporal coherence)
- Computing system architecture (parallel processor architectures, single and multi-GPU hardware, thin client, hardware occlusion culling, cell computers, multi-core CPUs).
- Scalable system architecture (preprocessing, large user communities, model configuration management, network transfer of basic geometry, variable form factor display devices).
- Practical implementation issues.

Each instructor has developed an approach that combines one or more of the above technologies into a successful interactive 3D viewing application. The course sets the context for the work, discusses overall performance improvement strategies, gives examples of industrial and academic approaches using both rasterization and ray tracing, and concludes with real world experience in a commercial environment.

1.1 Prerequisites

- General knowledge of the difference between ray tracing and rasterization.
- Familiarity with computing system architecture, display list concepts, graphics hardware accelerators, and parallel processing.

This course is intended for users of complex models and practitioners who build real-time 3D applications. The techniques are applicable to any community that commonly reduces model detail (e.g., games) or works only with model chunks (e.g., CAD). We contrast the strategies needed for real-time in a limited computing environment.

1.2 Speaker Information

Dave Kasik is the Boeing Enterprise Visualization Architect. His research interests include innovative combinations of basic 3D graphics and user interface technologies and increasing awareness of the impact of visualization technology inside and outside Boeing. Dave has a BA in Quantitative Studies from the Johns Hopkins University and an MS in Computer Science from the University of Colorado. He is a member of IEEE, ACM, ACM SIGGRAPH (he has attended all SIGGRAPH conferences), and ACM SIGCHI. He is a member of the editorial board for IEEE Computer Graphics and Applications.

Enrico Gobbetti is the founder and director of the Visual Computing (ViC) group at the Center for Advanced Studies, Research, and Development in Sardinia (CRS4). At CRS4, Enrico developed and manages a graphics research program supported through industrial and government grants. His research interests span many areas of computer graphics. His most recent contributions include a new breed of coarse-grained adaptive multiresolution techniques for processing and rendering large scale geometric models. Enrico holds an Engineering degree (1989) and a Ph.D. degree (1993) in Computer Science from the Swiss Federal Institute of Technology in Lausanne (EPFL). For more information, see www.crs4.it/vic

Fabio Marton Fabio Marton is a researcher in the Visual Computing (ViC) group at the Center for Advanced Studies, Research, and Development in Sardinia (CRS4). He holds a Laurea (M. Sc.) degree (1999) in Computer Engineering from the University of Padova. His current research interests include out-of-core data processing, multiresolution modeling and time-critical rendering. For more information, see www.crs4.it/vic

Dinesh Manocha is currently a distinguished professor of Computer Science at the University of North Carolina at Chapel Hill. He was selected as an Alfred P. Sloan Research Fellow and has received significant awards from NSF, ONR, Honda, and UNC. He has received numerous best paper awards and has served on program committees for computer graphics and geometric modeling conferences. Manocha has been working on large model visualization for more than 10 years. His research group at UNC Chapel Hill has published numerous papers at SIGGRAPH. He has also organized SIGGRAPH courses on interactive walkthroughs, large model visualization, and GPGPU.

Sung-Eui Yoon is currently a postdoctoral scholar at Lawrence Livermore National Laboratory. He received the B.S. and M.S. degrees in computer science from Seoul National University in 1999 and 2001 respectively. He received his Ph.D. degree in computer science from the University of North Carolina at Chapel Hill in 2005. His research interests include visualization, interactive rendering, geometric problems, and cache-coherent algorithms and layouts. He will join the faculty of Dept. of Computer Science at KAIST (Korea Advanced Institute of Science and Technology) as an assistant professor in July, 2007.

Wagner Correa is a Research Staff Member with the IBM Watson Research Center. He is part of the Visualization Systems Group, where he is the chief architect for SPVN (Scalable Parallel Visual Networking), a framework for developing distributed interactive visualization applications. Prior to joining IBM, Wagner was teaching Computer Graphics at the Federal University of Minas Gerais (UFMG) in Brazil. Wagner holds a B.S. degree (1994) and an M.S. degree (1996) in Computer Science from UFMG, and an M.A. degree (1998) and a Ph.D. degree (2004) in Computer Science from Princeton University.

Beat Bruderlin is professor of Computer Science at the Technical University of Ilmenau, Ger-

many. His work focuses on computer geometry with applications to computer aided design and engineering visualization. Other interests include new interaction techniques for 3D design. Beat Bruderlin received his M.S. degree in Physics from the University of Basel and a PhD in Computer Science from the Swiss Federal Institute of Technology (ETH) Zurich, Switzerland. He was a faculty member at the University of Utah, before joining TU Ilmenau. In 2004 he founded 3Dinteractive GmbH, a spin-off company developing interaction and rendering software.

Abe Stephens is a PhD student at the University of Utah's SCI Institute. His work focuses on adaptive rendering methods as well as large data visualization. He is a principle contributor to the Manta Interactive Ray Tracer and has worked with Intel and Silicon Graphics to improve interactive ray tracing techniques on parallel systems. Abe received a BS in Computer Science from Rensselaer Polytechnic Institute in 2003.

Philipp Slusallek is professor for computer graphics and digital media at Saarland University, Germany. Before joining Saarland University he was visiting assistant professor at Stanford University. He received a Diploma/MSc in physics from the University of Tbingen and a Doctor/PhD in computer science from the University of Erlangen. Philipp has published and taught extensively, including courses on real-time ray tracing at SIGGRAPH05 and SIGGRAPH06. He is the principal investigator for the OpenRT project to establish real-time ray-tracing as an alternative technology for interactive and photorealistic 3D graphics. Recently he co-founded "inTrace", a spin-off company that commercializes real-time ray tracing technology.

Andreas Dietrich is a research assistant and final-year PhD student at the Computer Graphics Group at Saarland University. He is working on real-time ray tracing for VR applications, natural phenomena rendering, and on interactive massive model visualization. Andreas received a masters degree (Dipl.-Technoinform.) in computer science from Kaiserslautern University in 2001.

Iñigo Quilez received a degree as a Telecommunications Engineer from the University of Basque Country (Spain), focusing on digital signal processing. He has extensively worked in real-time computer graphics, within the "demoscene" since 1998, especially in the subject of extreme procedural content creation and data compression. Well known in the fractals community, work is still needed to give aesthetics a more important role in the scientific work. Since joining VRcontext in 2003, he focuses on research and development of photorealistic rendering and massive model visualization techniques, specializing in shared memory multi-cpu and multipipe systems (especially the Silicon Graphics architecture).

2 Syllabus

2.1 Motivation and Challenges

Goal: Provide clear understanding of the implications of complex, massive model visualization and how the user community can effectively assist the research community.
Speaker: David Kasik

The human visual system provides an extremely efficient way to communicate both context and detail. The amount of data that's being generated is actually exceeding the rate of change of Moore's law. The domains in which data is expanding range from computer-aided design and manufacturing to arts and entertainment to intelligence analysis. The most effective way people have to comprehend and communicate the overall implications relies on computer graphics. Clear examples from specific domains show the effectiveness of interactive, real-time 3D graphics. The overall system implications of real-time interaction are essential to ultimately make the technology implementable. The attendees should understand all aspects that can make or break the widespread adoption of the techniques described in the other sections of the course.

2.2 Overview Technical Context

Goal: Give the attendee a generalized architectural overview of the issues that must be addressed to solve the massive model visualization
problem. Speaker: Enrico Gobbetti

Despite the rapid improvement in hardware performance, rendering today's multi-gigabyte datasets at interactive rates largely overloads the performance and memory capacity of state-of-the-art hardware platforms. To overcome this limitation, researchers have proposed a wide variety of output-sensitive processing and rendering algorithms, i.e., techniques whose runtime and memory footprint is proportional to the generated output complexity, not to the total model complexity. Each of the detailed presentations contained in this course present different solutions implemented in both industry and academia. This portion of the course gives a general technical context that lets the attendee understand that all methods tackle the same problems and are based on a common framework. This introduction will allow the attendee to relate and contrast individual approaches presented later.

2.3 Efficient Data Reduction and Cache-Coherent Techniques toward Real-time Performance

Goal: The general problem we address in this course is real-time performance on a general purpose computer. Multiple factors affect performance in any computing system: network, disk, memory, cache, hardware accelerators, multiple processors, operating system, software tricks, data compression, decimation, level-of-detail, display size, etc. This section surveys of recent methods to work around performance limitations for real-time interaction on a general purpose computing systems.
Speakers: Dinesh Manocha and Sung-Eui Yoon

Current CPU and GPUs are progressing at an exponential rate (e.g., Moore's Law). However, the growth rate of data access time is lower than that of CPU and GPU widens the gap between

processing speed and data access speed. Therefore, it is critical to reduce data access time, especially for massive models. We outline a number of algorithms to overcome these problems in various applications.

We first outline view-dependent simplification and visibility techniques to reduce amount of data necessary to perform visualization of massive models.ICache-coherent layouts allow us to minimize the data access time to the underlying data representation given CPUs and GPUs. We follow with a discussion of simplification, visibility, and layout techniques in other different applications including rasterization, shadow computation, ray tracing, isosurface extraction and collision detection. Furthermore, we describe novel hierarchical data structures to integrate these algorithms. . The other presenters use variations on all of the generic techniques we define.

We demonstrate the application of our algorithms to different types of complex models on a commodity desktop or laptop. The set of models include large scanned datasets, isosurfaces extracted from simulation data, CAD environments of powerplants, airplanes and tankers, and terrain datasets.

2.4 SPVN: A New Framework for Distributed Interactive Visualization of Large Datasets

Goal: Present SPVN, a new framework for developing distributed interactive visualization applications, and explain the techniques that it uses to achieve good performance and scalability. Speaker: Wagner Correa

This section of the course focuses on the development of distributed interactive visualization applications. We present SPVN (Scalable Parallel Visual Networking), a new programming framework created at IBM Research that allows programmers to easily leverage the performance and scalability of a rendering cluster to run interactive visualization applications dealing with massive datasets.

We start by presenting an overview of the SPVN architecture and discussing two application scenarios for SPVN. One scenario is the use of a rendering cluster to deliver images to a potentially remote desktop. Another scenario is the use of a rendering cluster to drive a display wall for immersive collaboration.

We then explain some of the techniques that SPVN uses to achieve good performance. These techniques include memory management, construction of spatial hierarchies, detail culling, visibility culling, and multithreading.

We conclude by describing two distributed rendering strategies supported by SPVN. The first strategy, called sort-first, partitions the output, dedicating each rendering server to a tile of the screen. The second strategy, called sort-last, partitions the input, dedicates each rendering server to a portion of the dataset.

2.5 Visibility-guided Rendering to Accelerate 3D Graphics Hardware Performance

Goal: The goal of this section is to understand the foundations of VGR and its application to visualization of complex scenes, using hardware-accelerated OpenGL.
Speaker: Beat Bruderlin

Visibility-guided Rendering (VGR) is an output-sensitive approach for real-time rendering of very large 3D data sets, using a GPU. The key issues covered in this section are hardware-based occlusion culling, memory management and out-of-core visibility determination, the data structures and algorithms as well as a host of low-level operating system issues.

The key advantage of VGR is to efficiently determine the visibility of the vast majority of polygons before they are sent to the graphics hardware. Different culling techniques are used in combination. Spatial data structures, as well as hardware features of the GPU can be exploited to implement the culling techniques optimally. The visibility-guided rendering approach relies heavily on efficient memory management, concurrency between CPU and GPU, as well as optimal use of low-level OS functionality to handle very large models out-of-core.

In the outlook we touch on modern multi GPU hardware architecture, the use of programmable shaders in the context of VGR, as well interactive object manipulation functionality: In addition, we show how indirect lighting can be accelerated by GPUs in a hybrid approach of VGR and ray tracing. The presentation concludes with a life demo of our software.

2.6 GPU-friendly Accelerated Mesh-based and Mesh-less Techniques for the Output-sensitive Rendering of Huge Complex 3D Models

Goal: Wrap-up the GPU section and provide clear understanding of output sensitive techniques for models that exhibit complicated geometry and topology, heterogeneous material attributes, as well as large variations in depth complexity. Describe how these techniques can be efficiently implemented on what are now commodity graphics platforms. Present an application to unconventional displays.

Speakers: Enrico Gobbetti and Fabio Marton

Previous presentations have focused on introducing a general framework and detailing techniques for working around performance in various kinds of models. In this talk, we focus on contrasting two state of the art approaches for managing models that have different characteristics, that range from extremely large dense models, such as those deriving from 3D scanning, to extremely complex objects composed of many loosely connected interweaving detailed parts of complex topological structure. In this context, we compare and contrast two different GPU friendly output sensitive techniques for harnessing the raw power and programmability features of graphics chips: a mesh-based framework based on multi-scale geometric models (Adaptive Tetra-Puzzles, SIGGRAPH 2004; Batched Multi-Triangulation, IEEE Viz 2005), that is well suited to models with dense geometric details, and a mesh-less framework (Far Voxels, SIGGRAPH 2005), that handles datasets that combine complicated geometry and appearance with a large depth complexity by modeling model appearance rather geometry.

The course section contains practical examples of the visual inspection of very different kinds of models, ranging from very large CAD assemblies, terrains, isosurfaces, and laser scans visualized on a laptop, to demonstration of multiresolution rendering on a spatial 3D display.

The presentation concludes by wrapping-up the GPU oriented section and setting some ground for the raytracing section of the course.

2.7 Ray Tracing with Multi-core/Shared Memory Systems

Goal: Understand the implications of software ray tracing on parallel systems.
Speaker: Abe Stephens

Shared memory computer systems provide a strong platform for interactive software rendering. They combine a large number of processors, enormous amounts of memory and many shared devices across a single system.

Special hardware in these systems provides many facilities that must be implemented by hand on a cluster. Still multi-processor servers, or even multi-core workstations, behave differently from standard single processor desktops and the graphics programmer must pay attention to the system architecture in order to obtain optimal performance. The challenges encountered implementing an interactive renderer on such a system are not well addressed by existing HPC tools and programming techniques and require the programmer to interact with the system at a much lower level.

Moving beyond the rendering algorithm content discussed in the SIGGRAPH 2005 and SIGGRAPH 2006 ray tracing courses, we provide detailed examples of how parallel system architecture effects renderer implementation. We examine both multi-core workstations and multiprocessor platforms. Both types of systems are used as an interactive ray tracing platforms for engineering design situations.

2.8 Massive Model Visualization Using Realtime Ray Tracing

Goal: Demonstrate how Real-Time Ray Tracing can easily deal with many of the challenges of visualizing massive models of surfaces and volumes while enabling advanced visual effects like shadows and global illumination. Both HW and SW approaches are discussed and demonstrated.
Speakers: Andreas Dietrich and Phillip Slusallek.

Real-time ray tracing has become an attractive alternative to rasterization based rendering, particularly for highly complex data sets including both surface and volume data. Ray tracing handles massive datasets well because of output sensitivity and the logarithmic complexity of the ray tracing computations with respect to the scene size. Ray tracing easily handles huge scenes as long as they fit into main memory. For even larger data sets, active memory management is necessary to always keep the working set in main memory and on-demand swap data in and out depending on its visibility.

All these operations require spatial index structures (e.g. kd-trees) that need to be built out-of-core and often offline. We discuss efficient techniques for this task, including approaches that allow efficient memory management at runtime. In addition, we discuss several extensions that are necessary for efficiently ray tracing large models.

Handling dynamic scenes has been a major issue with ray tracing. We outline two areas where significant progress has recently been made: (1) Designing scenes graphs for efficiently handling changes in very large models and (2) Novel index structures that allow fast updates after changes to the geometry.

An intrinsic property of ray tracing is that once a scene can be ray traced, adding advanced optical effects or lighting simulation is fairly straightforward. We discuss how such advanced effects can be used for achieving photorealistic visualization even of highly complex models such as natural environments with environment lighting, complex shading, and efficient anti-aliasing.

Finally, we discuss trends in hardware and how they help in making ray tracing available to a larger and larger set of applications. In particular we briefly compare the capabilities of multi-core CPUs (including the Cell processor), GPUs, and custom hardware with respect to ray tracing of complex models.

2.9 Putting Theory into Practice

Goal: Review the most common problems, limitations, and considerations that arise from using state-of-the-art techniques coming from the research community into a real product.
Speaker: Inigo Quilez

Moving academic results to a real product is not always easy. Not only is a big part of the papers normal focus on very specific cases but research experiments are also often made with different constraints than those found in production. Even when applicable to a product, many techniques have been only tested in a few well known models used in CG literature, while customers demand algorithms to work in all kind of data, from best case models to ill-formed geometry, that must not be removed from the dataset and that can degrade significantly an algorithm's performance (as accelerating data structures).

On the other hand, the kind of data used in most 3D massive datasets visualization or collision detection research papers handle high density (e.g., the happy Buddha) or medium density models (the PowerPlant or the Boeing 777). However, a product has to also deal with low density massive models. Low density models make some techniques not applicable or less efficient and add a new set of problems.

Another common source of problems is that many techniques assume that pre-computation time and effort is not important for the user. In practice, both the pre-calculation time and the complexity of the pre-compute process is of significant concern to production users.

From the marketing point of view, there is also a resistance present in users to change to new technologies (even if a lot better than old ones); and lot of work must be done to give the application the look and feeling of the tools users are already used to.

Finally, we demonstrate how a visualization and collision detection on massive model looks like in a real time application, both using OpenGL and software ray-tracing.

2.10 Closing Panel

At the end of the course a speaker's panel will allow attendees to pose questions about the material covered during the day or about future directions affecting real-time interactive systems.

3 Motivation and Challenges

SIGGRAPH2007

Motivation and Challenges in Massive Model Visualization

SIGGRAPH2007

Dave Kasik
Technical Fellow
The Boeing Company
Seattle WA USA
david.j.kasik@boeing.com
+1-425-830-4276

Section Goal

- Provide understanding of:
  - Usage scenarios for interactive, massive model visualization.
  - The technology implications of interactive, massive model visualization.
  - How the user community can assist the research community.

Section Outline

- Motivation for effort from a user's perspective, including sample use cases
- Characterization of user tasks that can be addressed by visual analysis
- General processing architecture alternatives
- Additional technical challenges:
- Pragmatics of getting data released to the research community

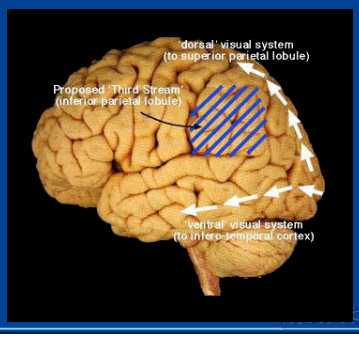
Data Explosion

- All storage media produced about 5 exabytes of new information in 2002.
  - 92% was stored on magnetic media, mostly hard disks.
- This amount of new information is about double of the amount stored in 1999.
- Information flows through electronic channels (telephone, radio, TV, and the Internet) contained ~18 exabytes of new information in 2002.
  - This is 3 1/2 times more than is stored.
  - 98% is voice and data sent telephonically via fixed lines and wireless

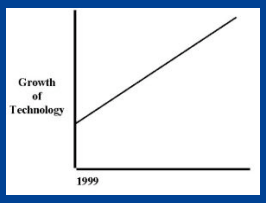
What Do These Numbers Mean?

Kilobyte (KB)	1,000 bytes, 10^3	Typewritten page
Megabyte (MB)	1,000,000 bytes, 10^6	Small novel
Gigabyte (GB)	1,000,000,000 bytes, 10^9	Pickup truck filled with books
Terabyte (TB)	1,000,000,000,000 bytes, 10^{12}	2 TB: academic research library
Petabyte (PB)	1,000,000,000,000,000 bytes, 10^{15}	200 PB: All printed material
Exabyte (EB)	1,000,000,000,000,000,000 bytes, 10^{18}	5 EB: All words ever spoken

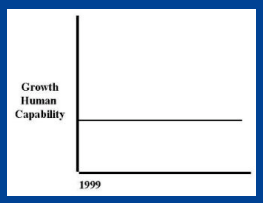
Human Visual Processor



Buxton's Conundrum



Moore's Law



God's Law

Use Cases

- What can a user do with interactive visualization alone?

Visual Task Analysis

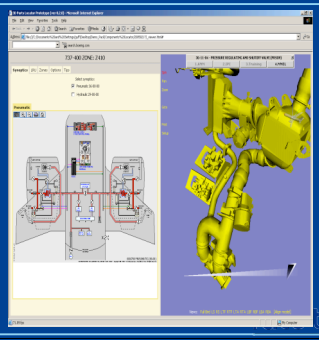
- Find an object in a complex scene.
- Understand surface characteristics (e.g., smoothness, roughness).
- Examine objects in the surrounding volume.
- Visually scan the scene.
- Observe dynamics in the entire scene (conventionally by animation).
- Work with multiple versions of the same set of objects to compare the two sets.

Potential Applications

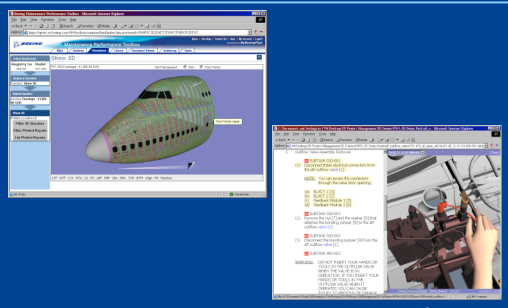
- Design reviews
- Engineering analysis
- Safety
- Survivability
- Part context
- Massive scans
- Quality inspection
- Manufacturing instructions
- Part catalogs
- Training
- Maintenance instructions
- Sales and marketing

- Basically, any process where quick navigation is needed to go anywhere in a digital model

Example 1 – Tracing Systems



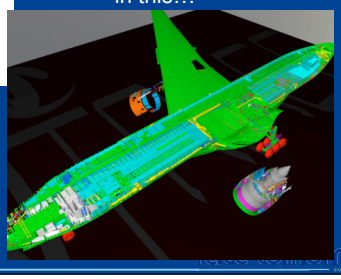
Example 2 – Maintenance Tasks



Example 3 – Design Review

- Find this

in this…



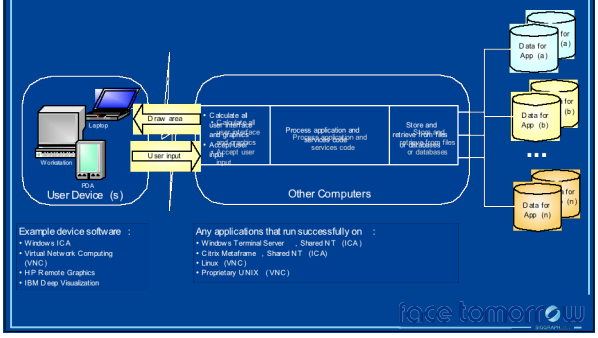
When Is Performance Acceptable?

- Model load time: instantaneous.
  - For groups, 'instantaneous' translates to less than one minute.
  - For an individual, five minutes just seems like an eternity unless the person can effectively time share tasks.
- 'Flying' time: New transformation matrices that respond to mouse action.
  - Ideally, 16 Hz (the human flicker fusion threshold for video).
  - Practically, 10 Hz or faster.
- Graphical selection. Feedback in < .25 seconds.

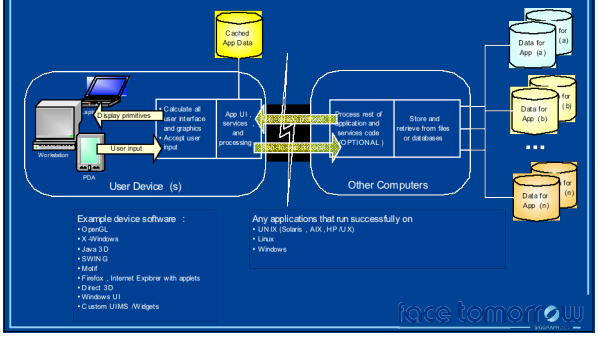
Processing Architectures

- Virtual Terminal
- Remote Data
- Local Data

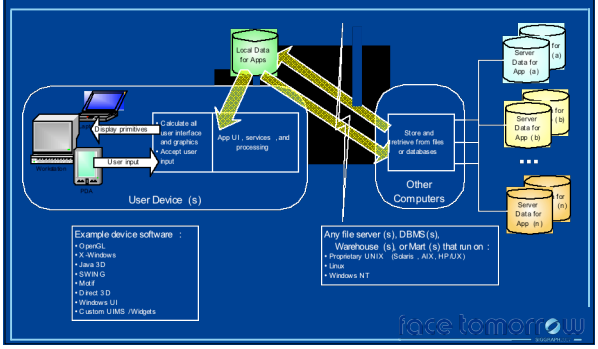
Virtual Terminal



Remote Data



Local Data

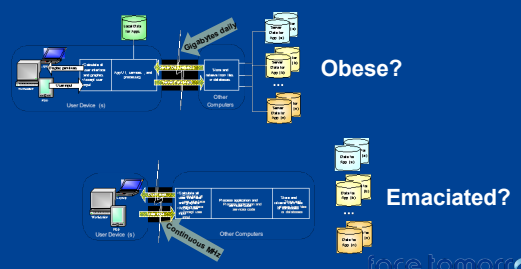


Other Technical Challenges

- Network impact
- Pre-processing
- Collision detection
- Rigid body motion
- Visual model update

Network Impact

- Obese, emaciated, or somewhere in between?



Pre-processing

- All current approaches (both GPU and CPU) pre-process to get interactive performance.
- Routinely costs hours.

Version Management

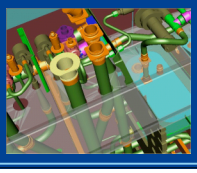
- A detailed design activity may release hundreds of new part versions nightly.
- The base model easily contains hundreds of thousands of parts.
- Two issues:
  - Pre-processing cost to handle the new versions.
  - Methods to select which version should be displayed.

Animation

- Rigid body motion allows parts to move relative to one another.
- Can be the result of all sorts of simulations:
  - Mechanisms
  - Manufacturing assembly plans
  - Training
  - …
- Simulations that result in shape deformation are much more difficult.

Collision Detection

- Common task: figure out what objects erroneously share the same space.
- Subtle problem
  - Some tangent conditions (e.g., parts bolted together) or collisions (e.g., flexible wire sheathing) are OK.



Pragmatics of Data Release

- Find the data owner.
- Be willing to work through a non-disclosure or proprietary information agreement.
- Be willing to subtly manipulate the data to remove intellectual property, export control, military sensitive, or other concerns that lawyers have.
- Be really patient.

Summary

- Outlined overall problem, usefulness quotient, and technical issues.
- The rest of the instructors will provide a deeper technical look and further examine the pragmatics of large model rendering in production.

Bibliography [Eye-Brain]

The Perception of the Visual World. James J. Gibson, 1950, Hougton Mifflin. A coherent study of the stimuli that allow vision to occur at all.

Eye and Brain: The Psychology of Seeing. R. L. Gregory, 1966, World University Library. Connects the vision system to the brain. In some ways, it updates Gibson's 1950 work.

The Intelligent Eye. R. L. Gregory, 1970, McGraw-Hill. An excellent discussion of stereo vision.

Foundations of Cyclopean Perception. Bela Julesz, 1971, U. of Chicago Press. Dealing with analglyphs, a computer-generated form of stereo image.

Bibliography [Visual Techniques]

Patterns in Nature. Peter S. Stevens, 1974, Little, Brown. An examination (by an architect) of the regularity of naturally occurring patterns. Gorgeous visual examples throughout.

The Visual Display of Quantitative Information. Edward R. Tufte, 1983, Graphic Press. The first in a series of four books that discuss different techniques to clearly communicate large volumes of data with meaningful graphic images.

The Psychology of Graphic Images. Manfredo Massironi, 2002, Lawrence Erlbaum. Ties the notion of how authors use graphic images to communicate…and why (from a psychological point of view) those image work.

Strategies for Consistent Image Partitioning

David J. Kasik
The Boeing Company

Graphic display technologies have traditionally targeted devices with midsize screens. However, devices with small and large screen sizes are gaining popularity, with users increasingly attempting to access complex images using small-screen devices. To display high-quality images irrespective of screen size, new methods of visualization become necessary.

Computer graphics display technology has reached a crossroads. New devices have dramatically different display characteristics and continue to rapidly evolve. The most compelling difference is screen size, which now ranges from tiny (as in cell phones or watches) to gargantuan (as in display walls and theater screens). Such variation creates significant problems in the design of images that communicate irrespective of screen size, especially considering the wide variety of displayable images. The "Designing for Variable Screen Sizes" sidebar discusses efforts to integrate images traditionally designed for midsize screens into devices with a variety of screen sizes.

As available computing technology expands, the overall user community has a significantly larger palette of graphics devices available to them. As Figure 1 illustrates, most current research investment aims at medium-sized (10- to 20-inch) screens. Although people are now buying numerous devices with small and large screens, investment in improving the interactive experience has lagged behind.

Graphical user interfaces (GUIs) dominate user interaction with midsize screens, but they neither translate well to small screens nor take advantage of the expanded area available on large screens. Adding complexity is the growing number of people with wide-ranging skill levels, education, and cultural backgrounds using devices with a variety of screen sizes. Finally, many companies' user population spans the globe; thus designers, developers, and deliverers must collaborate to ensure timely delivery of quality products.

This article deals with a fundamental problem: How can designers or users themselves partition images to maximize communication bandwidth to a person viewing the image on a small screen?

Maximizing communication impact

Virtually any 2D image, series of images that create a movie, or 3D image can be drawn just about anywhere. Not all images are created equally, however, and graphic communication techniques vary significantly. Massironi's excellent taxonomy delineates the types of images people have used to communicate visually throughout history.[1] Every image's basic goal is similar: communicate the right information in an easily consumable format.

Although many graphic styles are available, the design, analysis, assembly, and maintenance of complex physical products such as commercial airplanes and satellites use only a few. Boeing and other industrial companies that produce physical products primarily use two types of representational drawing, both subsets of Massironi's taxonomy: *technical drawings* and *descriptive geometry*.

These two types of graphic images appear in a number of different forms. Whereas engineering drawings must be dimensionally accurate and provide the textual information necessary for construction and assembly, other technical drawings don't require such accuracy. For example, technical illustrations, which depict parts relationships for maintenance, and production illustrations, which contain instructions for assembling hydraulic and electrical systems, don't require dimensional accuracy.

Early 3D concept drawings represent notions to customers and initiate more detailed configuration and preliminary design activities. Designers use descriptive geometry techniques to generate detailed product models with 3D surfaces and solids.

Visual task analysis

People who view graphic images of models representing physical products participate in the product's life cycle. Boeing's typical internal user community includes sales, design, building, and maintenance staff. A commercial airplane's user

© 2004 IEEE. Reprinted, with permission, from IEEE

1070-986X/04/$20.00 © 2004 IEEE

Published by the IEEE Computer Society

Designing for Variable Screen Sizes

The variety of graphics display devices has significantly increased since the early 1990s. Announcements of new devices seem to appear weekly. Advances in midsize graphics devices, inexpensive memory, and computing processors have largely removed restrictions on the types of pictures users could draw (images composed only of lines or text with a limited number of colors, for example).

Midsize screens

Current devices use raster technology to provide excellent display quality in terms of color, brightness, and pixel size. Newer devices (such as IBM's MD22292A1 liquid crystal display) come close to achieving a pixel size that attains the minimum angle of resolution (between .00833° and .01667°) for an individual.[1] Vendors are developing unique configurations with midsize screens for special effects such as:

- Automatic stereo (http://www.mrl.nyu.edu/projects/autostereo/)
- Multiple parallel planes for true 3D (http://www.3dmedia.com/products.html)
- Specially treated plastic cubes for true 3D (http://3dtl.com)

Small screens

Devices, like users, are becoming increasingly mobile. Cell phones, personal digital assistants (PDAs), wearable computers with glasses or head-mounted displays, tablet computers, and notebook computers rely on small screens and less capable computers than devices with midsize screens. Small screens generally have fewer pixels and a limited set of colors.

Such limitations haven't decreased these devices' popularity nor have they discouraged designers from customizing general devices to build dedicated, special-purpose devices (onboard navigation aids and barcode scanners, for example). The components needed to build portable devices continue to improve; thus, the only factors that won't change significantly are the display size and the number of pixels devices can display.

Boeing supplies cell phones to most of its employees, and many employees also carry other devices with small screens. Assembly-line mechanics, quality-assurance inspectors, shipping and receiving personnel, and others use the small devices to access the corporate graphic images they need to effectively do their jobs.

Large screens

Large-screen devices come in a range of geometric configurations. Plasma panels use an alternate display technology. Projectors, the de facto standard, increase image size on a planar screen. The Elumens (http://elumens.com) VisionDome products use a conventional projector with a special lens for a hemispherical screen. A truncated cylinder surrounds the viewer's head in a Panoscope360 (http://panoscope360.com). A CAVE (http://www.fakespacesystems.com/workspace1.shtml) projects images onto the planar walls to form a cube. Holograms (http://www.zebraimaging.com) have specially etched large-format film to create a full-3D illusion.

Large-screen device problems differ from those of midsize screens. Because large-screen graphics drivers are based on the pixel resolution of a midsize screen, users can experience a dramatic increase in pixel size depending on their distance from the screen. This phenomenon increases the viewer's angle of

continued on p. 34

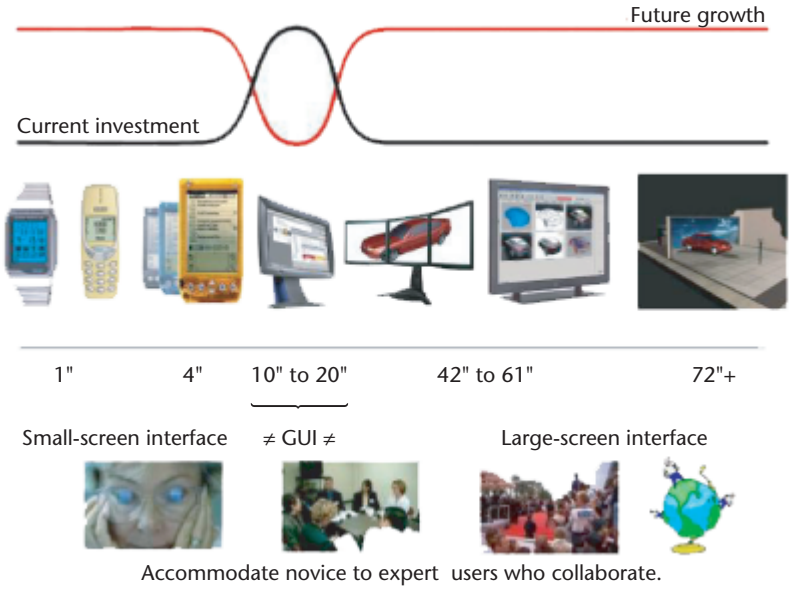


Figure 1. Users can choose among a wide array of devices with different screen sizes although most research investment still goes toward midsize screens.

continued from p. 33

resolution and reduces brightness (a measure of image crispness), making the image appear less smooth than it does on a midsize screen.[2]

User interface evolution

The standard user interface paradigm for application command and control has evolved significantly from text commands typed into glass teletypes. Although early graphics devices supported graphical selection of commands and objects with lightpens, thumbwheels, or tablets, pervasive use of graphic user interfaces (GUIs) didn't occur until raster graphics devices and mouse pointers became inexpensive in the mid-1980s. GUIs have been the basic interface standard since then.

The fundamental assumptions of a GUI for application command and control don't translate well to either small- or large-screen devices. Not only is screen area a problem on small devices, but interaction through a keyboard and mouse is also difficult.

Many large companies, startups, and research projects have suggested ways to interact with small screens. However, most of the efforts focus on effectively filling out forms and entering text rather than interacting with complex 2D or 3D graphics. For example, cell phones have developed enough thumbs-only 10-key typists to sustain text-messaging services. Other companies are introducing products that allow better interfaces for form users.

Microsoft, AT&T, IBM, and other large companies, as well as startups such as Aligo, Cysive, and Trigenix are introducing products for mobile, small-screen devices that focus on applications with forms and Web-based interfaces as opposed to applications in which the user directly interacts with graphics. Several interesting research projects are based on small-screen devices, including Geney (http://geney.juxta.com), a tool for teaching genetics using multiple cooperating PDAs.

Large screens expand the field of view and make both new interaction styles and devices possible. A GUI is well tuned to an individual's interacting with a screen. In fact, the most common technique for controlling interaction with a large-screen device is to engage a designated driver using a mouse, keyboard, and midsize screen. The large screen becomes a slave to the midsize screen, and their displays are identical.

Efforts to explore new user interface strategies for single users and large screens focus on devices (data gloves and head-mounted displays, for example) that enhance the virtual reality illusion. VR interactive tools generally offer interaction that is a GUI analogue but driven by a data glove. Two-handed input (see, for example, a list of Buxton's work at http://billbuxton.com/papers.html#anchor1442822) and tangible interfaces (such as the work by MIT's Tangible Media group, http://tangible.media.mit.edu) are interesting possibilities for extending the range of interface tools for large-screen devices beyond windows, icons, mouse, and pointer (WIMP).

Some early work (see for example, Cinematrix, http://cinematrix.com; HI-Space, http://www.hitl.washington.edu/projects/hispace/; and Roomware, http://www.ipsi.fhg.de/ambiente/english/projekte/projekte/roomware.html) investigated alternate user interfaces that work with groups of users and multiple large-screen devices.

Integrating graphics and user interaction

Current midsize devices let users efficiently and effectively control and manipulate 2D, video (that is, a collection of graphic images displayed one after another fast enough to give the viewer the illusion of motion), and 3D graphic images displaying user data in addition to user interface objects. Current large-screen devices let graphic images communicate effectively. A significant amount of work is needed to expand the interaction tools available to a single user and to make the devices more effective for colocated groups. Promising work is ongoing in both areas. The next logical extension is interactive techniques to improve user efficiency.

Similar work for small-screen devices is lagging. Basic interaction is limited to text and forms and excludes interaction with graphic images. Part of this can be attributed to the lack of interactive tools for entering text and coordinate positions. However, a more fundamental problem exists: It's difficult to understand how to shrink or partition a 2D, video, or 3D graphic image while retaining communication efficiency.

References

1. K.R. Boff, L. Kaufman, and J.P. Thomas, eds., *Handbook of Perception and Human Performance: Sensory Processes and Perception*, vol. 1, John Wiley & Sons, 1986.
2. D. Kasik et al., "Evaluating Graphics Displays for Complex 3D Models," *IEEE Computer Graphics and Applications*, vol. 22, no. 3, May/June 2002, pp. 56-64.

community extends far beyond company boundaries. For example, airline personnel buy the product, perform maintenance, and order spare parts. Partners design and analyze significant airplane components. Suppliers formulate proposals to build many of the parts. Government agencies determine if the final product performs according to published regulations and guidelines.

Although current practice still generates a lot of paper, the user communities rely on computer graphics in 2D, video, and 3D. A wide variety of visual tools developed both internally (such as FlyThru) and externally (such as Dassault Systemes' Digital Mock-up Utility) is available to users. The tools can show or hide objects, apply texture, identify spatial position, generate multi-

ple concurrent views, and so on. None of the tools fundamentally change the style in which the image is rendered.

Once a user has an image and a set of visual tools, he or she can extract a significant amount of information using visual analysis techniques. Users can perform a wide variety of tasks using visual analysis alone. By observing users at work and consulting with different groups of users, I developed a set of typical tasks that can be performed using visual analysis.

First, find an object in a complex scene given

- the physical object,
- a picture of the object,
- a mental image of the object, or
- a verbal description of the object.

Second, focus on the found object to better understand surface characteristics (smoothness or roughness, for example). A user can determine characteristics by visually inspecting the object or by interpreting its physical characteristics (such as the effects of aerodynamics flow or stress).

Next, look at objects in the immediate surrounding volume to

- identify discrepancies in space consumption. (Do the pieces occupy too much of the same space?)
- determine interference and overlap. Whether you do this through direct visual inspection or by recognizing the results of computing overlaps in a batch mode, you must perform a visual analysis to determine whether the interference is acceptable.
- find gaps or voids between objects (that is, ensure proper clearance). A method to measure distance between objects often supplements visual gap analysis.
- trace a path that connects objects. In an airplane, these connections are long, skinny things like hydraulic tubes and wire bundles.

Next, visually scan the scene to find

- misplaced objects,
- forgotten objects ("Drat, we forgot a wing!"),
- patterns from similar objects or members of a family of parts,
- objects that might not be in a production configuration (debugging or placeholder objects, for example),
- maintenance accessibility or manufacturing assembly problems, or
- a part's conformance to the design. For example, you must periodically examine tools that are in the field to determine needed upgrades.

The next task involves observing scene dynamics (typically by animation) to

- recognize dynamic interference conditions (such as display results from kinematics or mechanisms analysis, vibration, or tolerance buildup),
- follow system flow (fluid flow in hydraulic tubes, for example),
- detect effect of loads, aerodynamic flows, and so on over time, or
- receive instructions about assembly (for manufacturing) and disassembly and reassembly (for maintenance) sequences.

All these tasks assume a single window displaying one style. It's also useful to compare multiple versions of the same set of objects for

- subjective preference (for example, "I like the way that car's hood reflects the lights on the showroom floor") or
- net version change.

Boeing's user community has reasonable access to devices with small, midsize, and large screens. Users who need graphic information delivered directly to a job site are increasingly interested in small devices. These users perform only a subset of the visual analysis tasks listed above: they find objects in complex images, trace paths, check design conformance, follow flows, and understand assembly/disassembly/reassembly sequences and documentation.

Engineers with access to midsize and large screens, generally in an office environment, perform most other tasks during product design and analysis.

Supporting the thousands of Boeing users with small-screen devices requires (at a minimum) partitioning graphic images into meaningful chunks.

Seeing and understanding images

Substantial research examines how people actually see and perceive the physical world.[2] Other research[3] has determined how light is transmitted into the eye and what areas of the brain process the signals.

When a person starts navigating graphic images, additional brain processing occurs. Siegel and White differentiate navigation strategies (landmark, route, and survey) through landscapes[4] and show that such strategies are hierarchically related.[5] Other experiments show that navigation strategies might not be hierarchical; rather, they depend on whether the user recognizes the landscape in a scene.[6]

Scientists don't perfectly understand the cognitive processes a person uses to identify an image (for example, "Is that image an engine or a wing?"). Biederman has developed a reasonable theory[7] that describes a four-stage mental process. A person recognizes a 3D shape during the first three stages, and he introduces the notion of generic *geometric ions* (geons). The fourth stage determines the 3D shape's meaning and doesn't have a corresponding generic basis because the meaning depends on a person's culture and background.

Partitioning strategies

To develop effective partitioning strategies for small screens, a designer or image author must consider the areas discussed previously: limitations of small screens for the images used in complex products, tasks involved in visual analysis, and the fundamentals of human visual processing.

A partitioning strategy for the types of graphic images common in product development must preserve as much of the images' communication impact as possible. I applied Massironi's more general graphic communication techniques[1] to develop several principles for partitioning graphic images:

- Graphic images that don't require dimensional accuracy (such as some technical drawings) can be subtly changed. People recognize many objects even when they are subtly redrawn—for example, the 26-character alphabet is recognizable in many fonts.
- Cultural background impacts communication. Visually changing a font in the English alphabet doesn't communicate completely to people who only read Chinese or Greek.
- Users gather a substantial amount of information from an image's general context.
- A complex image, the kind routinely used in companies like Boeing, forces viewers to scan multiple zones to identify the most interesting zone. The image author can graphically enhance areas to attract initial interest.
- Graphic images have natural edges that define structure.

Designers can use any of the partitioning strategies to adapt graphic images to small screens. I haven't found a documented systematic approach to the partitioning problem.

Partitioning strategies differ from technologies that actually break an image into smaller parts. For example, a now defunct start-up company, Newstakes, used image processing to break images into multiple areas. However, Newstakes users had to know the area's size, which area to display first, and so on.

The most straightforward solution to the problem is to treat a small screen as you do any other screen: display the entire image (2D, video, or 3D) and give the user full control of viewing transformations. If adequate network bandwidth and local device memory are available, the problem becomes providing a user interface that allows extremely fast interaction with local controls that consume only a small amount of screen space.

Figure 2 shows a typical set of controls for an engineering drawing display intended for a midsize screen. When squeezed to the size of a typical PDA screen, the user controls alone dominate the display. The display gives no context, one of the key principles in graphic image communication. Other strategies subdivide the picture into discrete tiles, which helps the user overcome screen size, device processing power, and network bandwidth limits.

2D images

Millions of 2D images document essential information for existing Boeing airplanes. The vast majority are technical drawings consisting of lines, curves, and text, and were intended to be delivered on nothing smaller than letter-size paper.

A typical PDA screen is 2.25 × 3.0 inches, with a pixel resolution of 240 × 320. Both measurements affect partitioning strategies.

The most basic display technique is to draw the whole image in a single tile, as Figure 3 shows.

The graphical viewer provides navigation tools for panning and zooming. The user can figure out where to start based on the general graphical forms on the display device. Zooming can be problematic for raster images because the picture quality degrades as the image is magnified. Vector forms (such as computer graphics metafile [CGM] or scalable vector graphics [SVG]) retain better image quality because the raster image is regenerated at each step of the magnification process.

Most conventional pan tools use sliders to control *x* and *y*; zoom uses a magnifying glass or numeric value. Both are necessary for images such as Figure 3, where without them users can't read the text.

Conventional pan and zoom tools, however, rarely provide context. Bier et al. describe additional zoom techniques that led to the Magic Lens implementation.[8] Other context-preserving zoom tools are available, such as the Idelix Pliable Display Technology (http://www.idelix.com).

Partitioning strategies for 2D images. Some 2D graphic image-partitioning strategies split the initial image into a set of tiles. In multiple tiles,

- individual tiles should be of equal size for cross-tile navigation. The user initially receives a single tile at a higher zoom level than the image shown in Figure 2. Users can often understand information in the first picture, which avoids having to transmit the entire image across a low-bandwidth network to a limited-memory device.
- text, especially in technical drawings, represents a difficult challenge for any tiling scheme. The rule of thumb is that a text string with no blanks should stay on a single tile. If a string won't fit at an acceptable scale, the best strategy is to send a tile that's larger than the screen and give the user pan and zoom tools.
- each tile should slightly overlap its neighbors. This keeps context as the user navigates to new tiles and facilitates path following.
- any tiling scheme will occasionally fail. Reverting to a single tile containing the entire image (under either application or user control) is an acceptable risk-management technique.

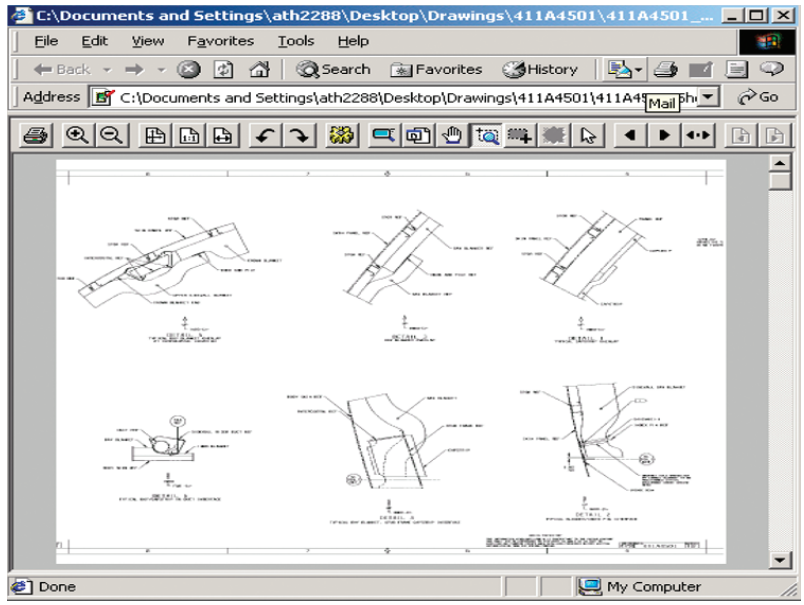


Figure 2. Typical engineering drawing and controls.

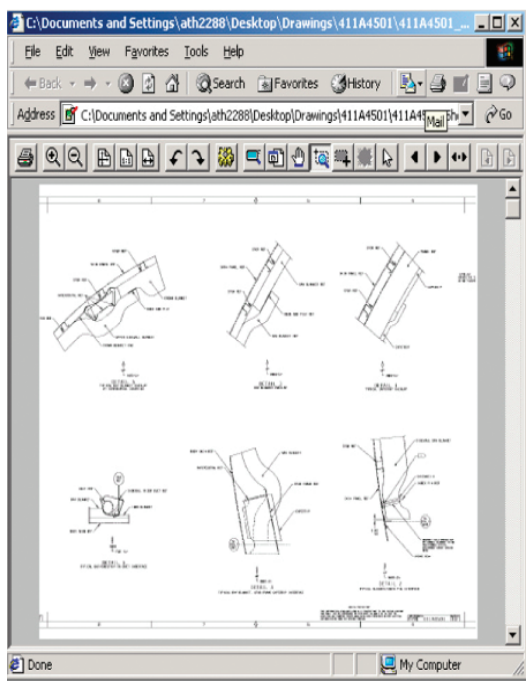


Figure 3. Engineering drawings redrawn for a 2.25″ x 3.0″ screen.

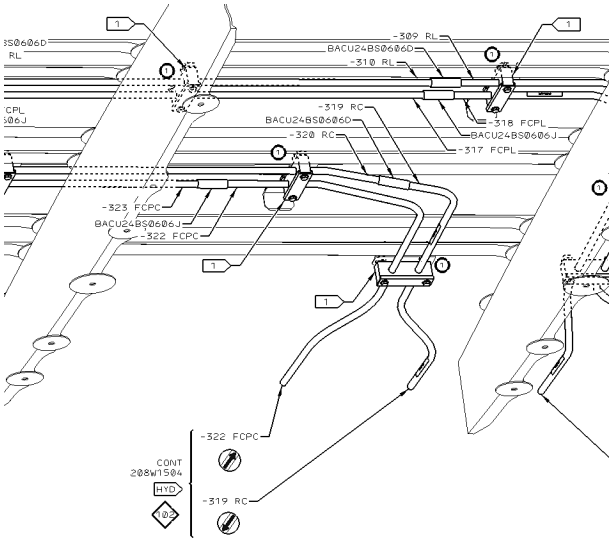


Figure 4. Set of assembly routing instructions for hydraulic tubing. The image author draws viewers' attention to several areas by marking them with a 1 in a bold-type circle.

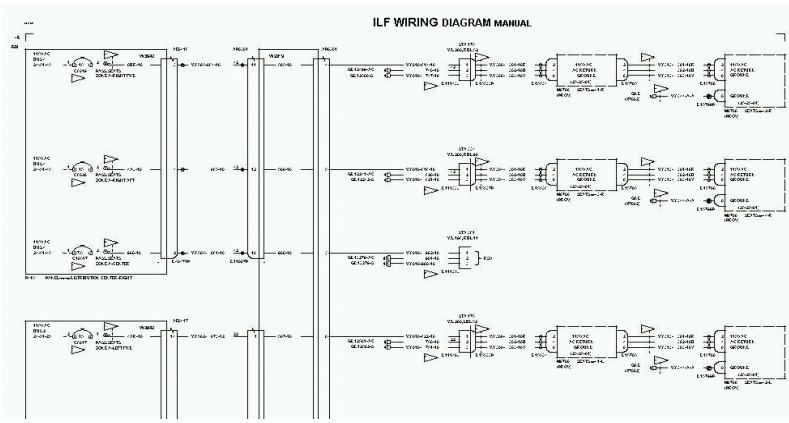


Figure 5. Sample wiring diagram. The long horizontal and vertical lines indicate the image's tile structure.

I've developed four strategies based on these principles.

Strategy 1: Author-defined visual targets. Authors of graphic images, especially complex images, typically use heavyweight lines or bold or italicized text to draw attention to a specific region, such as the bold borders around the number 1 in Figure 4.

The strategy should identify as many attention-getting landmarks as possible. If there's only one clear landmark on an image, it should be the first tile the user sees. Otherwise, the strategy should use a consistent heuristic to determine which highlight to show the user first (for example, the first tile shown contains a landmark that's above and to the left of other tiles with landmarks).

Strategy 2: Follow long horizontal and vertical lines. Not all graphic images contain attention-getting cues. In the wiring diagram in Figure 5, for example, users can infer the tile structure from the long horizontal and vertical lines.

After strategy 2 determines the basic tile size, strategy 1's results will identify the first tile to be displayed. If the image author didn't provide a specific landmark, the display strategy should use a consistent heuristic (for example, always draw the tile in the upper left corner first) to start.

Strategy 3: Follow long horizontal and vertical white space. Figure 3 shows a different technique for visually finding blocks in a complex graphic image. In this figure, long horizontal and vertical white spaces define the tile edges. White space clearly delineates interesting areas that we can use as tiles. As in strategy 2, if the image author didn't establish a landmark, the strategy should use a consistent heuristic to determine which tile to display first.

Strategy 4: Follow long, arbitrarily sloped lines. Another alternative when the author doesn't provide landmarks is to supplement the techniques from strategy 2 (long horizontal and vertical lines) with the identification of long lines with arbitrary slopes. Partitioning tiles along long lines facilitates path following. Again, the strategy should determine the first tile to display in a consistent manner.

Cross-tile navigation. These four strategies will likely result in tiles that are slightly larger than the screen to accommodate nonblank text strings.

Screen area is a significant constraint when providing context and navigation assistance for a graphic image that's split into multiple tiles. As noted previously, pan and zoom tools for an individual tile can consume a significant amount of screen space, although lens tools help in that situation.

One possible approach to cross-tile navigation lets users select from a list of tile names. This forces the user to make an extra selection, complicating path following. The key to providing some level of context-preserving navigation is to include navigation aids in each tile so a user need only take one action to move to the next tile.

A more straightforward technique is to use a set of dynamically computed navigation arrows, as Figure 6 shows. Each arrow sits at the physical edge of the screen.

Users can select the arrows, which indicate additional tiles in each of eight directions. If a computer program inserts arrows as it sends each

tile to the device, absence of an arrow indicates that navigation has taken the user to an edge or corner. This technique gives the user a sense of the image's size and a limited sense of context.

Intelligent Zoom[9] is an alternate technique initially used to shrink wall-sized displays of electric power grid control systems. Intelligent Zoom overlays a map showing the tiles composing the image. When a user selects a tile, the other tiles shrink and the camera seems to enlarge the selected tile. This technique generally consumes less screen space than conventional 2D navigation techniques and provides good visual context. However, the user must make an extra selection to move between tiles.

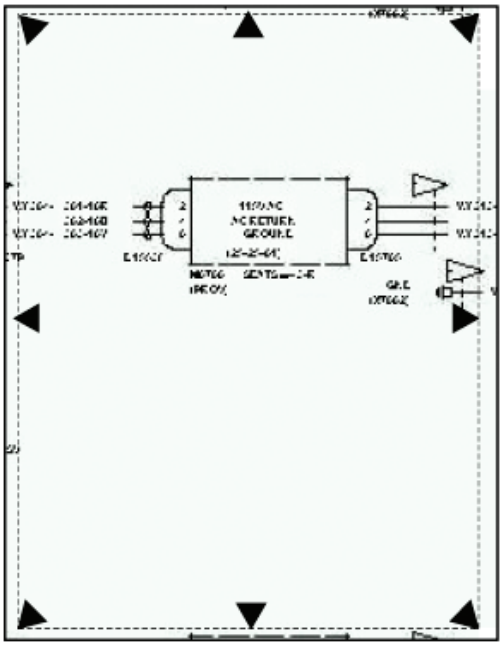


Figure 6. Navigation arrows on a tile.

Video

Many sources of video exist, from Flash animations to collections of camera-captured raster images. Although not required, sound generally accompanies video sequences.

Thus, video affects two senses. In terms of small screens, designers must balance the screen-size problem with the projection-rate problem. Time dominates screen resolution because the motion (both picture and sound) conveys critical information. Local device capacity and network performance drive the strategies more than maximizing communication impact.

Using strictly local device resources for storage and playback provides the most consistent video playback rate. However, device resource limitations make streaming necessary for the foreseeable future. Streaming can be successful as long as network bandwidth supports acceptable video playback rates. Given current and foreseeable variation in network rates (rates change during a playback session), there are four strategies that vary how a source sends individual frames during a single playback session. In other words, if the bandwidth is good enough, send a full resolution picture. If bandwidth degrades, use one of the following strategies to change the image quality.

Strategy 1: Make pixels bigger. In this technique, often used with videophones, the source sends fewer but larger pixels on a frame-by-frame basis. The number of pixels transmitted increases or decreases as network performance varies. For relatively unchanging pictures, some implementations transmit only deltas.

Strategy 2: Draw visual targets at high resolution. Strategy 1 for 2D images notes that authors of graphic images, especially complex images, leave hints about what's most important in the scene. The pixel budget for a single image can be spent in another way: give greater resolution to graphically important areas and less to others.

Strategy 3: Show full frames at fixed intervals. Not every video frame must be visible. The easiest technique is to change the frame rate so that it shows only every *n*th frame, but at full resolution. As bandwidth changes, *n*'s value changes.

Strategy 4: Show key frames. Video relies on intraframe coherence to create the illusion of motion. This means that some frames will be highly similar to preceding frames. A big change from one frame to another indicates a new sequence. By transmitting only key frames, the images have less movement but still communicate basic information.[10]

Degrading sound can help manage variable network speeds. Although voice alone can be slowed down (think of controls on voicemail systems), music is often part of the streaming media. A general strategy dedicates enough available bandwidth to keep the audio playing at a normal rate and changes the graphic images to fit into the remainder. People can more easily adapt to image changes and still obtain most of the information from them.

3D images

Boeing defines its airplanes and many other complex physical products in three dimensions. Most of these products are physically larger than

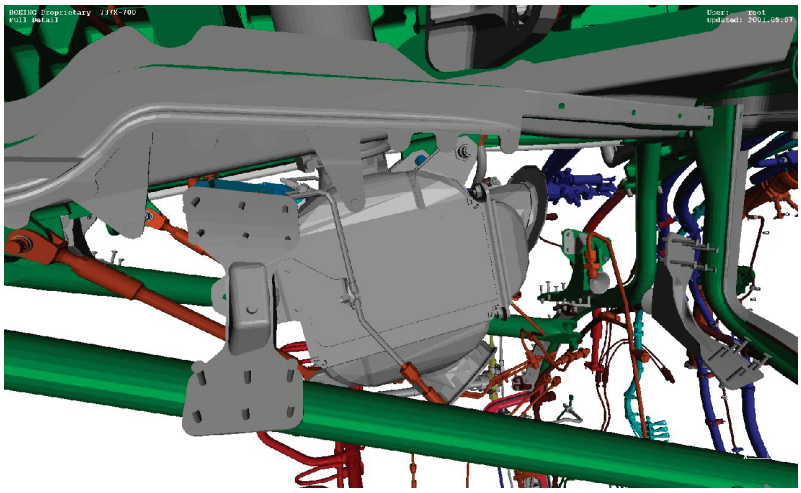


Figure 7. Typical 3D image of airplane structure.

any screen. Figure 7 contains a typical image from a Boeing commercial airplane.

These images fit in Massironi's descriptive geometry category. Computer users performing visual analysis tasks can use interactive techniques to help them understand the data's 3D nature. At a minimum, the interactive techniques give a user control of viewing transformations (scale, rotate, and translate). Computing these operations on the local device keeps performance consistent.

If implemented, the tiling strategies described for 2D images become rectangular solid subdivision strategies. Voxelization techniques can help determine the solid subdivisions. Understanding how voxels relate to their neighbors is a difficult cognitive task, however, because the neighbors are in 3D, not 2D. It's also difficult for users to navigate to a neighbor. Rather than the eight neighbors in 2D, in 3D they have 26.

Thus, the most effective way to deal with 3D on a small screen is to rely on user-controlled viewing transformations and standard techniques for showing context (displaying coordinate axes representing current orientation or thumbnail views, for example).

Implementation and challenges

When implemented, image partitioning strategies help users deal with the screen-space and resolution limitations of small screens. However, companies like Boeing currently have millions of digital graphic images, most of which were designed for midsize to large sheets of paper or display screens. In addition, authors of video sequences for training or assembly and maintenance instructions assume that frame rate will be adequate to depict motion. Manually modifying either form is impractical.

Thus, we need automated techniques to break 2D graphic images into tiles and analyze video frames to adapt to varying network performance levels. Newstakes provided a nontraditional use of image-processing algorithms to address both problems.

Automatic 2D partitioning needs methods to recognize characters (at a minimum, to break on blanks), find visual targets, and identify regions bounded by lines or white space.

A Boeing-Newstakes feasibility study sought to determine whether image processing was practical for images in the wiring diagram class shown in Figure 5. The study demonstrated feasibility but ran out of funding before applying the technology to other types of technical images, and found no general, published strategies resulting in high-quality partitions. Newstakes image-processing control algorithms were cleverly implemented and recognized the rectangular regularity of wiring diagrams. However, the study didn't find a good way to identify the first tile to be displayed.

The strategies I've presented should facilitate development of image-processing partitioning and initial position heuristics for all image classes.

Effective automation tools address a significant percentage of the legacy data problem. Individual window managers, applications, browsers, and browser plug-ins implement local pan and zoom tools inconsistently, and few provide an indication of context. This adds to user confusion, and additional work is needed to provide a more consistent set of user navigation tools. Even more variation exists for local 3D scene navigation.

Finally, I've focused on users who only look at the graphic images. Another large segment of the user community directly interacts with objects within the graphic image. Providing effective graphic selection tools for small screens might require different solutions from those currently implemented for larger screens.

Conclusion

The ability to push well beyond the perceived capabilities of a computing device characterizes all user communities. The world of graphics devices has expanded dramatically over the past few years, and all signs point to an even more dramatic expansion in the next 5 to 10 years.[11]

This article starts to address the difficult prob-

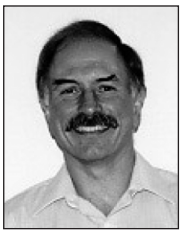
lem of finding acceptable methods to deal with screen-size variations. Instead of approaching the problem from a technology perspective, my techniques preserve graphic communication impact for specific visual analysis tasks without extensively modifying existing images or overly constraining authors of new images. Adapting visual content to retain graphic communication impact is essential to maximize the effectiveness of new devices.

Acknowledgments

Forrest Warner and Jim Ginn of Boeing provided invaluable assistance in building the inventory of visual analysis tasks. Bill Buxton of the University of Toronto collaborated in developing Figure 1. John Dill of Simon Fraser University and Brian Fisher of the University of British Columbia gave me pointers to neurophysiological literature. Gurminder Singh of the Naval Postgraduate School introduced me to Newstakes technology. Terry Porlier helped solidify early drafts of this article. The technical reviewers and IEEE editors gracefully pointed to areas that let me substantially improve the writing quality. **MM**

References

1. M. Massironi, *The Psychology of Graphic Images: Seeing, Drawing, Communicating*, Lawrence Erlbaum Assoc., 2002.
2. J.J. Gibson, *The Perception of the Visual World*, Houghton Mifflin, 1950.
3. D.H. Hubel and T.N. Wiesel, "Brain Mechanisms of Vision," *Scientific Am.*, vol. 241, no. 3, Sept. 1979, pp. 150-162.
4. A.W. Siegel and S.H. White, "The Development of Spatial Representations of Large-Scale Environments," *Advances in Childhood Development and Behavior*, H.W. Reese, ed., vol. 10, Academic Press, 1975, pp. 9-55.
5. P.W. Thorndyke and B. Hayes-Roth, "Differences in Spatial Knowledge Acquired from Maps and Navigation," *Cognitive Psychology*, vol. 4, 1982, pp. 560-589.
6. V. Aginsky et al., "Two Strategies for Learning a Route in a Driving Simulator," *J. Environmental Psychology*, vol. 17, 1997, pp. 317-331.
7. I. Biederman, "Recognition-by-Components: A Theory of Human Image Understanding," *Psychological Rev.*, vol. 94, no. 2, 1987, pp. 115-147.
8. E.A. Bier et al., "A Taxonomy of See through Tools," *Proc. Computer–Human Interaction* (CHI 94), ACM Press, 1994, pp. 358-364.
9. L. Bartram et al., "The Continuous Zoom: A Constrained Fisheye View Method for Visualizing and Navigating Large Information Spaces," *Proc. User Interface Software and Technology*, ACM Press, 1995.
10. V. Vasudevan, G. Singh, and M. Jesudoss, "Creating a Slide Show Presentation from Full Motion Video," US patent 09/215,004, Patent and Trademark Office, Washington, D.C., 1999.
11. S. Ortiz Jr., "New Monitor Technologies Are on Display," *Computer*, vol. 36, no. 2, Feb. 2003, pp. 13-16.



David J. Kasik is the Boeing Commercial Airplanes (BCA) Information Systems architect for software engineering, geometry and visualization, and user interface technology. He's currently working to understand the impact of large-screen devices on a large group of engineers and the usefulness of small-screen devices for untetherable factory users. Kasik has a BA in quantitative studies from The Johns Hopkins University and an MS in computer science from the University of Colorado. He's a member of the ACM, ACM Siggraph, ACM Sigchi, and the IEEE.

Readers may contact the author at Boeing Commercial Airplanes, PO Box 3707, Seattle, WA 98124; david.j.kasik@boeing.com.

For further information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.

Editor: Frank Bliss

David J. Kasik
The Boeing Company

William Buxton
Buxton Design

David R. Ferguson
DRF Associates

Ten CAD Challenges

We discuss the significant technical challenges facing the CAD industry and research community and present an approach to address these issues.

The world is enamored with the number 10. Perhaps it's because arithmetic is a lot easier when the number 10 is part of an operation. Or maybe David Letterman and many others have finally achieved a long term impact on society with their top and bottom 10 lists.

Ivan Sutherland used the number 10 to bring significant consistency to hidden surface algorithms for computer graphics in his classic paper.[1] It's in honor of Sutherland—developer of the original Sketchpad application, the forerunner of today's computer-aided design (CAD) applications—that we describe our 10 CAD challenges. The concept of CAD itself has expanded into computer-aided manufacturing (CAM) and computer-aided engineering (CAE). Basic CAD techniques are reapplied in a number of places, including electrical and mechanical product development, buildings, and entertainment. In this article, we focus on mechanical product development.

CAD, CAM, and CAE generate massive amounts of data that must be clearly organized and placed under strict configuration control. The CAD industry also provides support to these functions through product lifecycle management (PLM) systems. In assessing the state-of-the-art CAD, we're examining a reasonably mature, respected, and accepted form of technology. The industry's multibillion dollar yearly revenues have been reasonably flat for the past few years. For an introduction to CAD's history, see the "Brief CAD History" sidebar.

We could place our 10 challenges in a number of different categories. We've chosen three: computational geometry, interactive techniques, and scale. Given our knowledge of the state of the art, the categories we would have chosen 10 years ago would likely be different. Each of the categories presents an opportunity to expand CAD's penetration to new user communities and increase its long term impact.

Geometry represents the kernel data form that a designer uses to define a physical product in any CAD application. Correct assembly of the geometric shape, structure, and system components are the basis for figuring out how to produce the complete product (through CAM) and how the product will respond once in service (through CAE). Only a few commonly used libraries implement computational geometry algorithms, an indication of the technology's relative maturity. Our analysis identified three specific geometry challenges:

- geometry shape control;
- interoperability across CAD, CAM, and CAE applications; and
- automatically morphing geometry in a meaningful way during design optimization.

The key component of all CAD applications is the end user. Users' success or failure in interacting with CAD products ultimately governs the success of the products themselves. We extend the meaning of *interactive techniques* for this article to include more than low-level input device, display, and human factor issues. We deliberately include the way users work with the application to accomplish a work task. Using this extension lets us define three difficult challenges with interactive techniques:

- the order and flow of the tasks a user performs to accomplish something,
- the relationships among tasks in a complex design environment, and
- the manner in which old designs can effectively seed new ones.

The third category describes the challenge of scale. In many ways, the CAD market has reached a plateau just because it has not yet discovered ways of going beyond its current limits. We introduce scale challenges that we have observed as the definition of the current limits of CAD penetration. Should these challenges be addressed in a meaningful way, larger growth in CAD

© 2005 IEEE. Reprinted, with permission, from IEEE Computer Graphics and Applications.

Published by the IEEE Computer Society 0272-1716/05/$20.00 © 2005 IEEE

business will likely occur. The four scale challenges include

- finding ways to cope with vastly larger quantities of data,
- communicating key concepts to user communities traditionally outside the CAD community,
- reliably migrating data to new versions of software and hardware as product life spans increase, and
- improving productivity for geographically distributed project teams.

As we built this set of challenges, we realized that the sum of the challenges was greater than we expected. Therefore, we suggest an approach that can help others understand and manage the changes needed. We believe that the technical challenges we've identified will lead to fundamental changes in the way people work.

Geometry

Geometry lies at the core of all CAD/CAM/CAE systems and, while not the only item of interest in product design and development, it's the sine qua non of design. Geometry includes both the algorithms and mathematical forms used to create, combine, manipulate, and analyze the geometric properties of objects: points, lines (curves), surfaces, solids, and collections of objects.

Geometry begets three fundamental questions: What are the objects to be represented? What mathematical forms and approximations will be used to represent them? How will information about the representation be computed and used?

Given the power and generality of current design systems, we can imagine that all issues related to these questions had been adequately answered. However, this is not the case. We discuss three geometry challenges (shape control, interoperability, and geometry in design exploration) that still require extensive research and development.

History

To illustrate the evolution of geometry methods and usage we focus on five time periods: pre and early 19th century and early, mid, and late 20th century. In each of these periods there was a fundamental shift in the methods and usage of geometry driven by new design requirements.

Prior to the 19th century, the major use of geometry was to define and maintain line drawings for manufacturing and records keeping. The tools and methods were rudimentary: lines were constructed by ruler and compass methods and curves were traced from a draftsman's or loftsman's spline, a thin wooden or metal strip bent and held to a desired shape using weights (see Figure 1).

In the early 19th century, these methods were augmented with descriptive geometry, primarily using second degree algebraic equations, to provide more precise mathematical descriptions and increase accuracy and precision in engineering drawings. The objects of interest were still lines, but descriptive geometry required new tools (for example, slide rules and tables) to aid in calculating various curve properties (such as point location). Descriptive geometry changed from curves drawn on paper to precise mathematical formulas from which any point on a curve could be calculated accurately.

The next major advance took place early in the 20th

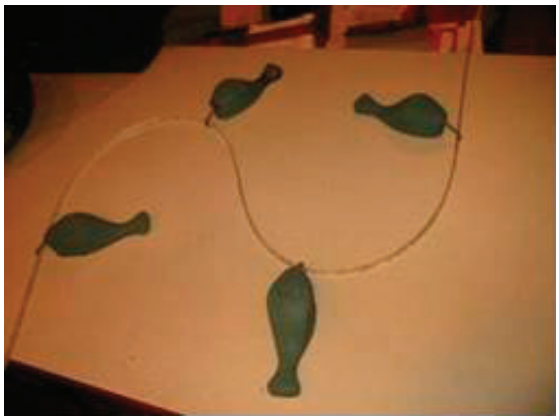
Brief CAD history

CAD was one of the first computer graphics applications in both academia and industry. Ivan Sutherland's Sketchpad at the Massachusetts Institute of Technology and the DAC-1 project at General Motors both started in the early 1960s. Industry developed its own CAD applications, delivered on multiuser mainframes, in the late 1960s and 1970s. The 1980s turnkey systems bundled hardware and software. Current CAD implementations separate hardware and software components. As a result, CAD software most often executes locally on powerful Unix or Wintel workstations with specialized 3D accelerator hardware and stores data on distributed servers.

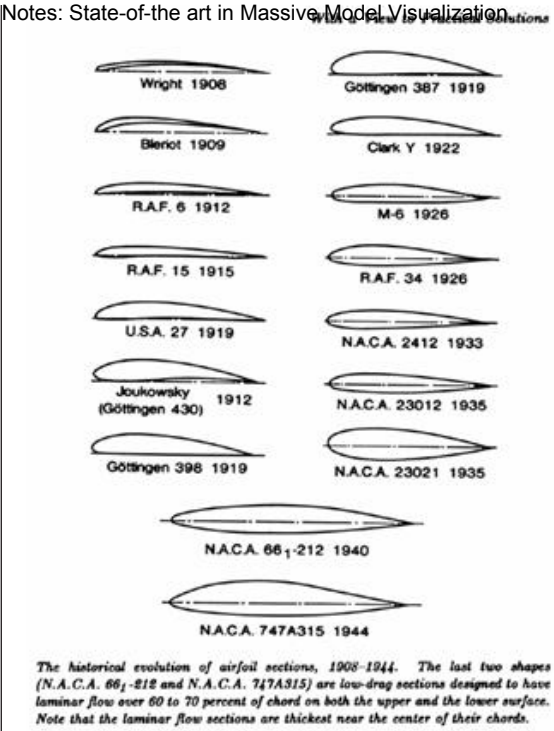
Even though engineering drawings were the dominant output of now defunct CAD vendors—for example, Lockheed's CADAM and commercial companies like AD2000, Applicon, Autotrol, ComputerVision, Gerber IDS, Intergraph, Matra Datavision, and VersaCAD—through the 1980s, early stages of design relied on a variety of unique curve and surface forms. Customized systems were used in aerospace for surface lofting (such as TX-95 at Boeing) and surface design (such as CADD at McDonnell-Douglas) and in automotive industry for surface fitting—such as Gordon surfaces (General Motors), Overhauser surfaces and Coons patches (Ford), and Bezier surfaces (Renault). Current CAD applications rely on more general geometric forms like nonuniform rational b-splines (NURBS).

Solid modeling also started in the late 1960s and early 1970s but from different roots. Larry Roberts worked at MIT to automatically identify solids from photographs. The Mathematics Application Group, Inc. used combinatorial solid geometry to define targets for nuclear incident analysis and subsequently developed ray traced rendering and solid modeling in Synthavision. Other efforts (for example, TIPS from Hokkaido University, Build-2 from Cambridge, (Part and Assembly Description Language PADL) from University of Rochester) had limited industrial impact.

Solid modeling is the 1990s preferred technique for defining 3D geometry in small through large companies, and good modeling software is readily available. The use of 3D is common in computer animation, aerospace, automotive, (and is becoming acceptable for building architecture), and 2D is also used effectively.



1 **Early meaning of spline weights. (Courtesy of Dave Ferguson)**



2 NACA Catalog of Airfoil Shapes. (Courtesy US Centennial of Flight Commission)

3 P51 Mustangs.

century. Catalogs describing functionality and application of particular families of curves in engineering began appearing. The catalogs were based on careful scientific analysis and engineering experimentation. For example, the National Advisory Committee for Aeronautics (NACA) generated a catalog of curves that classified airfoil shapes according to flow properties (see Figure 2 and http://www.centennialofflight.gov/essay/Evolution_of_Technology/airfoils/Tech5G1.htm and http://www.centennialofflight.gov/essay/Evolution_of_Technology/NACA/Tech1.htm). Objects became more than curves; they also had attached properties that described their appropriate use in various design activities. Design could now begin with curves known a priori to have properties needed for a viable design.

In the 1940s, Roy Liming at North American Aircraft introduced actual surface representations in his conic lofting system.[2] These were not like today's surface representations, but his system did provide for complete surface definitions rather than simply a family of lines. It was now possible to think of mathematically computing mass, aerodynamic, and hydrodynamic properties. At this point, geometry began changing from describing a physical object to becoming the base for calculating engineering characteristics. This lessened the dependence on physical models and began the push toward virtual design. Tools also began changing. With the increased emphasis on analysis, calculators and computers became required tools.

Liming's conic lofting methods proved their worth in the design of the P51 Mustang airplane (see Figure 3). Liming boasted that the Britain-based Mustangs could fly to Berlin and back because their surface contours did not deviate from the mathematical ideal.[3] However, it was not possible to visualize the geometry without a physical prototype or mock-up.

Modern graphics systems made it possible to display 3D geometry. Designers could visually inspect designs to find mismatched or ill-fitting parts and shape defects. The new display technologies required radical change in how geometry was represented: Everything in a design had to have a precise mathematical representation. Gone were the line drawings used by draftsmen. The old algebraic methods of descriptive geometry were displaced by mathematical splines, NURBS, tensor product splines, and other analytically based forms.

Having a complete mathematical description brought another fundamental change. Previously, the preferred method of constructing a curve or a surface was highly intuitive: lay out sequences of points; construct curves that pass through the points; and then construct a surface from the curves by cross-plotting, conic lofting, or some other means. Using purely mathematical algorithms to interpolate or approximate opened up new possibilities: Curves could be defined by approximating a sequence of points, surfaces could be defined by clouds of points, and geometry could be constrained directly to satisfy engineering constraints (for example, clearances and shape).

Geometric methods and objects used to support product design and definition have changed significantly. As design objectives and systems continue to respond to the ever-increasing need to design rapidly, efficiently, and virtually, geometric methods will need to meet the challenges and change accordingly.

Challenge 1: Shape control

The success that graphics had in forcing everything to have a precise mathematical representation actually increased concern over inflections. The hand-drawn and hand-constructed methods, including Liming, had implicit control of shape whereas the new, polynomial and piecewise polynomial methods (splines, B-splines, and so on) did not. Inflections in a curve passing through a sequence of data points are possible using polynomi-

als or piecewise polynomials even though the data points do not suggest inflections. Therefore, it became important to devise algorithms that not only fit points but also allowed shape control. Shape control defines the occurrence and position of inflection points (points at which the signed curvature of a planar curve changes). The control is needed for both engineering and manufacturing optimization.

Researchers have proposed various schemes for shape control. Most failed because there were always cases for which the methods failed to properly preserve shape. Many methods exist for detecting shape anomalies after the fact, and a person must fix the anomalies by hand. Removing the person in the loop lets geometry pass directly to other applications for optimization. The challenge becomes finding algorithms that avoid anomalies in the first place.

Attempts to modify the methods used previously to account for shape control did not work in general, and it wasn't until the 1980s that we realized the basic principles of those methods were wrong.[4] Although nonlinear methods[4] have proven themselves in a variety of shape control situations, most of these methods have not made their way into commercial CAD systems.

Challenge 2: Interoperability

Current CAD systems do not integrate well with CAE analysis (such as structural mechanics, fluid dynamics, and electromagnetics).[5] For example, computational fluid dynamics (CFD) must interrogate geometry quickly and reliably. Most CFD codes construct a computational grid from the geometry. Building the grid reliably means that there should be no unintended holes in the geometry—that is, the geometry should be what CFD practitioners refer to as watertight. Real geometry from CAD systems is rarely watertight.

Geometry from one CAD system is difficult to translate reliably into another. Estimates peg the cost of interoperability in the US auto industry at $1 billion per year.[6]

Holes, translation error, and other problems arise from two major sources: floating-point arithmetic and tolerances. Floating-point arithmetic, which forces approximations in numerical calculation, is addressable theoretically but not practically. We could impose higher precision (double, triple, and so on) to drive down the resulting errors. Or we could change to a rational arithmetic system and eliminate the need for floating point. Digital floating-point arithmetic is a research area by itself.[7]

Tolerances control the accuracy of computed solutions and are a fact of life in today's CAD systems. A simple example involves calculating the curve of intersection between two surfaces. When the algebraic equations representing the geometry are simple (for example, a plane or a circle), a closed form solution for the intersection exists. However, closed form solutions generally do not exist for operations on equations of a sufficiently high degree (for example, intersecting two bicubic surfaces). Computing the intersection curve uses approximation, a problem independent of precision. Some CAD systems will recompute intersection curves if more accuracy is needed. This doesn't solve the problem, especially if the intersection curve is used to generate other geometry.

Tolerances are needed to control the approximation.[8] Too loose a tolerance can give results that are fast but incorrect. Too tight a tolerance can result in poor performance or failure to converge. Even seemingly simple surface-to-surface intersections become difficult because of choosing tolerances.

Tolerances determine the success of downstream engineering (CFD, finite element) and manufacturing (numerical control programming, quality assurance) analyses. Selecting a tolerance that guarantees a high probability of success requires that the geometry generator understand the kinds of analyses to be employed, the environment of the analyses, and even the specific software to be used a priori.

In summary, digital arithmetic and current math theory are insufficient to perform reliably for complex geometry operations and to interoperate well with downstream analysis software. The geometry must be as watertight as possible for downstream use, and algorithms cannot result in topological inconsistencies (for example, self-intersections and overlaps). The challenge is to find ways to deal with poor results. Perhaps a new math theory that has closed-form solutions for complex surface operations and supports watertight representations for downstream analysis is the way to address this challenge.

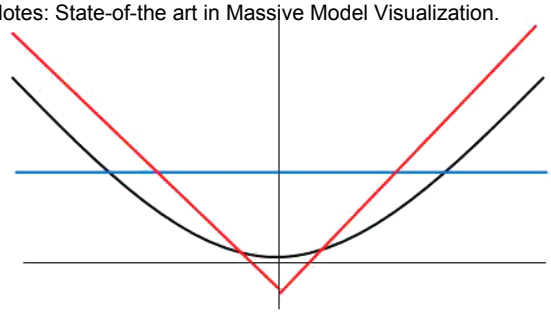
Challenge 3: Design exploration

Automated design exploration through multidisciplinary optimization presents the third challenge. Design optimization requires that geometry remains topologically valid as parameters are perturbed while preserving the designer's intent. There are two aspects to consider: how to parameterize the geometry for downstream analysis and how to structure geometry algorithms to support continuous morphing, a key to any optimization process. The former is primarily an engineering function, which we do not discuss here.

Morphing is a requirement that CAD systems do not currently support. Morphing algorithms today allow the hole in the upper block to flip into the lower block when the edges of the two blocks align. This is fine geometrically. However, this is a disaster for optimization, because the geometry does not morph continuously with the parameters.[9]

The challenge is to design and build geometry systems that ensure the continuity of morphing operations. Morphing continuity differs from geometric continuity. Geometry often has discontinuities (for example, tangents) that must be preserved during morphing. Morphing continuity means that the geometry doesn't change suddenly as parameters change.

Parameter values must be simultaneously set to reasonable values to ensure valid geometry for analysis and optimization. Automating morphing is a challenge because CAD systems have evolved as interactive systems that let users fix poor results. Design optimization needs a geometry system that automatically varies parameters without user guidance and yet maintains design integrity and intent.



4 Blue curve morphs discontinuously into red curve. (Courtesy Boeing)

Multidisciplinary design causes us to rethink the geometric design process as well as the algorithms. For example, many CAD systems use a piecewise quadratic or cubic algorithm for defining a curve through a sequence of points. These algorithms will not reproduce an embedded straight line exactly. Preserving embedded line segments forces the curve fit algorithm to be modified whenever three successive points lie on or are near (determined by some system tolerance) a straight line.

Figure 4 contains a second example of geometry that seems reasonable but causes problems during morphing. Consider the surface $H(x, y, z) = y - ax^2 = 0$ so its intersection with the $z = 0$ plane is described by the function $f(x) = ax^2$. Parameter a is a shape parameter that varies according to an optimization process. Suppose the algorithm for approximating intersection curves uses piecewise straight lines and the fewest join points possible to achieve a certain tolerance. Now suppose the tolerance is 0.5. For values of a approximately equal to 1, the approximation will alternate between a function with constant value 0.5—that is, a spline with no knots—and a piecewise linear function—that is, a spline with one interior knot. In other words, varying the parameter a from slightly less than 1 to slightly greater than 1 causes the model to change discontinuously. As a passes through the value 1, the intersection curve not only changes shape (the blue line morphs to the red line in Figure 4), but its properties (for example, arc length) change discontinuously. This algorithm design produces good static approximations but fails during morphing.

Summary

Addressing the geometry challenges outlined here will not be just a simple task of going through and improving or deleting offending algorithms. It also requires a fundamental rethinking of how geometry design systems should work.

Interactive techniques

Significant improvements in interaction are not going to be achieved by making more efficient menus or a better mouse. Rather, they are going to depend on rethinking the nature of the process. Three specific challenges result:

- changing the order of workflow,
- understanding the concept of *place* in the workplace, and
- intentional design.

History

Interaction technology has evolved slowly over the past 40 years. Input devices have not changed significantly. Physical buttons, such as keyboards, functions keys, voice, fingers, and so on, let people talk to the machine. Graphical pointers come in numerous shapes and sizes and let us move in two or three dimensions, with similar resolution to early devices. The graphical user interface has remained essentially unchanged since 1984.

Displays have gotten a lot smaller and a lot larger. Smaller displays are ubiquitous because of the phenomenal growth in the cell phone industry; larger display penetration is steadily growing. The basic resolution of display devices (number of units per square inch) is about the same as Sutherland's Sketchpad was in the early 1960s.

Improvements have been achieved largely by adding new functionality, enhanced graphic design, and better flow of control. However, systems are not easier to use, and the demands on the user today might be even higher than 20 years ago. The complexity of designs, data, and geometry are growing as fast, or faster, than the power of the tools to handle it.

Nevertheless, some things are changing, and these changes will afford the potential to break out of this situation. For example, more than 10 years ago, Nicholas Negroponte challenged people to imagine what would be possible if bandwidth was essentially free. The only thing that matched how amazing and unlikely that concept was at the time was its prescience.

The parallel challenge today would be, "Imagine what would be possible if screen real-estate were essentially free." Already, paper movie posters are being replaced by $10,000 plasma panels. Imagine the potential impact when the cost of a comparable display drops two orders of magnitude and it is cheaper to mount a 100 dpi display on your wall than it is to mount a conventional whiteboard today.

Large displays will be embedded in the architecture of our workspace. We will stroll through and interact with such spaces with agile small, portable, wireless devices. And many of the changes that are going to affect the future of CAD will emerge from the evolving behavior of both people and devices as they function within them.

Within this context, a divide-and-conquer approach will be used to address the complexity posed by today's CAD systems. While power will come in numbers, most will be relatively inexpensive and target a particular function. The isolated gadgets that first emerge as add-ons to existing systems will morph into the keystones of a new mosaic of integrated technologies that will transform the process.

To realize the potential of this, the following three challenges involve thinking about evolution in a human, not technology, centric way.

Challenge 4: Reversing engineering

Important changes do not result from doing the same things faster or for less money. They come when you flip approaches and methods on their head, that is, we must constantly ask ourselves, Do we do things the way that we do today because it is the right way, or because it's

the only way that we knew how when we started?

The inertia of the status quo blinds us to the recognition that major changes, such as those due to Moore's law, open up different and desirable approaches. If we overcome that inertia, we can explore other approaches.

In the field of aerodynamics, for example, it's truly important to get designs that are efficient and safe. Government agencies such as NASA and companies such as Boeing spend millions of dollars on tools, such as wind tunnels and CFD analysis, that help them test such systems. However, these tests typically come late in the design process. Now combine this fact with one of the most basic rules of design: The later in the process that a mistake is detected, the more expensive it is to fix.[10]

Replacing this rigorous testing at the back end is not the answer, but it would be more efficient if technology existed that allowed preliminary tests on their desktop or even on their PDA. Designers could then catch bad designs much earlier in the process. This would also help them discover, explore, refine, and understand the most promising designs as early in the process as possible.

This approach to CFD could be applied to a number of other parts of the CAD workflow. Consider the ability to introduce elements such as stress testing, volume or strength calculations, and so on, earlier in the workflow. This would allow designers to consider the interconnection and interoperability of parts much earlier in the design cycle than is currently the case.

Adding simulation earlier into the design process will enable consideration of behavior, rather than just form, to play into the process. The fundamental change in workflow is the essential interactive technique that will make the geometry design exploration challenge of the previous section a useful and usable capability.

Challenge 5: Everything in its place

There was a time when someone's location in a CAD environment indicated their current job. For example, in an automotive design studio, there is often one location where people deal with interiors, another contains the full-sized clay models, and some other location is set up for exploring colors.

However, the typical modern CAD environment contains a uniform sea of anonymous cubicles or desks. In general, anyone walking through such a space will not likely be able to tell if they are in the accounting or the engineering department. Because people don't move around anyhow and are essentially anchored to their desk, this organization isn't important in some ways. Yet, in the midst of all of this, we hear an ever-louder call for collaboration. We also hear the emergence of ubiquitous computing. How do these two points relate, if at all, in terms of transforming the CAD workplace?

To begin with, ubiquitous computing will break the chain that anchors the engineer to a general-purpose workstation. This change will not just enable but necessitate the designer to move from one specialized area to another in a manner harkening back to the best of past practice.

Not only will the workspace be broken up into specialized areas with specialized tools, but these tools will also consist of a combination of private and public displays and technologies. Some will be mobile and others embedded in the environment. Examples of the latter would include large format displays that function as digital corkboards, surfaces where you can view large parts on a 1:1 scale, and areas where you can generate physical parts using a 3D printer.

The physical mobility of a person and data can greatly impact agility of thought. Mobility brings increased opportunity for collaboration and increased visibility of a particular activity. Rather than design a system that lets us send more email or documentation to the person at the other side of the studio, the studio should be designed so that we have a greater probability and opportunity to bump into and work with that person face to face. If we are going to have work-across sites, then linking designers' efforts must be linked automatically as a consequence of undertaking a particular activity.

Our notion of space is not about making things more abstract or virtual. Rather, it concerns the recognition and exploitation of the attributes and affordances of movement and location in a technology-augmented conventional architectural space. We discuss the impact of collaboration among the geographically distributed in challenge 10.

The challenge to future systems is as much about human–human interaction as it is about new forms of interaction between human and machine.

Challenge 6: What we do

CAD companies might view themselves as primarily purveyors of tools for the creation of high-quality 3D models. Consequently, if they want to grow their business, they might conclude that they should make a better, more usable modeler.

However, this stream of thought might be as wrong as it is reasonable. Consider the following questions:

- Is there a shortage of trained people to fill the existing demand for creating 3D models?
- Are there things that need to be modeled that cannot be built by existing users with their current skills and tools?
- Is there a huge untapped market for 3D models waiting for an easy-to-use modeling package that takes little training to use?

In general, the basic answer to all of these questions is "No." It's not modeling—at least in the sense that it exists in today's CAD packages—that lies behind any of the fundamental challenges outlined in this article.

In fact, it might well be that all of these questions are poorly posed, since they all assume that the intent of the user is modeling. It is not. Rather, it's getting a product or a part made. Modeling is just one way of doing so, and in many cases, not always the best way.

We are then challenged to answer this better question: What besides modeling from scratch might enable us to achieve our product design? A good answer to that question should lead to a more innovative and effective solution than today's design-from-scratch modeling approach.

Our favorite recourse in such cases is to look to the

past. In this case, the clue lies in Figure 2, the NACA catalog of airfoil sections. As we discussed previously, such catalogs let designers build an aircraft by selecting components with known properties rather than working from scratch.

We suggest moving back to this approach—but with appropriate modifications. As product complexity increases, the CAD process will subdivide into those specialists who design and build components (often out of subcomponents) and those who make larger assemblies out of them. However, the components in future CAD systems will not be fixed objects made up of what computer scientists might call declarative data. Rather, they will include a strong procedural component.

The basic problem is that objects, even those as simple as an airfoil, don't easily scale. If you make exactly the same form, but larger, the plane might not fly. A simple example is the hummingbird. It can definitely fly. However, if you scaled it up in size by a factor of 10, it could not.

In this object-oriented process, designers generally do more than just select components and plug them into a larger assembly. Rather, they take components and transform them into what is needed. The components act as a starting point, something designers work from, rather than start from scratch.

Given that physical products don't scale, one key aspect of component design is the component's embedded capacity to be transformed along meaningful and desired dimensions, while maintaining specific tolerance, performance characteristics, manufacturability, maintainability, and so on.

Embedded in the component catalog, therefore, are just not the parts, but the knowledge of how parts can be transformed while maintaining essential properties. Consequently, the tools of the CAD engineer not only help the designer find the appropriate components but also support transforming them in such a way as to maintain the desired properties.

Finding, cloning, modifying, morphing, and adapting will largely replace constructing from scratch. In so doing, the assumption is that methods such as aerodynamic testing, stress analysis, and others, can occur in advance for each component and carry over into an assembly. This approach transforms the current practices of who does what, where, when, why, and how.

Summary

Emerging user-interface technology will allow us to transform how we work. Furthermore, the new technology can help us implement the transformation without any major discontinuity with respect to current practice.

Our sense and analysis of interaction needs to switch from how we interact with a specific computer or package to how we interact as people, how we interact with change, and how we interact with our materials—at what level, in what way, and to what objective.

Scale

The challenge of scale is one that scientists and engineers continue to encounter as they push the limits of macro- and nanotechnology. Moore's law governs the expansion of computing hardware's limits. Advanced hardware lets us produce larger quantities of data. Software advances tend to lag behind more powerful hardware, yet we seem to be able to consume computing hardware resources at a rate that always leaves CAD/CAM users begging for larger storage capacity, greater network bandwidth, and better performance.

Managing scale results in a delicate balancing act of knowing when "good enough" occurs. While there are numerous dimensions of scale in CAD/CAM/CAE, the primary areas of challenge today include

- Sheer data quantity that can exceed a project team's software, hardware, and cognitive capacity.
- Making critical data comprehensible to other users.
- Keeping data meaningful as product life spans become longer.
- Collaboration with a geographically distributed workforce.

The challenges of scale in this section derive from experience at Boeing, a large and varied aerospace manufacturer. While Boeing does not represent the norm, similar problems exist in automobile production, shipbuilding, and other manufacturing companies. More importantly, solving a Boeing-sized problem has often resulted in breakthroughs that have profoundly affected smaller efforts.

History

Scale has grown as the overall process of designing, building, and maintaining complex products has changed.

Humans have designed and built extraordinarily complex artifacts throughout history. The seven wonders of the ancient world are clear examples: Roman aqueducts still deliver water service to Italian cities, and the Great Wall of China still stands, acting as a tourist destination and is clearly visible from the ground and Earth orbit.

Documentation on the ancient design process is sketchy, but became more formal during the Renaissance. Leonardo da Vinci's sketches of various mechanical possibilities captured a more formal depiction and allowed designers to analyze possible configuration strengths and weaknesses before construction started. These practices continued to evolve into highly accurate renderings documented as engineering drawings in the first half of the 20th century. The evolution of computational geometry described earlier has taken us to the point we are now in the 21st century.

The challenges we discuss in the rest of this section are the direct result of change in the fundamental design-analyze-build-maintain process. Until the latter part of the 20th century, highly complex projects featured integrated design-build teams. Slave labor was the dominate force in the earliest build teams, and on-site designers oversaw every detail of construction. Large efforts took a long time and involved thousands of workers. There has been a desire to decrease construction time and the number of people ever since.

As increased specialization in individual aspects

became more prevalent, principal design and assembly continued to exist within close geographic proximity. Consider the evolution of design-build at Boeing. Boeing's first airplanes were designed and built in the Red Barn (see Figure 5). As build problems occurred, engineers could walk to the factory to make on-site corrections.

As automation and the complexity of the product increased, later airplane generations continued to be designed and built in the relatively close proximity of the Puget Sound region in Washington. Boeing located the engineers designing the 737 and 757 within minutes of the Renton assembly plant; Everett housed the 747, 767, and 777. These engineers were responsible for the design of all of the major sections of all airplanes.

With the 787, its next-generation airplane, Boeing is distributing the design, manufacture, and assembly of major subsections across the world. Each major supplier will deliver preassembled sections of the airplane to final assembly, rather than delivering smaller components as occurred with previous models. This approach will dramatically reduce final assembly times. Computing and communications systems are mandatory for this to occur, and huge amounts of data must be integrated and managed. Because the design is completely digital and 3D, a larger variety of people will look at images of the data during the 787's life cycle. Time and distance complete the notion of scale: mechanical products built today have longer and longer life times, and geographic distance separates people.

5 **Boeing Red Barn. (Courtesy Boeing)**

Challenge 7: Understanding vast quantities of data

Design, engineering, manufacturing, and maintenance processes have routinely existed in virtual isolation from one another. Part of the scope challenge relates to data, especially as products become complex. Another part relates to people because the graphics vocabulary used in the design process is dramatically different from the one used when the product is being manufactured and assembled.

Early efforts in product design used the waterfall model. In this apprach, designers handed drawings to engineers for analysis leading to design improvements. The drawings then made their way to manufacturing planners, people making and assembling parts, and others responsible for production maintenance. The late 1980s saw a change in the designer-engineering analysis interface because of the advent of 3D modeling systems with enough capacity to generate and manage an entire commercial airplane. Downstream users continued with master definitions represented as 2D drawings.

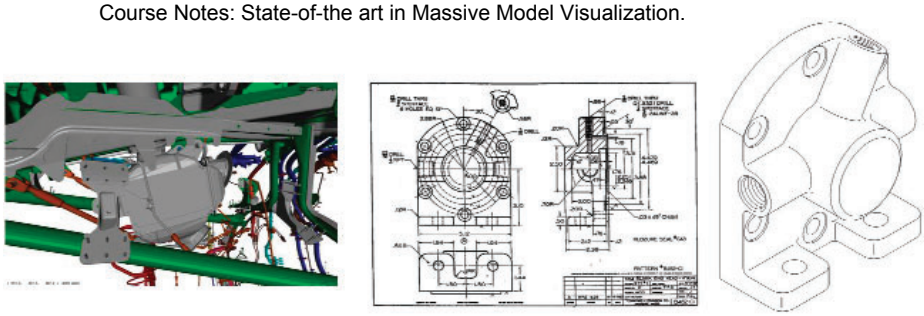
Initial efforts to deal with issues like manufacturability and maintainability focused on integrated design-build teams. The team members used different applications and kept data in segregated areas with different configuration management schemes.

CAD/CAM software vendors are starting to address this problem with product lifecycle management (PLM) systems that extend previous product data management (PDM) systems. PLM systems let companies store customer requirements, design geometry, engineering analysis results, manufacturing plans, factory process designs, maintenance designs, and so on, in a single repository that infuses configuration management throughout.

PLM systems are difficult to sell. The essential benefit (managing data across key aspects of a manufacturing company) generates more cost in each individual area but decreases overall product costs through forced pre-facto integration and better configuration management and visibility. PLM systems compete with other large corporate systems for managing personnel, enterprise resource planning (ERP), supply chain management (SCM), and so on. Personnel management, PLM, ERP, and SCM systems all replicate significant portions of the same data. Finally, getting new technology sold, especially when the technology works behind the scenes, is always a difficult problem.

The real problems occur after product integration becomes institutionalized. Integrated products must be examined from a number of different views. Each group has a different purpose and wants to believe that their view is the master. For example, a commercial airplane has the following views:

- *As-designed view* (engineering bill of materials). The relationships among individual components reflect a logical organization of the data (primarily geometry) as modified to satisfy engineering improvements for aerodynamics, structural strength, weight, and so on.
- *As-ordered view* (manufacturing bill-of-materials). The relationships among individual components reflect the parts that must be ordered and the specific attributes (including geometric definition) needed to successfully acquire the parts. Concepts like alternate suppliers are important in this view.
- *As-delivered view*. This view contains information about the physical configuration turned over to a customer after the assembly process is complete. Some components might have been replaced (for example, a supplier changed) or modified during assembly.



(a) (b) (c)

6 Communicating data: (a) solid shaded image, (b) engineering drawing, and (c) stylized maintenance image. (Courtesy Boeing)

- *As-owned view.* The actual components in a final product change over time. Components get routinely replaced or repaired as maintenance occurs.

Each of these views is essential during a product's lifecycle and each wants to be master. Providing each view without negatively impacting the integrity of the underlying data poses a difficult challenge.

In terms of sheer storage space, the amount of data needed to faithfully represent a product and to show how decisions were reached in its design is expanding by decimal orders of magnitude. The data ranges from marketing intelligence about customers to versions of complex geometry to multiple data sets from engineering analysis runs to manufacturing planning scripts to sensors monitoring manufacturing processes, physical testing, and real-time product health.

In addition to managing multiple configurations and views, people face a challenge when they have to interpret this vast quantity of information. For example, technology is on the horizon that will allow real-time display[11,12] of an entire commercial airplane model. Humans have a finite ability to comprehend complexity, and understanding how to best display the data in a meaningful way requires extensive work.

Challenge 8: Appropriate designs for other uses and users

This challenge focuses on the large number of people who are removed from the data creation and analysis process and rely on the digital product definition for their own job tasks. These people participate in sales, fabrication, assembly, certification, maintenance, and product operation. In the case of a commercial airplane, the user community extends far beyond company boundaries. For example, airline personnel buy Boeing products, perform maintenance, order spares, train pilots and attendants, and so on. Partners design and analyze significant airplane components. Suppliers formulate proposals to fabricate parts and assemble components. Government agencies certify that the final product performs according to published regulations and guidelines.

Many of these groups have their own graphic vocabulary because different information must be communicated to different audiences. Consider the following examples.

Engineers and designers commonly use images like Figure 6a. The colors are meaningful in terms of the engineering and design functions of individual components, but not for manufacturing or maintenance. This is a significant change from practice prior to 1990, where the master communication mechanism was the engineering drawing (see Figure 6b). The authority image for certification, manufacturing, and maintenance is still the engineering drawing. Communicating via the engineering drawing to a maintenance engineer causes a radical transformation, as Figure 6c shows.

Figures 6b and 6c are clear illustrations of this challenge. The two represent exactly the same 3D geometry. However, the graphic styles are dramatically different. Tools are commonly available to work with the base 3D model—for example, make everything in a scene translucent but objects of interest; show and hide; rotate, scale, and translate; and measure. We can generate the basic 3D hidden line image for Figure 6c, do some early guesses to explode parts, and so on.

These tools fundamentally change the basic style in which the graphic image is rendered, and some might even produce engineering drawings. None contains an ability to adapt an image for specific uses or users. Therefore, the challenge lies in knowing how to draw the image to communicate to a specific user community because the communication techniques and standards vary significantly from community to community and from company to company.

As a result, companies spend significant amounts of time and labor retouching or redrawing images. Automation also offers the possibility of providing new types of images that might be more task appropriate.

Generating the initial image itself presents a significant challenge. The taxonomy of effective graphic images (Figure 6 represents a small subset) is much larger. Massironi[13] has developed an excellent taxonomy that delineates the types of images and the fundamental graphic techniques people have used to communicate visually.

Challenge 9: Retrieving data years later

Today's product specifications for tolerance, fit, reliability, and so on, are greatly different than they were 40 years ago. For example, the Boeing 707 successfully introduced commercial aviation to the jet age. Yet the 707's part fit was loose enough that it received the nickname "the flying shim." On the other hand, the first Boeing 777 fit together so precisely (largely due to the use of CAD/CAM techniques from 10 years ago) that the number of discrepancies needing redesign was substantially less than what had appeared to be an extremely optimistic early prediction. Rather than the multiple mock-ups needed for previous models, the 777 manufacturing mock-up flew as part of the flight certification process. Similar stories exist in the automotive and other industries.

One constant remains: the engineering drawing serves as the design, manufacturing, and certification authority. While we realize that the engineering drawing has its limits, it has another important attribute: It can be archived for long periods of time and still be understood. The Mylar film that Boeing uses for its permanent records lasts longer than the 50-year life span of commercial airplanes.

In contrast, consider the challenge if the archives were stored digitally. The media itself is not the problem because it can be routinely copied to new media. The real challenge is changing software versions. Users routinely expect that some percentage of their data will not migrate successfully from one version to the next. Some algorithms will be tweaked, and what worked in the last version doesn't in the next (or vice versa). A substantial version change or choosing a different software supplier means massive amounts of data conversion and rework. The net result is that companies cannot afford to change to a new system version from the same vendor until years after initial release, let alone change to a new vendor entirely. As the amount of configuration-managed data increases, the data migration challenge becomes even more pronounced.

Challenge 10: Limited by the speed of light

Challenge 5 discussed the impact of users being released from the constraints of single user desktop and laptop computers into an environment where workspaces can be shared. This challenge addresses the users working in a geographically distributed environment, where walking across the hall becomes impossible.

A continued debate rages about the internationalization of large and small businesses throughout the world. In spite of specific national interests and boundaries, companies (many of which are small and mid-sized) are acquiring design, manufacturing, assembly, maintenance, and customer support service help throughout the world. The incentives range from specific technical skills to trade offsets to cheaper labor.

Doing effective distance collaboration for 3D design, which lacks the immediacy and extensive cues of a face-to-face session, is a long-term research and cultural challenge. The key productivity aspect for improved design cycle time is collaboration across a number of different stakeholders. There are a wide variety of collaboration models and a reasonable amount of research done in computer-supported collaborative work. Researchers are starting to pay attention to collaborative task analysis.[14] However, research on how to make such efforts more effective on a global scale is in its infancy, and the cognitive effects of distance on a collaborative 3D work environment have not been addressed.

Summary

This final set of challenges indicates that the way groups of people work on design problems and the longevity of the electronic versions of their products is undergoing a dramatic change. The magnitude of the challenges these changes are causing is at least as great as getting people to adopt CAD in the first place because the challenges impact the fundamental way people work.

How to proceed: Spurn the incremental

We conclude by suggesting an approach that can address our 10 challenges. The basic CAD business model is incremental refinement, and it can only take us so far.

Incremental refinement is based on bringing out the $n + 1$st version, where the new version release has improvements. For this model to work, the incremental improvement for release$_n$, ΔI_n, must be greater than some threshold value, VT, which represents the minimum improvement that will still motivate a company to purchase the new version.

The cost of achieving that degree of improvement increases with each release and can be approximated by

$$\Delta I_n \geq VT \Rightarrow \$\Delta I_n = \$\Delta I_2^{On}$$

In other words, to keep the incremental value of release$_n$ high enough to motivate purchases, the cost of those improvements is on the order of the cost of those of the second release (that is, the first upgrade), raised to the order's nth power.

Two factors contribute to this cost picture. First, as systems go through successive releases, they grow in complexity. This negatively affects their malleability and increases the cost of adding value or refactoring the software or basic functionality. It's not just the number of lines of code or additional features that cause this. As a system approaches maturity, the legacy of the initial underlying architecture, technologies, and paradigms creates a straightjacket that severely affects the cost of change.

Next, as products mature, the software more or less works as intended and markets approach saturation. For most users, the current version of the software is good enough. Changes increasingly tend toward tweaks and tuning rather than major improvements. Or they are directed at more specialized functions needed by a smaller segment of the market. Consequently, there is less to motivate most customers to upgrade.

As the product reaches late maturity, the accumulated impact is that development costs increase while the size of the addressable market decreases. Software sales are no longer sufficient to cover the growing development costs. At this point, companies increasingly rely on annual support contracts, a model difficult to sustain. At best, the switch to support revenue delays the collision of technology costs and economic viability. Even when a company totally reimplements its product for a new version, the tasks the user performs are generally the same as in the previous version and often give less functionality than the last version of the old system.

The CAD industry is reaching this state. How do we get to the next level, the one that should address these challenges?

Order-of-magnitude approach

If incremental refinement won't work, perhaps it will come from some new breakthrough. Perhaps some new invention will magically appear and save the day. The National Academy of Science recently released a report that studied the genesis of a large number of technologies, including graphical user interfaces, portable com-

munication, relational databases, and so on. For each, it looked at how long that technology took to get from discovery to a $1 billion industry. The average was about 20 years.[15] This suggests that any technologies that are going to have impact over the next 10 years are already known and have probably been known for at least 10 years.

We believe that the answer is not new technologies but new insights into technologies that are already known, whose potential is perhaps not appreciated, or which have so far not been technically or economically feasible. The engineering of significantly better products should come from fresh insights on what is already known or knowable.

By analogy, think about the marks on the door frame of the kitchen closet where parents record the growth of their children. Without that documentation, especially because you live with them day-to-day, you generally don't notice the changes. Long after it has actually happened, by measuring them again, you notice that they have passed on to the next stage of growth. This also happens in living with technology.

Our observation leads to the order-of-magnitude (OOM) rule, which says: If anything changes by an order of magnitude along any dimension, it is no longer the same thing.

OOM is a way to measure significance of the types of changes needed to meet the technical challenges we've described. Exploiting this rule forces us to notice OOM changes and understand their implications. However, it also lies in teasing out less obvious, but meaningful dimensions, along which to test for such OOM changes.

Because we have all been so intimately involved with these challenges and CAD technology, we have not taken the time to create the measures that allow us to see where profound changes have already occurred and where they need to occur. We believe that the approaches needed to address the 10 challenges are most likely to result from rethinking things from a human-centric rather than a technology-centric perspective.

Conclusion

The next wave in CAD will come about largely through the cumulative effect of the introduction of a number of small, lightweight technologies that collectively form a synergistic mosaic, rather than due to the introduction of some monolithic new technology. The value of this approach is that it can be introduced incrementally, without requiring some disruptive discontinuity in skills and production. Thus, change will occur through radical evolution. That is, incremental evolutionary change will happen in a way that leads us to a radically new approach to working.

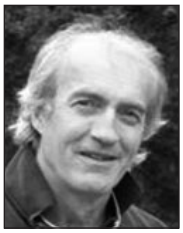
There will be changes in how we do things, and these will significantly affect the nature of CAD systems and how they function. We suggest that using OOM offers a strategy to cause revolution in an evolutionary manner. Ultimate success will happen if and only if all of the technical areas in which OOM changes occur are kept in balance. We conclude that institutionalizing radical evolution is the top CAD challenge we face in 2005, 2015, and beyond. ■

References

1. I. Sutherland, R. Sproull, and R. Schumacker, "A Characterization of Ten Hidden-surface Algorithms," *ACM Computing Surveys*, vol. 6, no. 1, Mar. 1974, pp. 1-55.
2. R.A. Liming, *Practical Analytic Geometry with Applications to Aircraft*, Macmillan Company, 1944.
3. P. Garrison, "How the P-51 Mustang Was Born," *Air & Space Magazine*, Aug./Sept. 1996.
4. D.R. Ferguson, P.D. Frank, and A.K. Jones, "Surface Shape Control Using Constrained Optimization on the B-spline Representation," *Computer Aided Geometric Design*, vol. 5, no. 2, 1988, pp. 87-103.
5. R.T. Farouki,"Closing the Gap between CAD Model and Downstream Application," *SIAM News*, vol. 32, no. 5, 1999, pp. 303-319.
6. Research Trianagle Inst., *Interoperability Cost Analysis of the US Automotive Supply Chain*, Nat'l Inst. of Science & Technology, March 1999; http://www.mel.nist.gov/msid/sima/interop_costs.pdf.
7. C. M. Hoffmann, *Geometric and Solid Modeling: An Introduction*, Morgan Kaufmann, 1989.
8. D.R. Ferguson et al., "Algorithmic Tolerances and Semantics in Data Exchange," *Proc. 13th ACM Symp. Computational Geometry*, ACM Press, 1997, pp. 403-405.
9. V. Shapiro, *Current Limitations of CAD Systems*; http://sal-cnc.me.wisc.edu/Research/parametric/limits.html.
10. F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1975.
11. W. Baxter et al., "GigaWalk: Interactive Walkthrough of Complex Environments," *Proc. Eurographics Workshop on Rendering*, ACM Int'l Conf. Proceeding Series, 2002, pp. 203-214.
12. I. Wald, C. Benthin, and P. Slusallek, "Distributed Interactive Ray Tracing of Dynamic Scenes," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics* (PVG), IEEE Press, 2003.
13. M. Massironi, *The Psychology of Graphic Images: Seeing, Drawing, Communicating*, Lawrence Erlbaum Assoc., 2002.
14. D. Pinelle, C. Gutwin, and S. Greenberg, "Task Analysis for Groupware Usability Evaluation: Modeling Shared-Workspace Tasks with Mechanics of Collaboration," *ACM Trans, Computer–Human Interaction*, vol. 10, no. 4, 2003, pp. 281-311.
15. Computer Science and Telecommunications Board, "Computer Science and Telecommunications Board of the National Research Council," *Innovation in Information Technology*, National Academies Press, 2003.

David J. Kasik *is an architect for software engineering, geometry and visualization, and user-interface technology at the Boeing Commercial Airplanes Group Information Systems. His research interests include an innovative combination of basic technologies and increasing awareness of the impact of computing innovation outside the computing community. Kasik has a BA in quantitative studies from the Johns Hopkins University and an MS in computer science from*

the University of Colorado. He is a member of ACM, ACM Siggraph, and ACM SIGCHI and received a Siggraph Special Recognition Award in 1992 for acting as Liaison to the Exhibitor Advisory Committee. Contact him at David.j.kasik@boeing.com.



William Buxton *is principal of his own boutique design and consulting firm, Buxton Design, where his time is split between working for clients, writing, and lecturing. He is also a chief scientist for Bruce Mau Design of Toronto, and is an associate professor in the Department of Computer Science at the University of Toronto. Buxton has a B.Mus. from Queens University and an MSc in computer science from the University of Toronto. He received the 1995 Canadian Human–Computer Communications Society Award, the 2000 New Media Visionary of the Year Award, and is a member of the CHI Academy. Contact him at bill@billbuxton.com.*

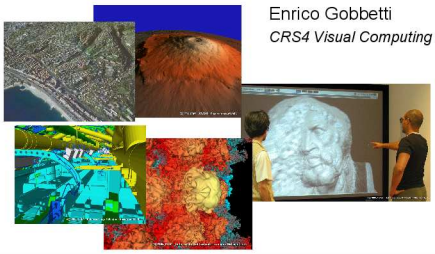
David R. Ferguson *was Boeing's leader in geometry research and development until his retirement. His research interests include the mathematics of curves and surfaces with special emphasis on nonlinear methods for geometric construction. Ferguson has a BS in mathematics from Seattle University, and an MS and PhD in mathematics from the University of Wisconsin. He is an editor of* Computer Aided Geometric Design. *Contact him at drf.assoc@comcast.net.*

For further information on this or any other computing topic, please visit our Digital Library at http://www.computer.org/publications/dlib.

4 Overview Technical Context

Massive Models Visualization: Technical context

Enrico Gobbetti
CRS4 Visual Computing



Goal: interactive inspection of massive models on PC platforms...

Xeon 2.4GHz / 1GB RAM / 70GB SCSI 320 Disk / NVIDIA 6800GTS

Application domains / data sources

Local Terrain Models
2.5D – Flat – Dense regular sampling

Planetary terrain models
2.5D – Spherical – Dense regular sampling

Laser scanned models
3D – Moderately simple topology – low depth complexity - dense

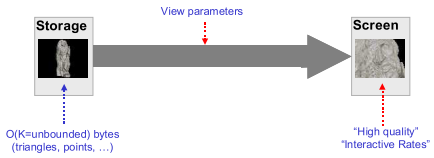
CAD models
3D – complex topology – high depth complexity – structured - 'ugly' mesh

Natural objects / Simulation results
3D – complex topology + high depth complexity + unstructured/high frequency details

- Many important application domains
- Models exceed
  - $O(10^8\text{-}10^9)$ samples
  - $O(10^9)$ bytes
- Varying
  - Dimensionality
  - Topology
  - Sampling distribution

The (minimal) challenge

- Explore very large models at interactive rates
  - Update screen at "interactive rates" as viewpoint changes



Interactive rendering constraints

Regular desktop displays
~1M pixels

Geowall-type displays
~1-10M pixels, stereo

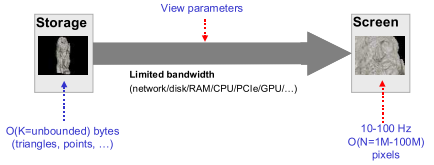
Tiled high resolution displays
~10-100M pixels

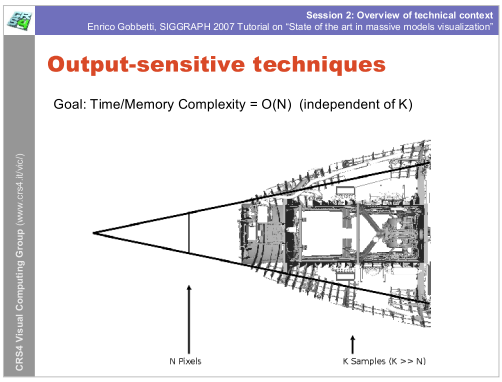
3D displays
~10-100M pixels, holo

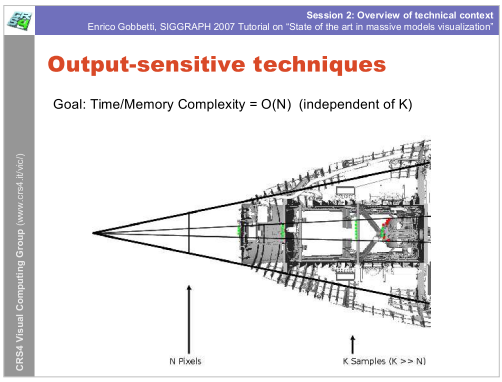
- Frequency, latency, resolution, should match human capabilities…
  - … or at least output device's ones!
- On today's displays
  - Frequency: 10-100Hz
  - Latency: ~0.1s
  - Resolution: $O(10^6\text{-}10^7)$ px

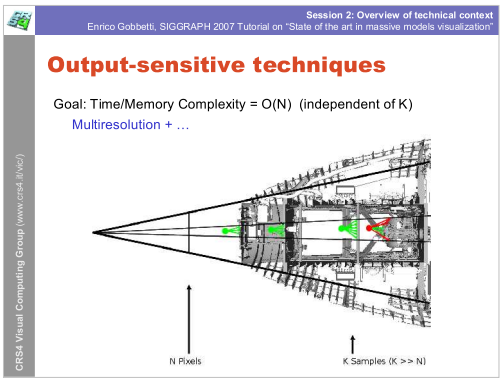
A real-time data filtering problem!

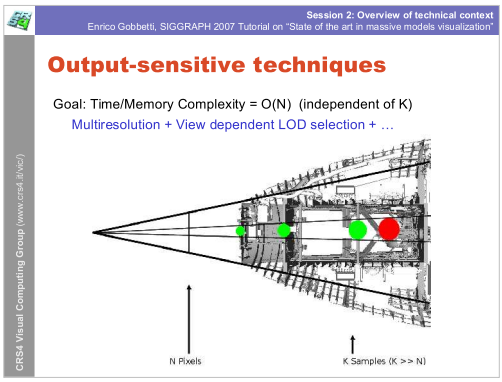
- Models of unbounded complexity on limited computers
  - We assume **less data on screen (N) than in model ($K \rightarrow \infty$)**
  - Need for **output-sensitive** techniques (O(N), not O(K))

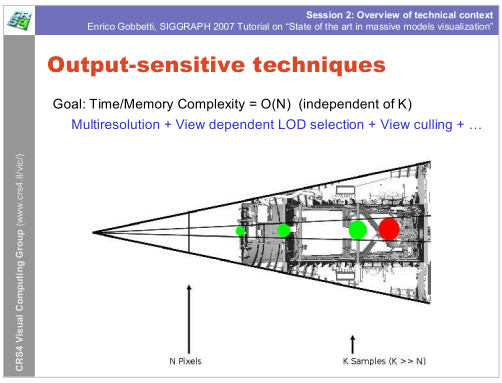


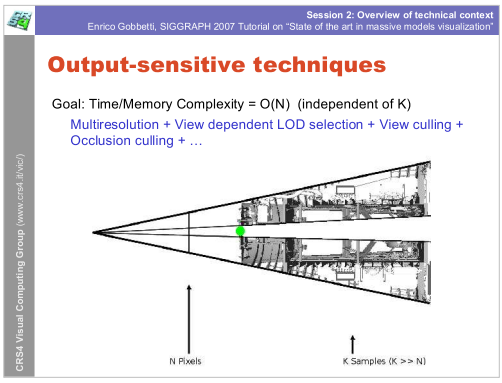

Session 2: Overview of technical context
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Output-sensitive techniques
Goal: Time/Memory Complexity = O(N) (independent of K)
N Pixels
K Samples (K >> N)


Session 2: Overview of technical context
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Output-sensitive techniques
Goal: Time/Memory Complexity = O(N) (independent of K)
N Pixels
K Samples (K >> N)


Session 2: Overview of technical context
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Output-sensitive techniques
Goal: Time/Memory Complexity = O(N) (independent of K)
Multiresolution + …
N Pixels
K Samples (K >> N)


Session 2: Overview of technical context
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Output-sensitive techniques
Goal: Time/Memory Complexity = O(N) (independent of K)
Multiresolution + View dependent LOD selection + …
N Pixels
K Samples (K >> N)

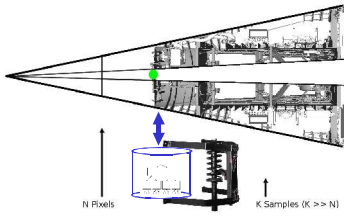

Session 2: Overview of technical context
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Output-sensitive techniques
Goal: Time/Memory Complexity = O(N) (independent of K)
Multiresolution + View dependent LOD selection + View culling + …
N Pixels
K Samples (K >> N)


Session 2: Overview of technical context
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Output-sensitive techniques
Goal: Time/Memory Complexity = O(N) (independent of K)
Multiresolution + View dependent LOD selection + View culling +
Occlusion culling + …
N Pixels
K Samples (K >> N)

Output-sensitive techniques

Goal: Time/Memory Complexity = O(N) (independent of K)

Multiresolution + View dependent LOD selection + View culling + Occlusion culling + External memory management/Compression



CRS4 Visual Computing Group (www.crs4.it/vic/)

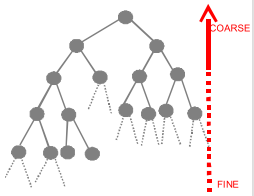
Ouput sensitive technique

- All techniques must combine
  - A multiresolution and spatial subdivision structure
    - Visual/geometric approximations
    - Spatial indexing
  - A view-dependent renderer
    - LOD culling
    - Visibility culling
  - An efficient memory management subsystem
- Relative weight of components varies depending on model kind
  - E.g., LODs more important than occlusion for flight simulators

CRS4 Visual Computing Group (www.crs4.it/vic/)

Output-sensitive techniques

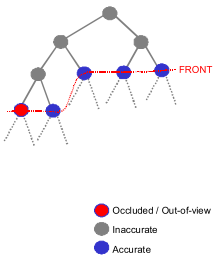
- At **preprocessing** time: build MR hierarchy
  - Data prefiltering!
  - Visibility + simplification
  - Not output sensitive



CRS4 Visual Computing Group (www.crs4.it/vic/)

Output-sensitive techniques

- At **preprocessing** time: build MR hierarchy
  - Data prefiltering!
  - Visibility + simplification
  - Not output sensitive
- At **run-time**: selective view-dependent refinement from out-of-core data
  - Must be output sensitive
  - Access to prefiltered data under real-time constraints
  - Visibility + LOD



CRS4 Visual Computing Group (www.crs4.it/vic/)

Output sensitive techniques

- Two main rendering approaches
  - Rasterization + Z-buffering (GPU)
    - Start from model
  - Ray-tracing (CPU/RPU)
    - Start from screen
- For large models, methods share many common points
  - Similar hierarchical structures
  - Need for approximate representations to build multiresolution hierarchies
  - Similar memory management subsystem, typically exploiting spatial/temporal coherence

CRS4 Visual Computing Group (www.crs4.it/vic/)

The remainder of the tutorial: a state-of-the art review

- Efficient data reduction/cache coherence techniques
- GPU based techniques
  - Scalable parallel visual networking
  - Visibility guided rendering
  - GPU accelerated coarse grained MR models
- Ray-tracing techniques
  - Multi-core/Shared memory systems
  - HW/SW approaches for real-time raytracing
- Putting theory into practice
- Discussion

CRS4 Visual Computing Group (www.crs4.it/vic/)

5 Efficient Data Reduction and Cache-Coherent Techniques toward Real-time Performance

Efficient Data Reduction and Layout Techniques for Interactive Rasterization and Ray Tracing of Massive Models

Sung-Eui Yoon[1] Dinesh Manocha[2]
[1] Lawrence Livermore National Laboratory
[2] Univ. of North Carolina at Chapel Hill

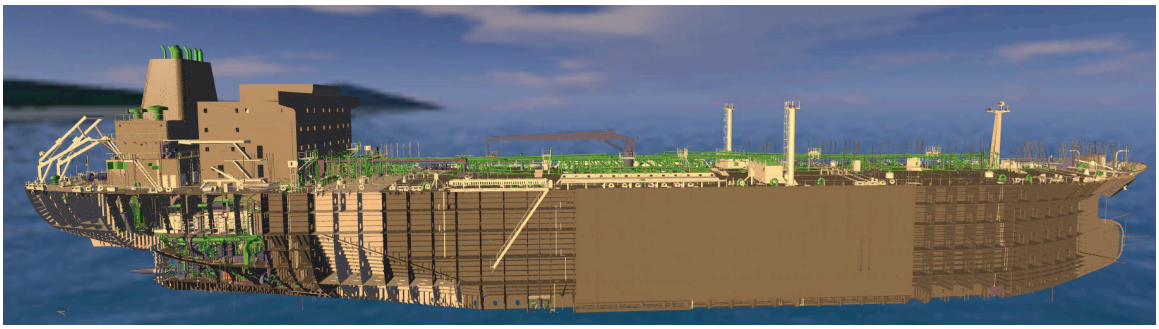


Figure 1: Application of our multi-resolution techniques and cache-efficient layouts to the rendering of the double eagle tanker (82 million triangles). By using these techniques, we are able to significantly improve the performance over the previous rasterization techniques and achieve an interactive performace on a commodity hardware. The performance improvement is mainly achieved by the significant reduction on the amount of necessary data and cache-coherent access patterns on the data.

1 Introduction

Over the last decade, advances in model acquisition, computer-aided design (CAD), and simulation technologies have resulted in massive data sets of complex geometric models. These data include complex CAD environments, scanned urban data, and various scientific simulation data. These massive data typically require gigabyte size and even terabyte size. Moreover, as we have more computation power, more complex models are used to represent more detail of various natural environment and scientific phenomena. Therefore, it is expected that the model complexity will increase significantly in a near future.

There are, however, increasing demands to interactively visualize and analyze these complex and massive geometric data sets to extract meaningful observations, intuitions, and scientific discoveries. Given these high model complexities, many traditional geometric algorithms do not provide an interactive performance. Moreover, a major trend over the last few decades has been the widening gap between processor speed and data access speed [Yoon et al. 2005a]. This makes challenging problems to process these data sets and visualize them in real-time for interactive applications.

At a broad level, algorithms for rendering geometric models can be classified into rasterization and ray tracing methods. Both these methods have relative advantages. The rasterization techniques can result in higher frame rates as they can exploit the capabilities of current GPUs. In the last few years, there has been renewed interest in interactive ray tracing and these algorithms easily map to current multi-core architectures. Moreover, ray tracing algorithms can easily generate effects such as shadows, reflections and refractions. In terms of application to large models, the performance of rasterization as well as ray tracing algorithms is mainly limited by the large data size and memory inefficiencies of the underlying algorithms. In this course note, we would like to discuss the following three major orthogonal approaches to improve the performance of rendering methods and other geometric applications:

- **Multi-resolution techniques:** Multi-resolution techniques target for reducing amount of data necessary for performing specific geometric applications within an error bound. An example includes a view-dependent

rendering [Clark 1976; Funkhouser and Squin 1993; Hoppe 1996; Lindstrom et al. 1996]. The main idea of view-dependent rendering is to use lower resolution on the portion of the mesh that is far away from the viewer.

- **Cache-coherent layouts:** These algorithms reduce the number of cache misses in various levels of memory hierarchies in order to improve the performance of applications. Cache-coherent layouts are constructed in a way that it achieves this goal by strong data that are likely to be accessed together closely in the layout.
- **Memory-efficient representations and compressions:** Many efficient representations have been proposed to lower the storage overhead for rasterization and ray tracing. Moreover, compression techniques reduce the redundancy in the data and compactly represent the data in disk or memory and they can also improve the runtime application.

We will discuss these three orthogonal techniques in detail with detail examples in various geometric applications including, view-dependent rendering, ray tracing, collision detection, and iso-surface extractions.

2 Multi-Resolution Techniques

There are many multi-resolution algorithms or levels-of-detail (LODs) methods for different geometric applications including interactive rendering and collision detection. These rendering and collision detection are frequently required to visualize various models and interact with geometric models under investigation. In this section we will discuss LOD techniques in two major rendering methods: rasterization and ray tracing.

2.1 Rasterization

LODs have been widely used to accelerate rasterization of large polygonal datasets [Luebke et al. 2002]. At a broad level, prior algorithms can be classified into static LODs, view-dependent simplification [Xia et al. 1997; Hoppe 1997; El-Sana and Varshney 1999; Yoon et al. 2004b; Cignoni et al. 2004], and hybrid combinations of geometric and image-based representations [Gobbetti and Marton 2005].

View-dependent rendering using dynamic simplification (or view-dependent simplification) originated as an extension of both the progressive mesh (PM) [Hoppe 1996] and view-dependent metrics measuring projected geometric error in screen space [Lindstrom et al. 1996]. A PM is a linear sequence of increasingly coarse meshes built from an input mesh by repeatedly applying edge collapse operations. It provides a continuous resolution representation of an input mesh and is useful for efficient storage, rendering and transmission. However, PMs are not well suited for view-dependent rendering due to the nature of its linear sequences of LOD meshes stored in the PMs.

To address this issue, many researchers [Xia et al. 1997; Hoppe 1997; Luebke and Erikson 1997; Floriani et al. 1998] organized the linear sequences of LOD meshes as a vertex hierarchy in order to address the shortcoming of PMs in view-dependent rendering. This representation allows a runtime application to take into account view-dependent effects such as silhouette preservation and lighting.

2.1.1 Toward-Scalable View-Dependent Rendering

Some of earlier representations and refinement algorithms for view-dependent rendering may not scale well to large models composed of tens or hundreds of millions of triangles. The refinement cost is a function of the front size and may be prohibitively expensive for massive models. Furthermore, resolving dependencies in the vertex hierarchy can be expensive (e.g. hundreds of milliseconds or more per frame).

In addition to reducing the refinement cost, it is necessary to integrate view-dependent simplification algorithms with occlusion culling and out-of-core rendering. Occlusion culling computes a set of potentially visible primitives during each frame and is needed to handle high depth complexity models. Out-of-core rendering techniques operate with a bounded memory footprint and are required to render massive models on commodity graphics systems with

limited memory [Silva et al. 2002]. Algorithms for occlusion culling and out-of-core techniques also perform computations based on the view parameters. However, no known algorithms integrate conservative occlusion culling and out-of-core rendering with vertex hierarchies.

Particularly, Yoon et al. [2004b; 2005b] proposed a *clustered hierarchy of progressive meshes*(CHPM) for interactive view-dependent rendering of massive models. The CHPM consists of two parts:

- **Cluster Hierarchy:** The entire dataset is represented as a hierarchy of clusters, which are spatially localized mesh regions. Each cluster consists of a few thousand triangles. Conceptually, a cluster hierarchy is similar to a vertex hierarchy. However, every node of a cluster hierarchy represents a set of vertices and faces rather than a single vertex. The clusters provide the capability to perform coarse-grained view-dependent (or selective) refinement of the model. They are also used for visibility computations and out-of-core rendering.

- **Progressive Mesh:** Then, each cluster is also represented as a linear sequence of edge collapses as a progressive mesh (PM). A PM is a mesh sequence built from an input mesh by a sequence of edge collapse operations. The inverse operation, a vertex split, restores the original vertices and replaces the removed triangles. The PMs are used for fine-grained local refinement and to compute an error-bounded simplification of each cluster at runtime. In practice, refining a PM is a very fast operation since PM can be stored in an array and can be refined without checking any dependency, which is typically required to guarantee correct LOD representation.

CHPM representation provides two levels of refinement for interactive view-dependent rendering of massive models. First we perform a coarse-grained refinement at the cluster level. Next we refine the PMs of the selected clusters. The PM refinement provides smooth LOD transitions.

This representation has been implemented as a system called *Quick-VDR* on a commodity PC. This system was demonstrated with several models including a complex CAD environment (12M triangles), scanned models (372M triangles), and an isosurface (100M triangles). Moreover, they were able to render these models at $10 - 35$ frames per second using a limited memory footprint of $400 - 600$MB.

2.2 Ray Tracing

LOD-based algorithms can also be applied to accelerate ray tracing. One of major issues on designing multi-resolution techniques for ray tracing is that LOD computations such as LOD metric evaluation have to be performed for each ray including primary and secondary rays unlike rasterization.

Christensen *et al.* [2003] introduce a LOD approach for an offline ray tracer based on ray differentials [Igehy 1999]. Wand and Straßer [2003] propose an algorithm for multi-resolution ray tracing of point-sampled geometry based on ray-differentials. Another approach is to integrate the LODs into the hierarchical structure [Wiley et al. 1997]. Recently, Stoll *et al.* [2006] proposed a novel architecture for dynamic multiresolution ray tracing. They proposed a watertight multiresolution method by interpolating between discrete LODs for each ray. Their discrete LODs are computed from choosing proper tessellation levels for subdivision meshes. Also, efficient algorithms based on depth images can be used to accelerate ray tracing [Lischinski and Rappoport 1998; Agrawala et al. 2000].

Also, Yoon et al. [2006] proposed a simple, but efficient, multi-resolution based ray tracing technique. This method uses both a multi-resolution representation, called R-LODs, and an efficient LOD selection mechanism controlled by a user-specified geometric error bound. They demonstrated up to two orders of the performance improvement, especially when we deal with models of multi-gigabyte data size. The main performance improvement is caused by drastic reduction on expensive disk I/O accesses. Although this multi-resolution method is shown with kd-trees, it can be applicable to BVH-based ray tracers [Wald et al. 2006; Lauterbach et al. 2006]. Fig. 2 shows St. Matthew model computed by R-LOD representations.

Collision detection: Collision detection is one of the fundamental tools in many different applications such as human-computer interaction, simulation, and computer games. Collision detection performs intersection queries

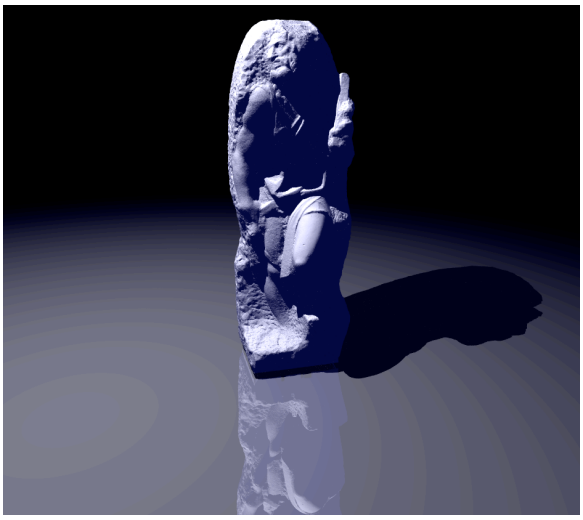


Figure 2: **St. Matthew Model:** *We use LOD-based algorithm to accelerate ray tracing of the St. Matthew model with shadows and reflections. We render the* $128M$ *triangle model at* 512×512 *resolution with* 2×2 *anti-aliasing and pixels-of-error* $= 4$*. We are able to achieve* $2 - 3$ *frames per second on two dual-core Xeon processors with 4GB of memory. We observe a* $2 - 20$ *times increase in the frame rate due to R-LODs with very little loss in image quality.*

between two objects. Ray tracing also can be seen to perform intersection queries, especially, between a ray and an object. To improve the performance of collision detection between massive models, there have been several LOD based collision detection methods [Otaduy and Lin 2003; Yoon et al. 2004a]. They applied the approximate collision detection based on LOD into many applications including dynamic simulation, navigation, and path planning.

3 Cache-Efficient Layouts

Since a major trend over the last few decades has been the widening gap between processor speed and data access speed, system architectures increasingly use caches and memory hierarchies to avoid memory latency. For example in GPU, there is a small GPU vertex cache to improve the performance of rendering triangles. To maximize the GPU vertex cache utilization, there have been a lot of work computing triangle strips or rendering sequences [Deering 1995; Hoppe 1999; Diaz-Gutierrez et al. 2005; Nehab and Sander 2006] and computing rendering sequences for view-dependent rendering [Karni et al. 2002; Yoon et al. 2005a].

One of the main characteristics of memory hierarchies of CPUs and GPUs is the use of block fetching whenever there is a cache miss. Therefore, it is critical to compute data that are likely accessed together and store them in each cache block. Recently, assuming this general I/O model of block-based caches, cache-efficient layouts for polygonal meshes [Yoon et al. 2005a; Yoon and Lindstrom 2006] and bounding volume hierarchies and spatial partitioning hierarchies [Yoon and Manocha 2006] have been proposed. Major goals of this work are to derive probabilistic models that measure the expected number of cache misses during runtime traversal on data. Based on these probabilistic models, they can compute cache-efficient layouts minimizing the number of cache misses during various applications.

Polygonal meshes: Given a directed graph representing access patterns of an application on the data, cache-aware and cache-oblivious metric have been proposed [Yoon and Lindstrom 2006]. These metrics showed strong correlation with the number of cache misses during random walks on the input graphs that well represent access patterns between data. Since many applications access polygonal meshes by accessing edges between vertices, we can represent the graph as an input polygonal model (e.g., triangular mesh). In a cache-aware case, the expected number of cache misses during random walks on the input graph has been shown to be a function of the number of

edges whose two nodes are stored in different cache blocks in the layout. For cache-oblivious cases, a geometric mean of edge lengths of a layout given the graph has high correlation with the expected number of cache misses. Once cache-aware or cache-oblivious metrics have been derived, computing cache-efficient layouts is reduced to a combinatorial optimization problem. Moreover, multi-level construction methods have been shown to be a good heuristic that can efficiently construct such cache-efficient layouts.

Bounding volume hierarchies: Bounding volume hierarchies (BVHs) are widely used to accelerate the performance of many geometric processing and interactive graphics applications including ray tracing, visibility culling, and collision detection. Most of these algorithms precompute a BVH and traverse the hierarchy at runtime to perform intersection tests or culling. A novel algorithm to compute cache-efficient layouts of BVHs of large models has been proposed [Yoon and Manocha 2006]. This approach is cache-oblivious as it does not require any knowledge of cache parameters or block sizes of the memory hierarchy and is applicable to all kinds of BVHs and spatial partitioning hierarchies that can be represented as a tree. This method is based on a new probabilistic model to predict the runtime access patterns of BVHs based on the geometric relationship between bounding volume information of bounding volume nodes. Also, this layout method has been able to achieve performance improvement on ray tracing and collision detection and show consistent performance improvement over the other well-known layouts.

4 Compression

Data compression is an orthogonal method to further reduce bandwidth and on-disk storage requirements. Data compression in computer graphics mainly focused on reducing file sizes or transmission time over slow networks. However, data compression has been explored to reduce the runtime performance of applications by reducing expensive disk I/O time [Lindstrom and Isenburg 2006; Burtscher and Ratanawoabhan 2007].

In mesh compression, there are a few methods providing random accesses on the compressed meshes to directly use the compressed meshes for runtime applications. These methods were demonstrated with rendering application [Choe et al. 2004; Kim et al. 2006]. These methods can provide partial decompression to provide random accesses for rendering application, while other existing compression techniques designed for optimal compression ratio require the entire decompression to be used for runtime applications, However, many compression techniques including methods mentioned above does not preserve an input order of triangles and vertices of a mesh. Therefore, although meshes are stored in cache-coherent layouts explained in Sec. 3, these layouts are not preserved during mesh compression and decompression. To address this problem and support random accesses on the compressed meshes for wider set of applications, Yoon and Lindstrom [2007] proposed an order-preserving mesh compression method. This method takes an streaming mesh with a finalization information as an input data and implicitly decomposes the mesh into a set of *clusters*, which are main granularities to providing random accesses. Moreover, this method presents a runtime framework that dynamically reconstruct connectivity information as decompressing a requested portion of the mesh and proposes general mesh traversal APIs. Therefore, these compressed meshes can be used for various applications. Also, they were able to achieve compression rates up to 29:1 compared to original uncompressed meshes and show 2 times runtime performance improvement of iso-contour application by reducing expensive disk I/O accesses.

Also, there have been several approach to compactly represent hierarchies for ray tracing [Mahovsky 2005; Cline et al. 2006]. Lauterbach et al. [2007] proposed *Ray-Strips* as a compact mesh and hierarchy representation for ray tracing. Ray-Strips is similar to triangle strips, which is used for accelerating the performance of rendering triangles. By using representing an input mesh with Ray-Strips, they reduce 80% of data sizes of prior representations with a minor performance degradation.

5 Conclusion

In this course note, we have discussed three orthogonal methods, 1) multi-resolution method to reduce amount of data necessary performing certain applications, 2) cache-coherent layouts to reduce the number of cache misses during accessing data, and 3) compression methods further reducing expensive disk I/O accesses, to efficiently deal with massive models on commodity hardware that has limited main memory.

References

AGRAWALA, M., RAMAMOORTHI, R., AND MOLL, A. 2000. Efficient image-based methods for rendering soft shadows. In *ACM SIGGRAPH*, 375–384.

BURTSCHER, M., AND RATANAWOABHAN, P. 2007. High throughput compression of double-precision floating-point data. In *IEEE Data Compression Conference*, 293–302.

CHOE, S., KIM, J., LEE, H., LEE, S., AND SEIDEL, H.-P. 2004. Mesh compression with random accessibility. In *Proc. of 5th Israel-Korea Bi-National conference*, 81–86.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum 22*, 3 (Sept.), 543–552.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2004. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph. 23*, 3, 796–803.

CLARK, J. H. 1976. Hierarchical geometric models for visible surface algorithms. *Commun. ACM 19*, 547–554.

CLINE, D., STEELE, K., AND EGBERT, P. K. 2006. Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools: JGT*.

DEERING, M. F. 1995. Geometry compression. In *ACM SIGGRAPH*, 13–20.

DIAZ-GUTIERREZ, P., BHUSHAN, A., GOPI, M., AND PAJAROLA, R. 2005. Constrained strip generation and management for efficient interactive 3d rendering. In *Computer Graphics International*, 115–121.

EL-SANA, J., AND VARSHNEY, A. 1999. Generalized view-dependent simplification. *Computer Graphics Forum*, C83–C94.

FLORIANI, L. D., MAGILLO, P., AND PUPPO, E. 1998. Efficient implementation of multi-triangulations. In *IEEE Visualization*.

FUNKHOUSER, T. A., AND SQUIN, C. H. 1993. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, J. T. Kajiya, Ed., vol. 27, 247–254.

GOBBETTI, E., AND MARTON, F. 2005. Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Trans. Graph. 24*, 3, 878–885.

HOPPE, H. 1996. Progressive meshes. In *Proc. of ACM SIGGRAPH*, 99–108.

HOPPE, H. 1997. View dependent refinement of progressive meshes. In *ACM SIGGRAPH Conference Proceedings*, 189–198.

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. *ACM SIGGRAPH*, 269–276.

IGEHY, H. 1999. Tracing ray differentials. In *ACM SIGGRAPH*, 179–186.

KARNI, Z., BOGOMJAKOV, A., AND GOTSMAN, C. 2002. Efficient compression and rendering of multi-resolution meshes. In *IEEE Visualization*, 347–54.

KIM, J., CHOE, S., AND LEE, S. 2006. Multiresolution random accessible mesh compression. *Computer Graphics Forum (Eurographics 2006) 25*, 3, 323–332.

LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs.

LAUTERBACH, C., EUI YOON, S., AND MANOCHA, D. 2007. Ray-strips: A compact mesh representation for interactive ray tracing. Tech. Rep. TR07-003, Dept. of CS, Univ. of North Carolina at Chapel Hill.

LINDSTROM, P., AND ISENBURG, M. 2006. Fast and efficient compression of floating-point data. *Transactions on Visualization and Computer Graphics 12*, 5, 1245–1250.

LINDSTROM, P., KOLLER, D., RIBARSKY, W., HUGHES, L. F., FAUST, N., AND TURNER, G. 1996. Real-Time, continuous level of detail rendering of height fields. In *SIGGRAPH 96 Conference Proceedings*, Addison Wesley, H. Rushmeier, Ed., Annual Conference Series, ACM SIGGRAPH, 109–118. held in New Orleans, Louisiana, 04-09 August 1996.

LISCHINSKI, D., AND RAPPOPORT, A. 1998. Image-based rendering for non-diffuse synthetic scenes. In *Eurographics Rendering Workshop 98*, 301–314.

LUEBKE, D., AND ERIKSON, C. 1997. View-dependent simplification of arbitrary polygon environments. In *Proc. of ACM SIGGRAPH*.

LUEBKE, D., REDDY, M., COHEN, J., VARSHNEY, A., WATSON, B., AND HUEBNER, R. 2002. *Level of Detail for 3D Graphics*. Morgan-Kaufmann.

MAHOVSKY, J. 2005. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary.

NEHAB, D., AND SANDER, J. B. P. V. 2006. Triangle order optimization for graphics hardware computation culling. *Symposium on Interactive 3D Graphics and Games*.

OTADUY, M. A., AND LIN, M. C. 2003. CLODs: Dual hierarchies for multiresolution collision detection. *Eurographics Symposium on Geometry Processing*, 94–101.

SILVA, C., CHIANG, Y.-J., CORREA, W., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization Course Notes*.

STOLL, G., MARK, W. R., DJEU, P., WANG, R., AND ELHASSAN, I. 2006. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Tech. Rep. TR-06-21, Dept. of CS, Univ. of Texas.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2006. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*.

WAND, M., AND STRASSER, W. 2003. Multi-resolution point-sampled raytracing. In *Graphics Interface*.

WILEY, C., A.T. CAMPBELL, I., SZYGENDA, S., FUSSELL, D., AND HUDSON, F. 1997. Multiresolution BSP trees applied to terrain, transparency, and general objects. In *Graphics Interface*, 88–96.

XIA, J., EL-SANA, J., AND VARSHNEY, A. 1997. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (June), 171–183.

YOON, S.-E., AND LINDSTROM, P. 2006. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2006) 12*, 5 (September-October).

YOON, S.-E., AND LINDSTROM, P. 2007. Random-Accessible Compressed Meshes. Tech. Rep. UCRL-JRNL-229656, Lawrence Livermore National Lab.

YOON, S.-E., AND MANOCHA, D. 2006. Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum (Eurographics) 25*, 507–516.

YOON, S., SALOMON, B., LIN, M. C., AND MANOCHA, D. 2004. Fast collision detection between massive models using dynamic simplification. *Eurographics Symposium on Geometry Processing*, 136–146.

YOON, S.-E., SALOMON, B., GAYLE, R., AND MANOCHA, D. 2004. Quick-VDR: Interactive View-dependent Rendering of Massive Models. *IEEE Visualization*, 131–138.

YOON, S.-E., LINDSTROM, P., PASCUCCI, V., AND MANOCHA, D. 2005. Cache-Oblivious Mesh Layouts. *Proc. of ACM SIGGRAPH*.

YOON, S.-E., SALOMON, B., GAYLE, R., AND MANOCHA, D. 2005. Quick-VDR: Out-of-Core View-Dependent Rendering of Gigantic Models. *IEEE Transactions on Visualization and Computer Graphics*, 369–382.

YOON, S.-E., LAUTERBACH, C., AND MANOCHA, D. 2006. R-LODs: Interactive LOD-based Ray Tracing of Massive Models. *The Visual Computer (Pacific Graphics)*.

© [2005] IEEE. Reprinted, with permission, from:

Quick-VDR: out-of-core view-dependent rendering of gigantic models

Yoon, S.-E.; Salomon, B.; Gayle, R.; Manocha, D.;
Visualization and Computer Graphics, IEEE Transactions on
Volume 11, Issue 4, July-Aug. 2005 Page(s):369 - 382

Quick-VDR: Out-of-Core View-Dependent Rendering of Gigantic Models

Sung-Eui Yoon, Brian Salomon, Russell Gayle, and Dinesh Manocha, *Member*, *IEEE*

Abstract—We present a novel approach for interactive view-dependent rendering of massive models. Our algorithm combines view-dependent simplification, occlusion culling, and out-of-core rendering. We represent the model as a clustered hierarchy of progressive meshes (CHPM). We use the cluster hierarchy for coarse-grained selective refinement and progressive meshes for fine-grained local refinement. We present an out-of-core algorithm for computation of a CHPM that includes cluster decomposition, hierarchy generation, and simplification. We introduce novel cluster dependencies in the preprocess to generate crack-free, drastic simplifications at runtime. The clusters are used for LOD selection, occlusion culling, and out-of-core rendering. We add a frame of latency to the rendering pipeline to fetch newly visible clusters from the disk and avoid stalls. The CHPM reduces the refinement cost of view-dependent rendering by more than an order of magnitude as compared to a vertex hierarchy. We have implemented our algorithm on a desktop PC. We can render massive CAD, isosurface, and scanned models, consisting of tens or a few hundred million triangles at 15-35 frames per second with little loss in image quality.

Index Terms—Interactive display, view-dependent rendering, occlusion culling, external-memory algorithm, out-of-core algorithms, levels-of-detail.

1 Introduction

Recent advances in acquisition, modeling, and simulation technologies have resulted in large databases of complex geometric models. These gigabyte-sized data sets consist of tens or hundreds of millions of polygons. The enormous size of these data sets poses a number of challenges in terms of interactive display and manipulation on current graphics systems.

View-dependent simplification and rendering have been actively researched for interactive display of large data sets [1], [2], [3]. These algorithms have many appealing properties because they compute different levels-of-detail (LODs) over different regions of the model. The selection of appropriate LODs is based on view-position, local illumination, and other properties, such as visibility and silhouettes. Most view-dependent algorithms precompute a vertex hierarchy of the model and perform incremental computations between successive frames. This reduces the "popping" artifacts that can occur while switching between different LODs. The algorithms generally maintain a cut, or *active vertex front*, across the hierarchy and use it for mesh refinement. The front is traversed each frame and is updated based on the change in view parameters. In order to preserve the local topology, dependencies are introduced between simplification operations.

Current representations and refinement algorithms for view-dependent rendering do not scale well to large models composed of tens or hundreds of millions of triangles. The refinement cost is a function of the front size and may be prohibitively expensive for massive models. Furthermore, resolving dependencies in the vertex hierarchy can be expensive (e.g., hundreds of milliseconds or more per frame).

In addition to reducing the refinement cost, it is necessary to integrate view-dependent simplification algorithms with occlusion culling and out-of-core rendering. Occlusion culling computes a set of potentially visible primitives during each frame and is needed to handle high depth complexity models. Out-of-core rendering techniques operate with a bounded memory footprint and are required to render massive models on commodity graphics systems with limited memory. Algorithms for occlusion culling and out-of-core techniques also perform computations based on the view parameters. However, no known algorithms integrate conservative occlusion culling and out-of-core rendering with vertex hierarchies.

1.1 Main Contributions

We present a new view-dependent rendering algorithm (Quick-VDR) for interactive display of massive models. The main contributions of the paper can be classified as the following:

1. **Model representation:** We use a novel scene representation, a *clustered hierarchy of progressive meshes* (CHPM). The cluster hierarchy is used for coarse-grained view-dependent refinement. The PMs provide fine-grained local refinement to reduce the popping between successive frames without high refinement cost.
2. **Construction algorithms:** Quick-VDR relies on an out-of-core algorithm to compute a CHPM that performs a hierarchical cluster decomposition and simplification. We introduce the concept of *cluster dependencies* between adjacent clusters to generate drastic crack-free simplifications of the original model.

• *The authors are with the Department of Computer Science, Campus Box 3175, Sitterson Hall, University of North Carolina-Chapel Hill, Chapel Hill, NC 27599. E-mail: {sungeui, salomon, rgayle, dm}@cs.unc.edu.*

Manuscript received 16 Sept. 2004; revised 31 Dec. 2004; accepted 7 Feb. 2005; published online 10 May 2005.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-0107-0904.

1077-2626/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

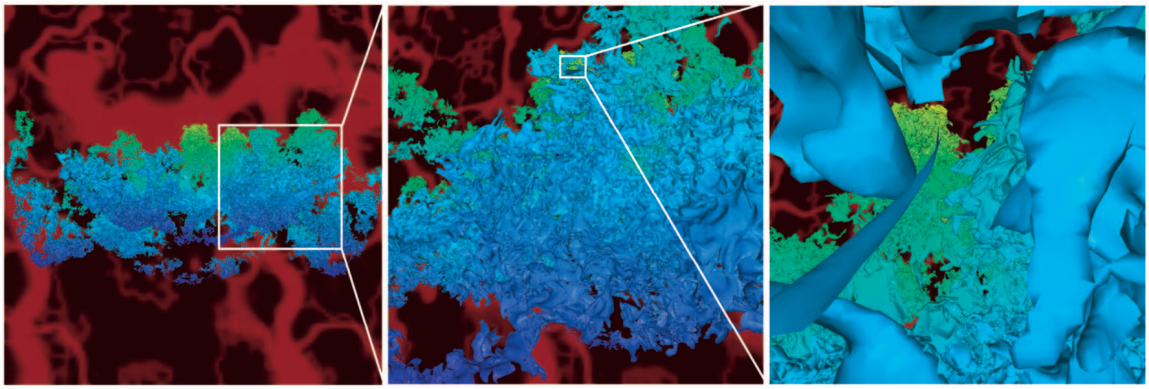


Fig. 1. These images show the application of Quick-VDR to a complex isosurface (100M triangles) generated from a very high resolution 3D simulation of Richtmyer-Meshkov instability and turbulence mixing. The middle and right images show zoomed views. The isosurface has high depth complexity, holes, and a very high genus. Quick-VDR can render it at 15-40 frames per second on a PC with NVIDIA GeForce 5950FX Ultra card and uses a memory footprint of 600MB.

3. **Rendering algorithms:** Our rendering algorithm uses temporal coherence and occlusion queries for visibility computations at the cluster level. We account for visibility events between successive frames by combining fetching and prefetching techniques for out-of-core rendering. Our rendering algorithm introduces one frame of latency to allow newly visible clusters to be fetched without stalling the pipeline.
4. **Implementation and Application:** We have implemented and tested Quick-VDR on a commodity PC with NVIDIA 5950FX Ultra card. To illustrate the generality of our approach, we have highlighted its performance on several models: a complex CAD environment (12M triangles), scanned models (372M triangles), and an isosurface (100M triangles). We can render these models at 15-35 frames per second using a limited memory footprint of 400-600 MB.

1.2 Advantages

Our approach integrates view-dependent simplification, conservative occlusion culling, and out-of-core rendering for high quality interactive display of massive models on current graphics systems. As compared to prior approaches, Quick-VDR offers the following benefits:

1. **Lower refinement cost:** The overhead of view-dependent refinement in the CHPM is one to two orders of magnitude lower than vertex hierarchies for large models.
2. **Massive models:** We are able to compute drastic simplifications of massive models, using hierarchical simplification with cluster dependencies, necessary for interactive rendering.
3. **Runtime performance:** Quick-VDR renders CHPMs using a bounded memory footprint and exploits the features of current graphics processors to obtain a high frame rate.
4. **Rendering quality:** We significantly improve the frame rate with little loss in image quality and alleviate popping artifacts between successive frames.
5. **Generality:** Quick-VDR is a general algorithm and applicable to all types of polygonal models, including CAD, scanned, and isosurfaces.

This paper extends our previous work [4] and presents a more detailed exposition of the representations, algorithms, and implementation issues. We also perform an extensive comparison with prior techniques and present a detailed analysis of our algorithms. Moreover, we describe several novel optimization techniques to improve the performance of preprocessing and runtime algorithms. As a result, we are able to obtain 80-150 percent improvement in the preprocessing time and 20-80 percent improvement in the running time as compared to the algorithms presented in [4].

1.3 Organization

The rest of the paper is organized in the following manner: We give a brief overview of related work in Section 2 and describe our scene representation and refinement algorithm in Section 3. Section 4 describes our out-of-core algorithm to generate a CHPM for a large environment. We present the rendering algorithm in Section 5 and highlight its performance in Section 6. We compare our algorithm with other approaches in Section 7 and discuss some of its limitations.

2 Related Work

We give a brief overview of the previous work in view-dependent rendering, out-of-core rendering, occlusion culling, and hybrid approaches to massive model rendering.

2.1 View-Dependent Simplification

View-dependent simplification of complex models has been an active area of research over the last decade. View-dependent rendering originated as an extension of the progressive mesh (PM) [5]. A PM is a linear sequence of increasingly coarse meshes built from an input mesh by repeatedly applying edge collapse operations. It provides a continuous resolution representation of an input mesh and is useful for efficient storage, rendering, and transmission.

Xia et al. [3] and Hoppe [1] organized the PM as a vertex hierarchy (or view-dependent progressive mesh (VDPM)) instead of a linear sequence. Luebke and Erikson [2] developed a similar approach employing octree-based vertex clustering operations and used it for dynamic

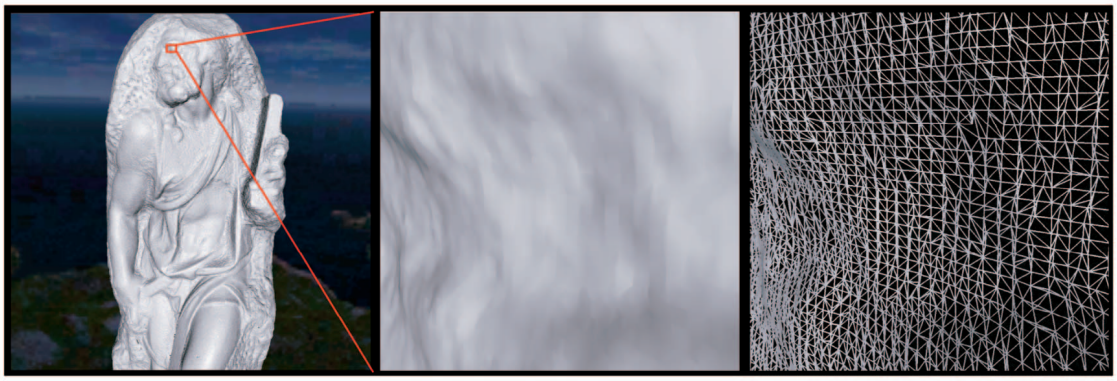


Fig. 2. Scan of Michelangelo's St. Matthew. This 9.6 GB scanned model consists of 372M triangles. The middle image is a zoomed view and the right image shows its triangulation. Quick-VDR is able to render this model at 13-25 frames per second on a dual Pentium IV PC with a GeForce 5950FX Ultra GPU using a memory footprint of 600 MB.

simplification. The Multi-Triangulation (MT) is a multi-resolution representation that has been used for view-dependent rendering [6]. All possible simplifications are explicitly represented in an MT. This property has been shown useful for perceptually guided simplification [7]. Duchaineau et al. [8] presented a view-dependent simplification algorithm for terrain models.

2.1.1 Acceleration Techniques

Many techniques have been presented to improve the performance of view dependent rendering algorithms. El-Sana and Varshney [9] used a uniform error metric based on cubic interpolants and reduced the cost of runtime tests. They also proposed *implicit dependencies* to improve refinement performance by reducing nonlocal memory access. El-Sana et al. [10] applied skip lists to improve the rendering performance of view-dependent rendering. Pajarola [11] improved the update rate of runtime mesh selection by exploiting properties of the half-edge mesh representation and applied it to manifold objects. Bogomjakov and Gotsman [12] presented novel universal sequences to improve rendering performance of progressive meshes. El-Sana and Bachmat [13] present a mesh refinement prioritization scheme to improve the runtime performance.

2.2 Out-of-Core Simplification and Rendering

Many algorithms have been proposed for out-of-core simplification. These include [14], [15], [16] for generating static LODs. Hoppe [17] extended the VDPM framework for terrain rendering by decomposing the terrain data into blocks, generating a block hierarchy, and simplifying each block independently. Prince [18] extended this out-of-core terrain simplification to handle arbitrary polygonal models.

El-Sana and Chiang [19] segment the input mesh into submeshes such that the boundary faces are preserved while performing edge-collapse operations. DeCoro and Pajarola [20] present an external data structure for the half-edge hierarchy and an explicit paging system for out-of-core management of view-dependent rendering. Lindstrom [21] presents an end-to-end approach for out-of-core simplification and view-dependent visualization. It is based on memory insensitive simplification [22] and has been applied to scanned models and isosurfaces.

Cignoni et al. [23] present Adaptive TetraPuzzles, which builds a hierarchy of tetrahedrons and parallelized the computation of static LODs for nodes of the hierarchy. In Section 7, we compare our approach with the Adaptive TetraPuzzles algorithm.

2.3 Occlusion Culling

The problem of computing portions of the scene visible from a given viewpoint has been well-studied [24]. Occlusion culling algorithms may be classified as region or point-based, image or object space, and conservative or approximate. Many specialized object-space algorithms have been developed for architectural models or urban environments.

For general environments, object space algorithms make use of spatial partitioning or bounding volume hierarchies [25], [26]; however, performing "occluder fusion" on scenes composed of small occluders with object space methods is difficult. Image-based occlusion representations are widely used and the resulting algorithms use graphics hardware to perform visibility computations [27], [28], [29]. Also, image-based occlusion representations are generally more capable of capturing occluder fusion. In some cases, additional graphics processors have been used for visibility computations [30], [31]. All of these algorithms load the entire scene graph into main memory.

2.4 Hybrid Algorithms for Rendering Acceleration

Many hybrid algorithms have been proposed that combine model simplification with visibility culling or out-of-core data management. The Berkeley Walkthrough system [32] combines cells and portals based on visibility computation algorithms with static LODs for architectural models. The MMR system [33] combines static LODs with occlusion culling and out-of-core computation and is applicable to models that can be partitioned into rectangular cells.

Other approaches combining precomputed static LODs and conservative occlusion culling have been proposed [31], [34], [35]. These algorithms represent the environment as a scene graph, precompute HLODs (hierarchical levels-of-detail) [36] for intermediate nodes, and use them for occlusion culling. However, switching between static LODs and HLODs can cause popping. Moreover, these algorithms use additional graphics processors to perform occlusion queries and introduce one frame of latency in the overall pipeline.

Wald et al. [37] combine out-of-core management with ray tracing and use volumetric approximation for unloaded

geometry to improve out-of-core rendering performance. Govindaraju et al. [31] use hierarchies of static LODs (HLODs) and conservative occlusion culling for interactive display of large CAD environments. Yoon et al. [38] presented an in-core algorithm to combine view-dependent simplification with conservative occlusion culling. El-Sana et al. [39] combined view-dependent rendering with approximate occlusion culling. The *iWalk* system [40], [41] partitions the space into cells and performs out-of-core rendering of large architectural and CAD models on commodity hardware using approximate and conservative occlusion culling.

2.4.1 Point-Based Rendering

An alternative rendering method to a traditional polygonal rendering is point-based rendering. The QSplat system [42] uses a compact bounding volume hierarchy of spheres for view-frustum and backface culling, level-of-detail control, and point-based rendering. It has been applied to large scanned models and works very well in practice. It is not clear whether point-based rendering algorithms would work on CAD models with sharp features or edges. Moreover, current graphics systems are well optimized to rasterize triangulated models. Recently, Dachsbacher et al. [43] exploited the programmability features of current GPUs to improve the rendering of performance of point primitives.

3 Overview

In this section, we introduce some of the terminology and representations used by Quick-VDR. We also give a brief overview of our approach for out-of-core hierarchical simplification and rendering.

3.1 View-Dependent Rendering of Massive Data Sets

Most of the prior work on view-dependent simplification and rendering of large data sets uses vertex hierarchies such as VDPM [1]. These approaches augment each edge collapse with *dependency* information related to the local neighborhood at the time of the edge collapse during construction. This information is used to prevent "fold-overs" whereby a face normal is reversed at runtime. However, many issues arise in applying these approaches to massive data sets composed of tens or hundreds of millions of triangles. Traversing and refining an *active vertex front* across a vertex hierarchy composed of tens of millions of polygons can take hundreds of milliseconds per frame. Resolving the dependencies can lead to nonlocalized memory accesses which can be problematic for out-of-core rendering. Moreover, performing occlusion culling and out-of-core rendering using vertex hierarchies can become expensive.

3.2 Scene Representation

We propose a novel representation, a clustered hierarchy of progressive meshes (CHPM), for view-dependent rendering of massive data sets. The CHPM consists of two parts:

Cluster Hierarchy. We represent the entire data set as a hierarchy of clusters, which are spatially localized mesh regions. Each cluster consists of a few thousand triangles. The clusters provide the capability to perform coarse-grained view-dependent (or selective) refinement of the model. They are also used for visibility computations and out-of-core rendering.

Progressive Mesh. We precompute a simplification of each cluster and represent linear sequence of edge collapses as a progressive mesh (PM). The PMs are used for fine-grained local refinement and to compute an error-bounded simplification of each cluster at runtime.

We refine the CHPM at two levels. First, we perform a coarse-grained refinement at the cluster level. Next, we refine the PMs of the selected clusters. The PM refinement provides smooth LOD transitions.

3.2.1 Cluster Hierarchy

Conceptually, a cluster hierarchy is similar to a vertex hierarchy. However, every node of a cluster hierarchy represents a set of vertices and faces rather than a single vertex. At runtime, we maintain an *active cluster list* (ACL), which is similar to an active front in a vertex hierarchy, and perform selective refinement on this list via the following operations:

- **Cluster-split:** A cluster in the active cluster list is replaced by its children.
- **Cluster-collapse:** Sibling clusters are replaced by their parent.

These operations are analogous to the vertex split and collapse in a vertex hierarchy, but provide a more coarse-grained approach to selective refinement.

3.2.2 Progressive Meshes and Refinement

Each cluster contains a PM, which is a mesh sequence built from an input mesh by a sequence of edge collapse operations. The inverse operation, a vertex split, restores the original vertices and replaces the removed triangles. Each PM is stored as the most simplified or *base mesh* combined with a series of vertex split operations. In practice, refining a PM is a very fast operation and requires no dependency checks.

We use the notation M_A^0 to represent a base mesh of a cluster A. Moreover, M_A^i is computed by applying a vertex split operation to M_A^{i-1}. A PM can be refined within a range of object space error values. We refer to this range as the *error range* of a cluster and is expressed as a pair: (*min bound*, *max error*). The *min bound* is the error value associated with the base mesh (M^0) and the *max error* is the error value associated with the highest resolution mesh (e.g., M_C^k, M_A^i, and M_B^j (as shown in Fig. 3).

The PMs allow us to perform smooth LOD transitions at the level of a single cluster. In order to perform globally smooth LOD transitions, we require that the changes to the ACL between successive frames are also smooth. If cluster C is the parent of clusters A and B, we set the highest resolution mesh approximation of cluster C's PM to be the union of the base meshes of cluster A and B's PMs. That is, $M_C^k = M_A^0 \bigcup M_B^0$ (see Fig. 3). Therefore, the cluster-collapse and cluster-split operations introduce no popping artifacts.

3.2.3 Dual Hierarchies

The CHPM representation can be seen as dual hierarchies: an LOD hierarchy for view-dependent rendering and a bounding volume hierarchy (BVH) for occlusion culling. As an LOD hierarchy, each interior cluster contains a coarser representation of its children's meshes. As a bounding

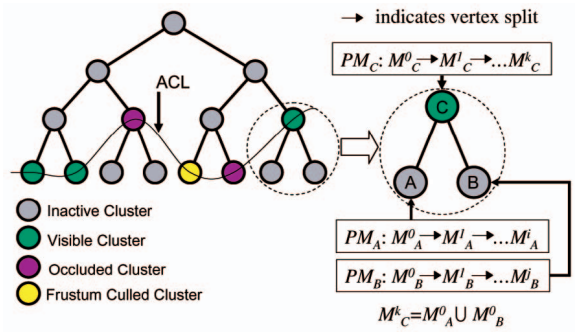


Fig. 3. CHPM: Clustered Hierarchy of Progressive Meshes. At runtime, the active cluster list (ACL) represents a front in the cluster hierarchy containing the clusters of the current mesh (left). Clusters on the ACL are classified as visible, frustum-culled, or occlusion-culled. The PMs (right) of visible clusters are refined to meet the screen space error bound by selecting a mesh from the PM mesh sequence. When the ACL changes, smooth LOD transitions occur because the most refined mesh of each PM is equal to the union of the base meshes of its children.

volume hierarchy, each cluster has an associated bounding volume (BV) which contains all the mesh primitives represented by its subtree. We use the oriented bounding box as the BV representation.

By combining an LOD hierarchy with a BVH, we are able to improve a memory requirement of the representation and simplify the integration of view-dependent rendering and occlusion culling.

3.3 Algorithms

Quick-VDR consists of two major parts: a preprocess and a runtime rendering algorithm.

3.3.1 Preprocess

Given a large data set, we compute a CHPM representation. Our out-of-core algorithm begins by decomposing the input mesh into a set of clusters. The clusters are passed to a cluster hierarchy generation algorithm, which builds a balanced hierarchy in a top-down manner. We perform out-of-core hierarchical simplification using the cluster hierarchy as a final step. We introduce *cluster dependencies* that allow boundary simplification while maintaining crack-free simplification and achieving efficient rendering performance at runtime.

3.3.2 Rendering Algorithm

Quick-VDR uses the CHPM as a scene representation for out-of-core view-dependent rendering and occlusion culling. The CHPM is refined by performing two levels of refinement: a coarse-grained refinement at the cluster level and a fine-grained local refinement using a PM. Cluster dependencies assure that consistent cluster boundaries are rendered and that we are able to compute drastic simplifications. We use temporal coherence to accelerate refinement and to perform occlusion culling at the cluster level using hardware accelerated visibility queries. Quick-VDR uses the operating system's virtual memory manager through a memory mapped file for out-of-core rendering. In order to overcome the problem of accurately predicting the occlusion events, we introduce one frame of latency in the runtime pipeline. This allows us to load newly visible clusters to avoid stalling the rendering pipeline.

4 Building a CHPM

In this section, we present an out-of-core algorithm to compute CHPMs for large data sets, such as CAD models, large isosurfaces, or scanned models. Our algorithm proceeds in three steps. First, we decompose the input mesh into a set of clusters. The decomposition occurs in several passes to avoid loading the entire input mesh at once. These clusters facilitate out-of-core access to the mesh for the remaining steps. Next, we construct the cluster hierarchy by repeatedly subdividing the mesh in a top-down manner. Finally, we compute progressive meshes for each cluster by performing a bottom-up traversal of the hierarchy.

4.1 Cluster Decomposition

The clusters form the underlying representation for both the preprocessing step as well as out-of-core view-dependent rendering with occlusion culling. We decompose the model into clusters, which are spatially localized portions of the input mesh. The generated clusters should be nearly equally sized in terms of number of triangles for several reasons. This property is desirable for out-of-core mesh processing to minimize the memory requirements. If the cluster size as well as the number of clusters required in memory at one time are bounded, then simplification and hierarchy construction can be performed with a constant memory footprint. Moreover, enforcing spatial locality and uniform size provides higher performance for occlusion culling and selective refinement.

The out-of-core cluster decomposition algorithm proceeds in four passes. The first three passes only consider the vertices of the original model and create the clusters, while the fourth pass assigns the faces to the clusters. We use a variation of the cluster decomposition algorithm for out-of-core compression of large data sets presented in [44]. However, our goal is to decompose the mesh for out-of-core processing and view-dependent rendering. As a result, we need only compute and store the connectivity information used by the simplification algorithm. To support transparent accesses on a large mesh during simplification, we also preserve all intercluster connectivity information.

4.1.1 Connectivity

It is desirable to have compact connectivity and easy access to the connectivity of out-of-core meshes during simplification. To meet these goals, we use corner-based connectivity for out-of-core meshes. A triangle consists of three corners, each of which has a index to an incident vertex and an index for the next corner that shares the same incident vertex. Each vertex also has the index information. Since the vertices and triangles are grouped into clusters, we represent each index as two components: a cluster id and a local id. This index information can be packed in 4 bytes integer.

Given this connectivity information, we are able to support all the necessary operations (e.g., decimation operations) during simplification. Moreover, we can easily reconstruct the connectivity as we read triangles from the disk without storing them explicitly in the main memory.

4.1.2 Algorithm

The out-of-core cluster decomposition algorithm proceeds in four passes. The four passes of the algorithm are:

First vertex pass: We compute the bounding box of the mesh.

Fig. 4. An example of cluster hierarchy: These images highlight different clusters of the hierarchy of the dragon model. The leaf clusters are shown in the top left image. The root cluster is shown in the top right, the second level clusters are shown in the bottom right, and the third level clusters are shown in the bottom left.

Second vertex pass: We compute balanced-size clusters of vertices (e.g., 3K vertices). Vertices are assigned to cells of a uniform 3D grid which may be subdivided to deal with irregular distribution of geometry. A graph is built with nodes representing the nonempty cells weighted by vertex count. Edges are inserted between each cell and its k nearest neighbors using an approximate nearest neighbor algorithm [45] (e.g., $k = 6$). We use a graph partitioning algorithm [46] to partition the graph and compute balanced-size clusters.

Third vertex pass: Based on the output of the partitioning, we assign vertices to clusters and reindex the vertices. The new index is a cluster/vertex pair that is used to locate the vertex in the decomposition. A mapping is created that maps the original vertex indices to the new pair of indices. This mapping can be quite large, so it is stored in a file that can be accessed in blocks with LRU paging to allow the remainder of the preprocess to operate in a constant memory size.

Face pass: In the final pass, we assign each face to a single cluster that contains at least one of its vertices. The mapping file created in the previous pass is used to locate the vertices. The vertices of faces spanning multiple clusters are marked as constrained for simplification. These vertices make up the boundaries between clusters and are referred to as *shared vertices*, while the remaining vertices are referred to as *interior vertices*.

The resulting cluster decomposition consists of manageable mesh pieces that can be transparently accessed in an out-of-core manner for hierarchy generation and simplification, while preserving all the original connectivity information. Different clusters computed from the dragon model are shown in Fig. 4.

4.2 Cluster Hierarchy Generation

In this section, we present an algorithm to compute the cluster hierarchy. The clusters computed by the decomposition algorithm described in the previous section are used as the input to hierarchy generation. Our goal is to compute a hierarchy of clusters with the following properties:

Nearly equal cluster size. As previously discussed, consistent cluster size is important for memory management, occlusion culling, and selective refinement. Clusters at all levels of the hierarchy must possess this property.

Balanced cluster hierarchy. During hierarchical simplification, cluster geometry is repeatedly simplified and merged in a bottom up traversal. The hierarchy must be well-balanced so that merged clusters have nearly identical *error ranges*.

Minimize shared vertices. The number of shared vertices at the cluster boundary should be minimized for simplification. Otherwise, in order to maintain consistent cluster boundaries, the simplification will be overconstrained and may result in lower fidelity approximations of the original model.

High spatial locality. The cluster hierarchy should have high spatial locality for occlusion culling and selective refinement.

We achieve these goals by transforming the problem of computing a cluster hierarchy into a graph partitioning problem and computing the hierarchy in a top down manner. Each cluster is represented as a node in a graph, weighted by the number of vertices. Clusters are connected by an edge in the graph if they share vertices or are within a threshold distance of each other. The edges are weighted by the number of shared vertices and the inverse of the distance between the clusters, with greater priority placed on the number of shared vertices. The cluster hierarchy is then constructed in a top-down manner by recursively partitioning the graph into halves considering the weights, thus producing a binary tree. The weights guide the partitioning algorithm [47] to produce clusters with spatial locality while tending toward fewer shared vertices. The top down partitioning creates an almost balanced hierarchy. An example of the cluster hierarchy of the dragon model is shown in Fig. 4.

4.3 Out-of-Core Hierarchical Simplification

We simplify the mesh by traversing the cluster hierarchy in a bottom-up manner. Each level of the cluster hierarchy is simplified in a single pass, so the simplification requires $\lceil log_2(n) + 1 \rceil$ passes where n is the number of leaf clusters. During each pass, only the cluster being simplified and clusters with which it shares vertices must be resident in memory.

Simplification operations are ordered by a priority queue based upon quadric errors [48]. We build the progressive meshes (PMs) for each cluster by applying "half-edge collapses." The half-edge collapse, in which an edge is contracted to one of the original vertices, is used to avoid creation of new vertices during simplification. Edges adjacent to shared vertices are not collapsed during simplification. The edge collapses and associated error values are stored, along with the most refined mesh of a PM. After creating the PM, the *error range* of the cluster is computed based on the errors of the PM's original and base mesh.

When proceeding to the next level up the hierarchy, the mesh within each cluster's PM is initialized by merging the base meshes of the children. Constraints on vertices that are shared by two clusters being merged are removed, thereby allowing simplification of the merged boundary. Since the intermediate clusters should be nearly the same size as the leaf-level clusters, each cluster is simplified to half its original face count at each level of the hierarchy.

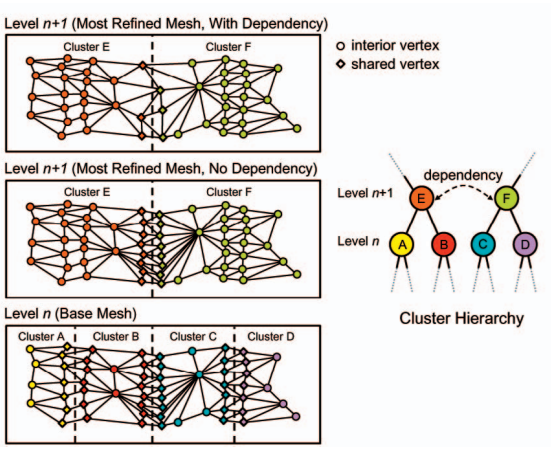


Fig. 5. Dependencies. After simplifying level n of the hierarchy, the boundaries AB, BC, and CD are all undersimplified because they are constrained. When initializing the base meshes of E and F prior to simplifying level $n+1$, two of these boundaries, AB and CD, are no longer constrained because they have been merged. The boundary BC was not merged and will remain undersimplified. We can, however, simplify the faces along this boundary if we mark E and F as dependent.

As simplification proceeds, a file is created for the progressive mesh of each cluster. However, handling many small files is inefficient at runtime. The PM files are merged into one file which can be memory mapped to allow the OS to perform memory management of the PMs and optimize disk access patterns during runtime rendering. The file is stored in a breadth first manner in an attempt to match the probable access pattern during runtime refinement.

4.4 Boundary Constraints and Cluster Dependencies

In order to support out-of-core rendering and to allow efficient refinement at runtime, it should be possible to refine the PM of each cluster independently and, at the same time, maintain a crack-free consistent mesh. To achieve this, our algorithm detects the shared vertices and restricts collapsing the edges adjacent to them during hierarchical simplification. As simplification proceeds up the hierarchy, these constraints are removed when the clusters sharing the vertices have been merged.

While these constraints assure crack-free boundaries between clusters at runtime, they can be overly restrictive. After simplifying several levels of the hierarchy most of the vertices in the base mesh of the PM are shared vertices. As illustrated in Fig. 5, this problem arises along boundaries between clusters that are merged at higher levels in the hierarchy. This can degrade the quality of simplification, and impedes drastic simplification. In Fig. 5, notice that the boundaries between clusters A and B and clusters C and D are merged in the next level of the hierarchy (E and F). However, the boundary between B and C is not merged until higher up the hierarchy, but is already drastically undersimplified compared to the interior. This constraint problem is common to many hierarchical simplification algorithms that decompose a large mesh for view-dependent rendering [17], [18] or compute hierarchies of static LODs (HLODs) [31].

We introduce *cluster-level dependencies* to address this constraint problem. The intuition behind dependencies is that precomputed simplification constraints on shared vertices can be replaced by runtime dependencies. During hierarchical simplification, we may collapse an edge adjacent to a shared vertex. The clusters sharing that vertex are marked as dependent upon each other. Boundary simplification occurs on the merged meshes prior to PM generation, thereby allowing the computed PMs to be refined independently at runtime. In Fig. 5, clusters E and F are marked dependent and thereby allow the boundary to be simplified.

At runtime, splitting a cluster forces all its dependent clusters to split so that the boundaries are rendered without cracks. Likewise, a parent cluster cannot be collapsed unless all of its dependent clusters have also been collapsed. In Fig. 5, clusters E and F must be split together and clusters A, B, C, and D must be collapsed together (assuming E and F are dependent). For example, if clusters B and F are rendered during the same frame, their boundary will be rendered inconsistently and may have cracks.

4.4.1 Dependencies Criteria

Although cluster dependencies allow boundary simplification, we need to use them carefully. Since splitting a cluster forces its dependent clusters to split, dependencies will cause some clusters to be rendered that are overly conservative in terms of view-dependent error bounds. Furthermore, the boundaries change in one frame, which may cause popping artifacts. This can be exacerbated by "chained" dependencies in which one cluster is dependent upon another cluster which is, in turn, dependent upon a third cluster, and so on.

To avoid these potential runtime problems, we prioritize clusters for boundary simplification. At each level of hierarchical simplification, the clusters are entered into a priority queue. Priorities are assigned as the ratio of average error of shared vertices to the average error of interior vertices. A cluster, A, is removed from the head of the priority queue. For each cluster, B, that shares at least j (e.g., five) vertices with A, we apply boundary simplification between A and B if the following conditions are met:

1. A and B will not be merged within a small number of levels up the cluster hierarchy (e.g., two).
2. A and B have similar *error ranges*.
3. A dependency between A and B will not introduce a chain (unless all the clusters in the chain share vertices).

This is repeated for each cluster in the priority queue. The first condition avoids creating dependencies between clusters that are resolved within only a few additional hierarchy levels. The second condition discourages dependencies between those clusters that are unlikely to be simultaneously present in the ACL at runtime. The third condition prevents long dependency chains and preserves selective refinement at the cluster level. The cluster dependencies ensure that a sufficient number of shared vertices are collapsed at each level of the hierarchy while still generating and rendering crack-free simplifications at runtime. An example of posing cluster dependencies in Lucy model is shown in Fig. 6.

4.5 Buffer-Based Processing

The hierarchical simplification algorithm described in the previous section carefully computes dependencies between

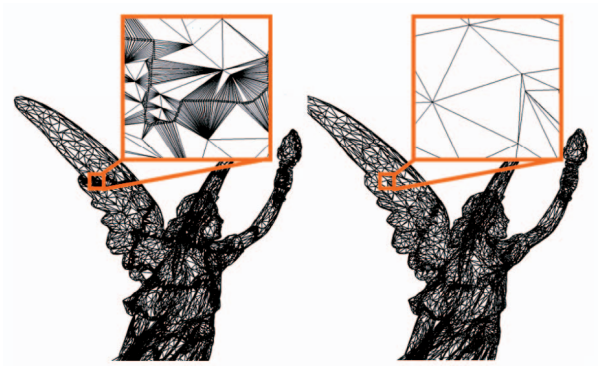


Fig. 6. An example of cluster dependencies. These images highlight meshes with and without posing cluster dependencies in the Lucy model. The left image shows a simplified Lucy model without the cluster dependencies. It consists of 227K triangles at the specific pixels of error. The right images show a simplified Lucy model with cluster dependencies. It is composed of only 19K triangles with the same pixels of error. Therefore, we are able to achieve more than one order of magnitude speedup on rendering time by using cluster dependencies. The zoomed areas are shown in pink rectangles.

the clusters for high quality simplification and fast rendering. However, this algorithm would access the clusters in a random order due to the priority queue representation. This can degrade the performance of the simplification preprocess. To overcome this problem, we apply *buffer-based processing* to improve the access pattern of the algorithm. The buffer is used to hold the clusters, which are input to the priority queue. The size of a buffer is limited by maximum memory used for preprocessing. As we traverse the clusters in a breadth first order, we allocate visited-clusters into the buffer and compute their priorities in the priority queue. Once the visited-clusters reach the maximum size of the buffer, the simplifications are processed according to the priorities within the priority queue. The priority queue is then emptied and processing is continued until all the clusters have been visited by the traversal algorithm.

5 Interactive Out-of-Core Display

In the previous section, we described an algorithm to compute CHPM. In this section, we present a novel rendering algorithm that uses CHPM representation for occlusion culling, view-dependent refinement, and out-of-core rendering. The entire representation, including the PMs, is stored on the disk. We load the coarse-grained cluster hierarchy into main memory and keep a working set of PMs in main memory. The cluster hierarchy without the PMs is typically a few megabytes for our benchmark models (e.g., 5MB for the St. Matthew model). We perform coarse-grained refinement at the cluster level and fine-grained refinement at the level of PMs. We introduce a frame of latency in the rendering pipeline in order to fetch the PMs of newly visible clusters from the disk and avoid stalls in the rendering pipeline.

5.1 Simplification Error Bounds

A key issue of view-dependent refinement is computation of errors associated with the LODs generated at runtime. The allowable runtime error is expressed in screen-space as a *pixels-of-error* (POE) value. Using the POE value and the minimum distance between a cluster and the viewpoint, we compute the maximum object-space error that is allowed for the cluster. We call this value the *error bound*. We use the *error bound* for a cluster to refine both clusters and PMs. This approach allows us to efficiently perform view-dependent computations using a single object-space comparison between the *error bound* and a stored error value in the clusters and PMs.

5.2 View-Dependent Refinement

Our algorithm maintains an active cluster list (ACL), which is a cut of clusters in the hierarchy representing the scene. During each frame, we refine the ACL based on the current viewing parameters. Specifically, we traverse the ACL and compute the *error bound* for each cluster. Each cluster on the active front whose *error bound* is less than the *min bound* of its *error range* is split because the PM cannot meet the *error bound*. Similarly, sibling clusters that have a greater *error bound* than *max error* are collapsed. Each PM in the ACL is refined prior to being rendered by choosing the mesh in the PM mesh sequence with the lowest face count that meets the *error bound*.

To accelerate the view-dependent refinement, we take advantage of temporal coherence between successive frames. We start with the position within the PM from the previous frame and perform the aforementioned view-dependent computation.

5.3 Handling Cluster Dependencies

Our simplification algorithm introduces dependencies between the clusters so that we can simplify their boundaries during the preprocess. We use these dependencies to generate a crack-free simplification at runtime.

Handling cluster dependencies is very similar to an approach of handling vertex dependencies. Cluster-collapses occur to reduce the polygon count in the current refinement. However, prior to collapsing a pair of sibling clusters, we must check the parent's dependencies. If the children of any dependent clusters cannot also be collapsed, then the initial cluster collapse cannot occur. Cluster-splits also occur to increase the polygon count to meet the error bound. If there are dependencies on a parent cluster, we force the cluster-splits of all the dependent clusters as well as the parent cluster. These checks occur at the cluster level and are inexpensive.

5.4 Conservative Occlusion Culling

We exploit temporal coherence in occlusion culling. Each frame our algorithm computes a potentially visible set of clusters (PVS) and a newly visible set (NVS), which is a subset of the PVS. The PVS for frame i is denoted as PVS_i and the NVS as NVS_i. An occlusion representation (OR_i), represented as a depth buffer, is computed by rendering PVS_{i-1} as an occluder set. Using OR_i, we determine PVS_i. The overall rendering algorithm is:

Step 1: Refine ACL. The ACL is refined as described in Section 5.2 based on the camera parameters for frame i.

Step 2: Render PVS_{i-1} to compute OR_i. We refine clusters in PVS_{i-1} based on the viewpoint, compute a simplification for each cluster, and render them to compute OR_i. OR_i is represented as a depth map that is used for occlusion culling. These clusters are rendered to both the depth and color buffers.

Step 3: Compute NVS_i and PVS_i. The bounding boxes of all the clusters in the ACL are tested for occlusion against

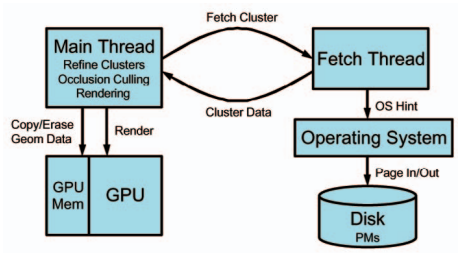


Fig. 7. Overall data flow. Quick-VDR uses two threads. The fetch thread manages the out-of-core PMs through interaction with the operating system. The main thread handles refinement, occlusion culling, and rendering. Geometry data for a working set of PMs is stored in GPU memory.

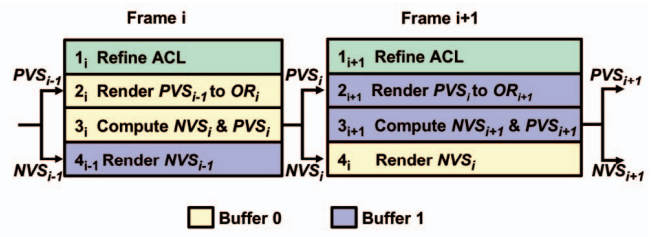


Fig. 8. Our rendering pipeline. In frame i, occlusion culling is performed for frame i, but the final image for frame $i-1$ is displayed. This allows extra time for loading the PMs of newly visible clusters. Two off-screen buffers facilitate this interleaving of successive frames. The partial rendering of frame i is stored in one buffer while occlusion culling for frame $i+1$ occurs in the other buffer.

OR_i. This test is performed with hardware occlusion queries at the resolution of image precision. The depth and color writes are disabled during this step to prevent overwriting of the depth and color values from Step 2. PVS_i contains all the clusters with visible bounding boxes, while NVS_i contains the clusters with visible bounding boxes that were not in PVS_{i-1}.

Step 4: Render NVS_i. The PMs of clusters in NVS_i are refined and rendered, generating the final image for frame i.

5.5 Out-of-Core Rendering

Our algorithm works with a fixed memory footprint of main memory and graphics card memory. The entire cluster hierarchy is in main memory and we fetch the PMs of the clusters needed for the current frame as well as prefetch some PMs of clusters for subsequent frames. Additionally, we store the vertices and faces of active clusters in GPU memory. By rendering the primitives directly from GPU memory, AGP bus bandwidth requirement is reduced and we obtain an increased triangle throughput.

5.5.1 Out-of-Core Framework

Our out-of-core rendering algorithm uses the paging mechanism in the operating system by mapping a file into read-only logical address space [22]. We choose the OS's virtual memory management because it can effectively optimize the disk access patterns and perform efficient memory management, which simplifies the design of our out-of-core algorithm. However, an application controlled paging mechanism can improve the performance of out-of-core memory management [49]. To fully take advantage of this mechanism, we store our view-dependent representation in a memory coherent manner, as described in Section 4.3. However, there is a limitation (e.g., 2GB in Windows XP) in 32-bit machine for mapping a file to user-accessible address space. We overcome this limitation by mapping only a 32MB portion of the file at a time and remapping when data is required from outside this range.

Our out-of-core rendering algorithm uses two separate threads: a main thread and a fetch thread. The rendering thread performs view-dependent refinement, occlusion culling, and rendering. The fetch thread is used to prepare data for PMs that are likely to be used in the future. This thread provides hints to OS and converts the PM data to the runtime format. The overall data flow is shown in Fig. 7.

5.5.2 LOD Prefetching

When we update clusters in the ACL by performing cluster-collapse and cluster-split operations, the children and parent clusters are activated. The PMs of these clusters may not be loaded in the main memory and this can stall the rendering pipeline. To prevent these stalls, whenever a cluster is added to the ACL, we prefetch its parent and children clusters. Thus, we attempt to keep one level of the hierarchy above and below the current ACL in memory.

5.5.3 Visibility Fetching

Predicting visibility or occlusion events is difficult, especially in complex models with high depth complexity and small holes. As a result, our algorithm introduces a frame of latency in the rendering pipeline and fetches the PMs of the newly visible cluster in the ACL from the disk.

In our rendering algorithm, visibility events are detected in Step 3 and the newly visible clusters are added to NVS_i (Section 5.4). These clusters are then rendered in Step 4, which will likely not allow enough time to load these clusters without stalling. Step 2, rendering OR_i, is the most time-consuming step of the rendering algorithm. Therefore, we delay the rendering of NVS_i until the end of Step 2 of the next frame and render PVS_{i-1} while fetching PMs from the disk in parallel (as shown in Fig. 7). Our rendering pipeline is reordered to include a frame of latency, thereby increasing the time allowed to load a cluster to avoid stall.

During frame i, we perform Steps 1 through 3 of the rendering algorithm with the camera parameters for frame i. However, we perform Step 4 for frame $i-1$ and generate the final image for frame $i-1$. The overall pipeline of the algorithm proceeds as: $1_i, 2_i, 3_i, 4_{i-1}, 1_{i+1}, 2_{i+1}, 3_{i+1}, 4_i, \ldots$, where n_j refers to Step n of frame j (as shown in Fig. 8).

In this reordered pipeline, the PM of a cluster in NVS_i will first have to be rendered during Step 2_{i+1} as this step renders PVS_i and clusters added to NVS_i are also added PVS_i in Step 3_i (refer to Fig. 8). During Step 2_{i+1}, we first render all the PMs that are already in memory. Since this is the most time-consuming step of the rendering algorithm, most of the PMs of the newly visible clusters are loaded during this time. As a result, we are able to balance the load between fetching PMs from the disk and rendering without stalls.

To implement this pipeline, we use a pair of off-screen buffers. One buffer holds the partial rendering of a frame from Step 2 so that it may be composited with the newly visible clusters in Step 4, the following frame. The odd numbered frames use the first buffer, while the

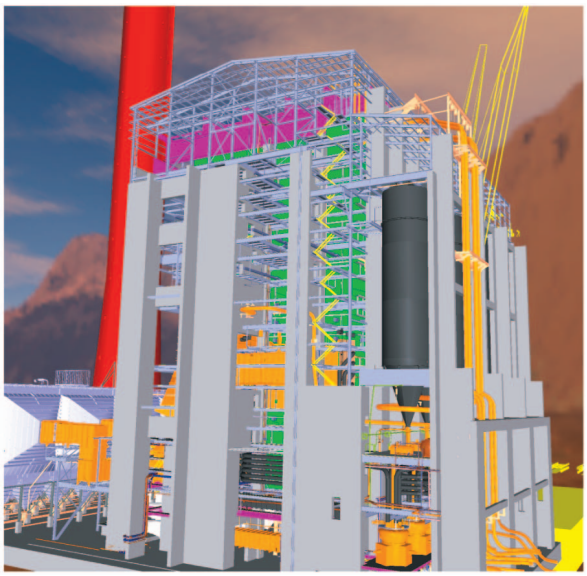


Fig. 9. Power plant. A rendering of the power plant model using our runtime algorithm. This model consists of over 12M triangles and has high depth complexity with small occluders. It is rendered at an average of 28 FPS using 400MB of main memory by our system.

TABLE 1
Preprocess Timings and Storage Requirements for Test Models

Model	PP	Isosurface	St. Matthew
Triangles (M)	12.2	100	372
Original Size (MB)	485	2,543	9,611
Num Clusters (K)	5.8	16	65
Memory footprint used (MB)	32	256	512
Size of CHPM (MB)	625	3,726	13,992
Size of cluster hierarchy (MB)	0.3	2.1	5.1
Processing time (min)	35	182	682

We are able to compute a CHPM for each environment using an out-of-core algorithm and a memory footprint of 256-512MB.

TABLE 2
Runtime Performance

Model	POE	Avg FPS	Avg Front Size	Avg # Ecol/Vsplit	Avg # Tri(K)
Power plant	1	26	2279	181	592
Isosurface	20	24	372	488	920
St. Matthew	1	17	434	2196	1121

We highlight the performance on the three benchmarks. The average frame rate, average font size, and average number of edge collapse and vertex splits are presented for a sample path in each model. All the data is acquired at 512×512 resolution. We use a 400MB memory footprint for the power plant model and 600MB for other models.

even-numbered frames use the second buffer so that each consecutive pair of frames can render to separate buffers. Fig. 8 illustrates how the buffers are used for two consecutive frames.

5.6 Utilizing GPUs

We achieve high throughput from graphics cards by storing the mesh data on the GPU, thereby reducing the data transferred from the CPU to the GPU during each frame. We use the GL_ARB_vertex_buffer_object OpenGL extension that performs GPU memory management for both the vertex and the face arrays. We use the half-edge collapse decimation operation so that the set of vertices used in the PMs is a subset of the vertices of the original model. However, we generate some new faces during each frame by performing vertex splits or edge collapse operations during local refinement of each PM. In practice, only a small number (i.e., 5 percent) of PMs require refinement during each frame. As a result, we only transmit the faces of these PMs to the GPU and the other faces are cached in the GPU memory. By utilizing the GL_ARB_vertex_buffer_object OpenGL extension, we have been able to achieve an average throughput of 23 million triangles on a PC with GeForce 5950FX Ultra GPU.

6 Implementation and Performance

In this section, we describe our implementation and highlight its performance on massive models.

6.1 Implementation

We have implemented our out-of-core simplification and runtime system on a dual 2.4 GHz Pentium-IV PC, with 1GB of RAM and a GeForce 5950FX Ultra GPU with 128MB of video memory. Our system runs on Windows XP and uses the operating system's virtual memory through memory mapped files.

We use the METIS graph partitioning library [47] for cluster computation. Since the METIS library does not guarantee that partitioned graphs are connected, we perform a postprocessing step to connect partitioned graphs if possible. We use NVIDIA OpenGL extension GL_NV_occulsion_query to perform occlusion queries. We are able to perform an average of approximately 400K occlusion queries per second on the bounding boxes. In practice, the ACL consists of hundreds of clusters and we are able to perform occlusion culling in 1-3 milliseconds per frame.

6.2 Massive Models

Our algorithm has been applied to three complex models, a coal-fired power plant composed of more than 12 million polygons and 1,200 objects (Fig. 9), the St. Matthew model consisting of a single 372 million polygon object (Fig. 2) and an isosurface model consisting of 100 million polygons, also originally represented as a single object (Fig. 1). The details of these models are shown in Table 1. We generated paths in each of our test models and used them to test the performance of our algorithm. These paths are shown in the accompanying video.

6.3 Performance

We have applied our out-of-core CHPM generation preprocess to each of the models. Table 1 presents preprocessing time for each model on the PC. We guarantee the maximum memory requirement is less than a user specified threshold (i.e., 256MB). We fully utilize the specified memory footprint during the cluster decomposition. However, other preprocessing steps including the cluster hierarchy generation and hierarchical simplification require a very small portion of the memory footprint. Hierarchical simplification takes approximately 85 percent of the preprocess time. The remainder of the time is dominated by the face pass of the cluster decomposition. This pass makes random accesses to the out-of-core vertex index mapping table to locate face vertices in the cluster

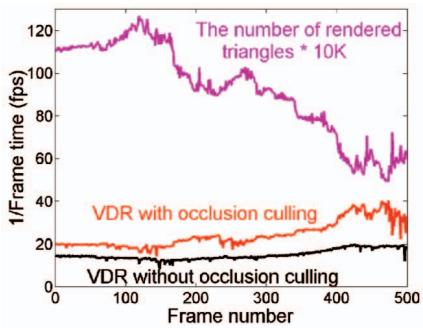


Fig. 10. Frame rate in isosurface model. Frame rates are shown for a sample path using our system. for comparison, we show our system without occlusion culling. The number of rendered triangles is also shown.

decomposition. We could use an external sort of the mapping table to improve access patterns, as in [14].

We are able to render all these models at interactive rates (10-35 frames per second) on a single PC. In Table 2, we report the runtime performance of our algorithm on the three models.

Fig. 10 illustrates the performance of the system on a complex path in isosurface model. Table 2 shows the average frame rate, front size, and number of edge collapse and vertex split operations performed for paths in each of our test models.

Table 3 shows the average breakdown of the frame time for each model. Rendering costs dominate the frame time.

6.3.1 Out-of-Core

Our system relies on the underlying operating systems virtual memory management for paging of PMs and, as discussed in Section 5.5.3, uses a frame of latency to hide load times of newly visible clusters. The frame rates of a sample path of the isosurface model are shown in Fig. 10. Please note that there is no severe stalling that would cause a large downward spikes in the frame rate. We achieve an average frame rate of 24 frames per second.

6.3.2 Occlusion Culling

Occlusion culling is very important for rendering models with high depth complexity, such as the power plant and isosurface models. In Fig. 10, highlights the benefit of occlusion culling by comparing the frame rate of our system over a path with occlusion culling enabled and disabled. On average, the frame rate is 25-55 percent higher when occlusion culling enabled. However, we achieve the improvement by spending a very small portion (e.g., less than 3 percent) of the frame time on occlusion culling (see Table 3). This suggests that we can further improve the overall runtime performance by reducing the granularity of occlusion culling. One possible choice is to use *subobjects* in each cluster for occlusion culling.

7 Analysis and Limitations

In this section, we analyze the performance of Quick-VDR. We also highlight the benefits over prior approaches and describe some of its limitations.

Refinement Cost of CHPMs versus Vertex Hierarchies. Most of the earlier algorithms for view-dependent simplification use a vertex hierarchy. These algorithms compute an active vertex front in the hierarchy and handle dependencies at the vertex or edge level.

TABLE 3
Runtime Timing Breakdown

Model	Refining	Occlusion Culling	Rendering	Stalling
Power plant	1.8%	13.9%	83.3%	1.0%
Isosurface	2.2%	6.6%	90.1%	1.1%
St. Matthew	4.1%	1.4%	93.8%	0.7%

This table shows the percentage of frame time spent on the four major computations of the runtime algorithm. More than 80 percent of the time is spent in rendering the potential occluders and visible primitives. The overhead of performing occlusion queries, refinement, and stalling is relatively small.

TABLE 4
Refinement Performance

Method	Vertex Hierarchy	CHPM
Num Dependency Checks	4.2M	223
Refinement Time(ms)	1,221	32

A comparison of the refinement cost between a CHPM and vertex hierarchy in a 1M triangle isosurface. This table measures the time to fully refine the mesh from the base mesh.

We compared the refinement cost of CHPM with an implementation of a vertex hierarchy (VDPM) for an isosurface with about 1M triangles (see Table 4). We have observed that the CHPM refinement cost is one-two orders of magnitude lower, even without occlusion culling. This lowered cost is due to the following factors:

1. Our clusters consist of thousands of triangles. As a result, the size of the ACL is typically more than one-two orders of magnitude smaller than the size of the active front in a vertex hierarchy.
2. We perform coarse-grained refinement operations at the cluster level and use a single conservative error bound for an entire cluster. Therefore, refinement of individual PMs is much faster than it would be by performing per-vertex computations across an active vertex front.
3. Handling dependencies at the cluster level is significantly cheaper than those at the vertex level.

Furthermore, occlusion culling helps us in further reducing the refinement cost as we do not need to refine PMs of the clusters that are not visible.

Conservative Occlusion Culling. Quick-VDR performs conservative occlusion culling up to image precision. The occlusion computations are performed at the cluster level. The size of the ACL is typically a few hundred clusters, so performing occlusion culling takes 1-10 percent of the total frame time.

Storage Overhead. Our CHPM implementation requires, on average, 88MB per million vertices. This is low compared to Hoppe's [1] VDPM representation (224MB) and XFastMesh (108MB) [20]. Moreover, CHPM can easily represent models with nonmanifold topologies. According to Hoppe [17], the compression ratio of PMs decreases as the size of the model increases. In a CHPM, the size of each PM is independent of the total mesh size. Furthermore, we

can use the relationship between the PMs of a parent cluster with its children cluster to achieve higher compression.

Out-of-Core Computation. Our out-of-core preprocess is able to construct a CHPM from large data sets using a constant-sized memory footprint. Moreover, our hierarchical simplification algorithm produces nearly in-core quality progressive meshes and preserves the mesh connectivity.

Our current implementation does not achieve the same performance of Lindstrom [21] in terms of triangles simplified per second. Lindstrom [21] applies external-memory sorts to his out-of-core data structures to improve the access patterns and we can also use them to improve the performance of our system. However, Lindstrom [21] does not preserve all the faces and vertices in the leaf level of the hierarchy.

Quick-VDR introduces a frame of latency to fetch PMs of the newly visible cluster from the disk. This is needed to take into account the visibility events that can occur between successive frames. Earlier algorithms that combine visibility computations with out-of-core rendering decompose large CAD environments into rectangular cells and do not introduce additional latency [33], [50]. However, it may not be easy to decompose large isosurfaces for visibility-based prefetching. Moreover, the MMR system [33] uses image-based impostors and can introduce additional popping artifacts.

7.1 Comparisons

In this section, we compare the performance of our algorithm with prior approaches. Our algorithm has been applied to complex models composed of a few hundred million polygons. In contrast, view-dependent algorithms were applied to scanned models with 8-10 million triangles [20], 2M triangle isosurface, and the power plant model [38] or were combined with approximate occlusion culling [13]. Lindstrom's algorithm [21] does not perform occlusion culling and has been applied it to a 47M triangle isosurface. It is difficult to perform direct comparisons with these approaches as they used an older generation of hardware and it may not have the same set of features (e.g., occlusion queries). Yoon et al. [38] also used clusters for occlusion culling. However, their underlying representation is a vertex hierarchy and their algorithm does not scale to massive models. Quick-VDR is able to achieve significant speedups over the technique described in [38]. The main reasons for the high frame-rate performance of Quick-VDR on massive models are:

- low refinement cost during each frame,
- high GPU throughput obtained by rendering PMs directly from GPU memory,
- significant occlusion culling based on the cluster hierarchy, and
- out-of-core computations at the cluster level.

7.1.1 Adaptive TetraPuzzles

Recently, the Adaptive TetraPuzzles approach [23] has been proposed for rendering models consisting of hundreds of millions of triangles. Adaptive TetraPuzzles uses a precomputed regular hierarchy of tetrahedra to spatially partition the model. A set of tetrahedrons sharing a longest edge form a "diamond" and are subdivided together. Each tetrahedron contains static LODs, which are stored as indexed triangle strips for faster rendering. On the other hand, CHPM employs progressive meshes in a cluster hierarchy to further provide smooth local refinement and reduce the number of triangles in each cluster. As a result, it is hard to compare the rendering quality of the images generated by these algorithms. Our implementation does not use triangle strips or index reordering to exploit the GPU's vertex cache. As a result, the rendering performance and throughput of CHPMs is lower than that of TetraPuzzles.

To simplify boundary triangles for high-quality simplification and faster rendering performance at runtime, we introduce explicit dependencies between arbitrary clusters. On the other hand, in a regular hierarchy of tetrahedra, boundary triangles alternate between different levels of the hierarchy based on diamonds. This can also be thought of as posing *implicit cluster dependencies*. The TetraPuzzles approach has been applied to scanned models, which have a uniform distribution of geometry. Therefore, it is not clear whether the regular hierarchy of tetrahedra with implicit dependencies works with a nonuniform distribution of geometry. Moreover, the effectiveness of occlusion culling with the tetrahedron hierarchy has not been evaluated.

7.1.2 iWalk System

The iWalk system [40], [41] has been proposed for interactive rendering of large models. iWalk can support high-resolution $(4,096 \times 3,072)$ and multitiled displays by employing a sort-first parallel out-of-core rendering. iWalk also uses static LODs and an octree for conservative occlusion culling. At runtime, iWalk predicts visibility events based on visibility coefficients stored in the octree nodes. iWalk was applied to the power plant model and an isosurface model consisting of 473 million triangles. Because the iWalk system has been tested in clusters of low-end commodity PCs, it is difficult to directly compare performances with Quick-VDR. iWalk achieves 10 frames per second on the power plant model and eight frames per second on average for the isosurface model with eight rendering servers.

7.2 Limitations

The main limitation of our approach is one frame of latency in the rendering pipeline. Other limitations include:

Drawbacks of CHPM. The set of possible dynamic simplifications represented by a CHPM is less than that of a full vertex hierarchy. This is caused by decomposing the model into clusters and representing each cluster as a linear sequence of edge collapses. As a result, a single *error bound* value must be used for an entire cluster. Thus, a given screen space error may be met with a slightly higher triangle count by a CHPM than a vertex hierarchy.

Dependencies. Cluster dependencies that force us to perform additional cluster-split operations might cause popping artifacts. These popping artifacts are always from lower to higher LOD, which may be preferable to pops that decrease LOD.

Occlusion culling and coherence. Our occlusion culling algorithm assumes high temporal coherence between successive frames. Its effectiveness varies as a function of coherence between successive frames. Furthermore, if a scene has very little or no occlusion, the additional overhead of performing occlusion queries could lower the frame rate.

Cluster decomposition and hierarchy. The performance of our preprocessing and runtime algorithm depends on good cluster decomposition and hierarchy generation. We reduce the problem to graph partitioning algorithms and the current algorithms for partitioning cannot guarantee good decomposition for every input.

8 Conclusion and Future Work

We have presented Quick-VDR, a novel algorithm for view-dependent rendering of massive models. Quick-VDR represents the scene using a CHPM and provides us with the ability to perform coarse-grained as well as fine-grained refinement. It significantly reduces the refinement cost as compared to earlier approaches based on vertex hierarchies. The cluster hierarchy enables occlusion culling and out-of-core rendering. Quick-VDR relies on an out-of-core algorithm to compute a CHPM and combines view-dependent simplification, occlusion culling, and out-of-core rendering. Quick-VDR has been applied to massive models with a few hundred million triangles and can render them at interactive rates using commodity graphics systems.

Many avenues for future work lie ahead. In addition to overcoming the limitations of the current approach, we would like to use geomorphing [5] to smooth the popping artifacts. Moreover, the two-level refinement could be extended to consider view-dependent effects such as specular highlights, silhouettes, and shadows. Also, we would like to improve the rendering performance of progressive meshes. We would also like to explore methods for predicting occlusion events so that we can further improve our out-of-core computation and eliminate the frame of latency in the rendering pipeline. Finally, we would like to use the CHPM representation for multi-resolution compression and collision detection between massive models [51].

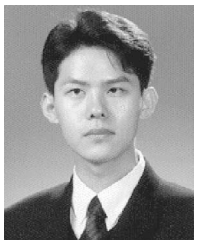
Acknowledgments

This work was supported in part by US Army Research Office Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, US National Science Foundation awards 0400134 and 0118743, US Office of Naval Research Contracts N00014-01-1-0067 and N00014-01-1-0496, US Defense Advanced Research Projects Agency Contract N61339-04-C-0043, and Intel. The St. Matthew statue images are courtesy of the Digital Michelangelo Project at Stanford University. The isosurface model is courtesy of the LLNL ASCI VIEWS Visualization project. The power plant environment is courtesy of an anonymous donor. The authors would like to thank Naga Govindaraju, Martin Isenburg, Stefan Gumhold, Mark Duchaineau, Peter Lindstrom, and the members of the UNC Walkthrough group for their useful discussions and support. Finally, they would like to thank the anonymous reviewers for their constructive suggestions.

References

[1] H. Hoppe, "View Dependent Refinement of Progressive Meshes," *ACM SIGGRAPH Conf. Proc.*, pp. 189-198, 1997.
[2] D. Luebke and C. Erikson, "View-Dependent Simplification of Arbitrary Polygon Environments," *Proc. ACM SIGGRAPH*, 1997.
[3] J. Xia, J. El-Sana, and A. Varshney, "Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 2, pp. 171-183, Apr.-June 1997.
[4] S. Yoon, B. Salomon, R. Gayle, and D. Manocha, "Quick-VDR: Interactive View-Dependent Rendering of Massive Models," *Proc. IEEE Visualization*, 2004.
[5] H. Hoppe, "Progressive Meshes," *Proc. ACM SIGGRAPH*, pp. 99-108, 1996.
[6] E.P.L. De Floriani and P. Magillo, "Building and Traversing a Surface at Variable Resolution," *Proc. IEEE Visualization*, 1997.
[7] N. Williams, D. Luebke, J.D. Cohen, M. Kelley, and B. Schubert, "Perceptually Guided Simplification of Lit, Textured Meshes," *Proc. ACM Symp. Interactive 3D Graphics*, 2003.
[8] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein, "ROAMing Terrain: Real-Time Optimally Adapting Meshes," *Proc. IEEE Visualization*, pp. 81-88, 1997.
[9] J. El-Sana and A. Varshney, "Generalized View-Dependent Simplification," *Computer Graphics Forum*, pp. C83-C94, 1999.
[10] J. El-Sana, E. Azanli, and A. Varshney, "Skip Strips: Maintaining Triangle Strips for View-Dependent Rendering," *Proc. IEEE Visualization*, 1999.
[11] R. Pajarola, "Fastmesh: Efficient View-Dependent Mesh," *Proc. Pacific Graphics*, pp. 22-30, 2001.
[12] A. Bogomjakov and C. Gotsman, "Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes," *Computer Graphics Forum*, 2002.
[13] J. El-Sana and E. Bachmat, "Optimized View-Dependent Rendering for Large Polygonal Data Set," *Proc. IEEE Visualization*, pp. 77-84, 2002.
[14] P. Lindstrom and C. Silva, "A Memory Insensitive Technique for Large Model Simplification," *Proc. IEEE Visualization*, pp. 121-126, 2001.
[15] E. Shaffer and M. Garland, "Effient Adaptive Simplification of Massive Meshes," *Proc. IEEE Visualization*, 2001.
[16] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno, "External Memory Management and Simplification of Huge Meshes," *IEEE Trans. Visualization and Computer Graphics*, 2003.
[17] H. Hoppe, "Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering," *IEEE Visualization Conf. Proc.*, pp. 35-42, 1998.
[18] C. Prince, "Progressive Meshes for Large Models of Arbitrary Topology," master's thesis, Univ. of Washington, 2000.
[19] J. El-Sana and Y.-J. Chiang, "External Memory View-Dependent Simplification," *Computer Graphics Forum*, vol. 19, no. 3, pp. 139-150, 2000.
[20] C. DeCoro and R. Pajarola, "Xfastmesh: View-Dependent Meshing from External Memory," *Proc. IEEE Visualization*, 2002.
[21] P. Lindstrom, "Out-of-Core Construction and Visualization of Multiresolution Surfaces," *Proc. ACM Symp. Interactive 3D Graphics*, 2003.
[22] P. Lindstrom and V. Pascucci, "Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization," *IEEE Trans. Visualization and Computer Graphics*, vol. 8, pp. 239-254, 2002.
[23] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Adaptive TetraPuzzles—Efficient Out-of-Core Construction and Visualization of Gigantic Polygonal Models," *Computer Graphics Forum*, vol. 23, no. 3, Aug. 2004.
[24] D. Cohen-Or, Y. Chrysanthou, C. Silva, and F. Durand, "A Survey of Visibility for Walkthrough Applications," *IEEE Trans. Visualization and Computer Graphics*, vol. 9, 2003.
[25] S. Coorg and S. Teller, "Real-Time Occlusion Culling for Models with Large Occluders," *Proc. ACM Symp. Interactive 3D Graphics*, 1997.
[26] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang, "Accelerated Occlusion Culling Using Shadow Frusta," *Proc. ACM Symp. Computational Geometry*, pp. 1-10, 1997.
[27] N. Greene, M. Kass, and G. Miller, "Hierarchical z-Buffer Visibility," *Proc. ACM SIGGRAPH*, pp. 231-238, 1993.
[28] H. Zhang, D. Manocha, T. Hudson, and K. Hoff, "Visibility Culling Using Hierarchical Occlusion Maps," *Proc. ACM SIGGRAPH*, 1997.

[29] J. Klosowski and C. Silva, "Efficient Conservative Visiblity Culling Using the Prioritized-Layered Projection Algorithm," *IEEE Trans. Visualization and Computer Graphics,* vol. 7, no. 4, pp. 365-379, Oct.-Dec. 2001.
[30] P. Wonka, M. Wimmer, and F. Sillion, "Instant Visibility," *Proc. Eurographics,* 2001.
[31] N. Govindaraju, A. Sud, S. Yoon, and D. Manocha, "Interactive Visibility Culling in Complex Environments with Occlusion-Switches," *Proc. ACM Symp. Interactive 3D Graphics,* pp. 103-112, 2003.
[32] T. Funkhouser, D. Khorramabadi, C. Sequin, and S. Teller, "The UCB System for Interactive Visualization of Large Architectural Models," *Presence,* vol. 5, no. 1, pp. 13-44, 1996.
[33] 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," *Proc. ACM Symp. Interactive 3D Graphics,* pp. 199-206, 1999.
[34] B. Baxter, A. Sud, N. Govindaraju, and D. Manocha, "GigaWalk: Interactive Walkthrough of Complex 3D Environments," *Proc. Eurographics Workshop Rendering,* pp. 203-214, 2002.
[35] N. Govindaraju, B. Lloyd, S. Yoon, A. Sud, and D. Manocha, "Interactive Shadow Generation in Complex Environments," *Proc. ACM SIGGRAPH/ACM Trans. Graphics,* vol. 22, no. 3, pp. 501-510, 2003.
[36] C. Erikson, D. Manocha, and B. Baxter, "Hlods for Fast Display of Large Static and Dynamic Environments," *Proc. ACM Symp. Interactive 3D Graphics,* 2001.
[37] I. Wald, A. Dietrich, and P. Slusallek, "An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models," *Proc. Eurographics Symp. Rendering,* 2004.
[38] S. Yoon, B. Salomon, and D. Manocha, "Interactive View-Dependent Rendering with Conservative Occlusion Culling in Complex Environments," *Proc. IEEE Visualization,* 2003.
[39] J. El-Sana, N. Sokolovsky, and C. Silva, "Integrating Occlusion Culling with View-Dependent Rendering," *Proc. IEEE Visualization,* 2001.
[40] W.T. Correa, J.T. Klosowski, and C.T. Silva, "Visibility-Based Prefetching for Interactive Out-of-Core Rendering," *Proc. PVG 2003 (Sixth IEEE Symp. Parallel and Large-Data Visualization and Graphics),* pp. 1-8, 2003.
[41] W.T. Correa, "New Techniques for Out-of-Core Visualization of Large Datasets," PhD dissertation, Princeton Univ., 2004.
[42] S. Rusinkiewicz and M. Levoy, "Qsplat: A Multiresolution Point Rendering System for Large Meshes," *Proc. ACM SIGGRAPH,* 2000.
[43] C. Dachsbacher, C. Vogelgsang, and M. Stamminger, "Sequential Point Trees," *Proc. ACM SIGGRAPH,* 2003.
[44] M. Isenburg and S. Gumhold, "Out-of-Core Compression for Gigantic Polygon Meshes," *ACM Trans. Graphics (Proc. ACM SIGGRAPH),* vol. 22, 2003.
[45] S. Arya and D.M. Mount, "Approximate Nearest Neighbor Queries in Fixed Dimensions," *Proc. Fourth ACM-SIAM Symp. Discrete Algorithms,* pp. 271-280, 1993.
[46] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs," *Proc. Super Computing,* 1995.
[47] G. Karypis and V. Kumar, "Multilevel k-Way Partitioning Scheme for Irregular Graphs," *J. Parallel and Distributed Computing,* 1998.
[48] M. Garland and P. Heckbert, "Surface Simplification Using Quadric Error Bounds," *Proc. ACM SIGGRAPH,* pp. 209-216, 1997.
[49] M. Cox and D. Ellsworth, "Application-Controlled Demand Paging for Out-of-Core Visualization," *Proc. IEEE Visualization,* pp. 235-244, 1997.
[50] W. Correa, J. Klosowski, and C. Silva, "iwalk: Interactive Out-of-Core Rendering of Large Models," Technical Report TR-653-02, Princeton Univ., 2002.
[51] S. Yoon, B. Salomon, M.C. Lin, and D. Manocha, "Fast Collision Detection between Massive Models Using Dynamic Simplification," *Proc. Eurographics Symp. Geometry Processing,* 2004.



Sung-Eui Yoon is a PhD student in the Department of Computer Science at the University of North Carolina at Chapel Hill. He received the BS degree in computer science from Seoul National University and the MS degree in computer science from Seoul National University and the University of North Carolina at Chapel Hill. His research interests include real-time rendering, collision detection, and cache-coherent algorithms and data structures.

Brian Salomon is a PhD student in the Department of Computer Science at the University of North Carolina at Chapel Hill. In 2001, he received the BS degree in computer science from the University of Virginia. His reasearch interests include real-time rendering, path planning, and visibility computations.

Russell Gayle is currently a PhD student in the Department of Computer Science at the University of North Carolina at Chapel Hill. He received BS degrees in computer science and in mathematics from the University of Texas at Austin. He has also been awarded a US Department of Energy High-Performance Computer Science Fellowship. He has research interests in real-time rendering, simulation, motion planning, and visualization.

Dinesh Manocha received the BTech degree in computer science and engineering from the Indian Institute of Technology, Delhi, in 1987 and the MS and PhD degrees in computer science from the University of California at Berkeley in 1990 and 1992, respectively. He is currently a professor of computer science at the University of North Carolina (UNC) at Chapel Hill. He has published more than 130 papers in leading conferences and journals in computer graphics, geometric modeling, computational geometry, robotics, and symbolic computation. He was selected an Alfred P. Sloan Research Fellow, received a US National Science Foundation Career Award in 1995 and US Office of Naval Research Young Investigator Award in 1996, Honda Research Initiation Award in 1997, and Hettleman Prize for scholarly achievement at UNC Chapel Hill in 1998. He has also received best paper awards at the ACM SuperComputing, ACM Multimedia, and Eurographics Conferences. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Cache-Oblivious Mesh Layouts

ung ui oon[1] Peter indstrom[2] alerio Pascucci[2] inesh Manocha[1]

[1]University of orth Carolina at Chapel Hill [2] awrence ivermore ational aboratory
http://gamma.cs.unc.edu/CO

Abstract

We present a novel method for computing cache-oblivious layouts of large meshes that improve the performance of interactive visualization and geometric processing algorithms. Given that the mesh is accessed in a reasonably coherent manner, we assume no particular data access patterns or cache parameters of the memory hierarchy involved in the computation. Furthermore, our formulation extends directly to computing layouts of multi-resolution and bounding volume hierarchies of large meshes.

We develop a simple and practical cache-oblivious metric for estimating cache misses. Computing a coherent mesh layout is reduced to a combinatorial optimization problem. We designed and implemented an out-of-core multilevel minimization algorithm and tested its performance on unstructured meshes composed of tens to hundreds of millions of triangles. Our layouts can significantly reduce the number of cache misses. We have observed $2-20$ times speedups in view-dependent rendering, collision detection, and isocontour extraction without any modification of the algorithms or runtime applications.

1 Introduction

Over the last few years, advances in model acquisition, computer-aided design, and simulation technologies have resulted in massive databases of complex geometric models. Meshes composed of tens or hundreds of millions of triangles are frequently used to represent CAD environments, terrains, isosurfaces, and scanned models.

Efficient algorithms for processing large meshes utilize the computational power of CPUs and GPUs for interactive visualization and geometric applications. A major computing trend over the last few decades has been the widening gap between processor speed and main memory speed. As a result, system architectures increasingly use caches and memory hierarchies to avoid memory latency. The access times of different levels of a memory hierarchy typically vary by orders of magnitude. In some cases, the running time of a program is as much a function of its cache access pattern and efficiency as it is of operation count [Frigo et al. 1999; Sen et al. 2002].

Our goal is to design cache efficient algorithms to process large meshes. The two standard techniques to reduce cache misses are:

1. **C mputati n Re rdering**: Reorder the computation to improve program locality. This is performed using compiler optimizations or application specific hand-tuning.
2. **Data Lay ut Optimizati n:** Compute a cache-coherent layout of the data in memory according to the access pattern.

In this paper, we focus on data layout optimization of large meshes to improve cache coherence. A triangle mesh is represented by linear sequences of vertices and triangles. Therefore, the problem becomes one of computing a cache efficient layout of the vertices and triangles.

Copyright © 2005 by the Association for Computing Machinery, Inc.
ACM acknowledges that this contribution was authored or co-authored by a contractor of affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
© 2005 ACM 0730-0301/05/0700-0886 $5.00

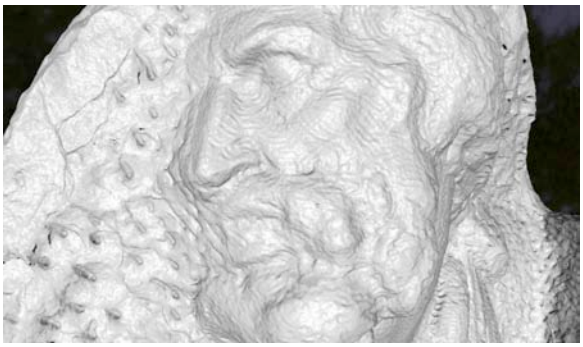


Figure 1: **Scan of Michelangelo's St. Matthew:** We precompute a cache-oblivious layout of this 9.6GB scanned model with 372M triangles. Our novel metric results in a cache-oblivious layout and at runtime reduces the vertex cache misses by more than a factor of four for interactive view-dependent rendering. As a result, we improve the frame rate by almost five times. We achieve a throughput of 106M tri/sec (at 82 fps) on an NVIDIA GeForce 6800 GPU.

Many layout algorithms and representations have been proposed for optimizing the cache access patterns for specific applications. The representations include *rendering sequences* (e.g. triangle strips) that are used to improve the rendering performance of large meshes on GPUs. Recent extensions include *processing sequences* (e.g. streaming meshes), which work well for applications that can access the data in a fixed order. Some algorithms for image processing and visualization of large datasets use space filling curves as a heuristic to improve cache coherence of a layout. These algorithms work well on models with a regular structure; however, they do not take into account the topological structure of a mesh and are not general enough to handle unstructured datasets.

Main Results: We present a novel method to compute cache-oblivious layouts of large triangle meshes. Our approach is general in terms of handling all kinds of polygonal models and cache-oblivious as it does not require any knowledge of the cache parameters or block sizes of the memory hierarchy involved in the computation.

We represent the mesh as an undirected graph $G = (V, E)$, where $|V| = n$ is the number of vertices. The mesh layout problem reduces to computing an optimal one-to-one mapping of vertices to positions in the layout, $\varphi : V \rightarrow \{1, ..., n\}$, that reduces the number of cache misses. Our specific contributions include:

1. Deriving a practical cache-oblivious metric that estimates the number of cache misses.
2. Transforming the layout computation to an optimization problem based on our metric.
3. Solving the combinatorial optimization problem using a multilevel minimization algorithm.

We also extend our graph-based formulation to compute cache-oblivious layouts of bounding volume and multiresolution hierarchies of large meshes.

We use cache-oblivious layouts for three applications: view-dependent rendering of massive models, collision detection between complex models, and isocontour extraction. In order to show the generality of our approach, we compute layouts of several kinds of geometric models including CAD environments, scanned models, isosurfaces, and terrains. We use these layouts directly without any modification to the runtime application. Our layouts significantly

reduce the number of cache misses and improve the overall performance. Compared to a variety of popular mesh layouts, we achieve on average:

1. Over an order of magnitude improvement in performance for isocontour extraction.
2. A five time improvement in rendering throughput for view-dependent rendering of multi-resolution meshes.
3. A two time speedup in collision detection queries based on bounding volume hierarchies.

Organization: The rest of the paper is organized as follows. We give a brief summary of related work on cache-efficient algorithms and mesh layouts in Section 2. Section 3 gives an overview of our approach and presents techniques for computing the graph layout of hierarchical representations. We present our cache-oblivious metric in Section 4 and describe the multilevel optimization algorithm for computing the layouts in Section 5. Section 6 highlights the use of our layouts in three different applications. We analyze our algorithms and discuss some of their limitations in Section 7.

2 Related Work

In this section we briefly review related work on cache-efficient algorithms, out-of-core techniques, mesh sequences, and layouts.

2.1 Cache-Efficient Algorithms

Cache-efficient algorithms have received considerable attention over last two decades in theoretical computer science and compiler literature. These algorithms include theoretical models of cache behavior [Vitter 2001; Sen et al. 2002], and compiler optimizations based on tiling, strip-mining, and loop interchanging; all of these can minimize cache misses [Coleman and McKinley 1995].

At a high level, cache-efficient algorithms can be classified as either cache-aware or cache-oblivious. Cache-aware algorithms utilize knowledge of cache parameters, such as cache block size [Vitter 2001]. On the other hand, cache-oblivious algorithms do not assume any knowledge of cache parameters [Frigo et al. 1999]. There is a considerable amount of literature on developing cache-efficient algorithms for specific problems and applications, including numerical programs, sorting, geometric computations, matrix multiplication, FFT, and graph algorithms. Most of these algorithms reorganize the data structures for the underlying application, i.e., computation reordering. More details are given in recent surveys [Arge et al. 2004; Vitter 2001]. There exists relatively little work on computing cache-coherent layouts for a wide variety of applications.

2.2 Out-of-Core Mesh Processing

Out-of-core algorithms are designed to handle massive geometric datasets on computers with finite memory. A recent survey of these algorithms and their applications is given in [Silva et al. 2002]. The survey includes techniques for efficient disk layouts that reduce the number of disk accesses and the time taken to load the data required at runtime. These algorithms have been used for model simplification [Cignoni et al. 2003], interactive display of large datasets composed of point primitives [Rusinkiewicz and Levoy 2000] or polygons [Corrêa et al. 2003; Yoon et al. 2004b], model compression [Isenburg and Gumhold 2003], and collision detection [Franquesa-Niubo and Brunet 2003; Wilson et al. 1999]. Out-of-core techniques are complimentary to cache-coherent mesh layouts.

2.3 Mesh Sequences and Layouts

The order in which a mesh is laid out can affect the performance of algorithms operating on the mesh. Several possibilities have been considered.

Rendering Sequences: Modern GPUs maintain a small buffer to reuse recently accessed vertices. In order to maximize the benefits of vertex buffers for fast rendering, triangle reordering is necessary. This approach was pioneered by Deering [1995]. The resulting ordering of triangles is called a triangle strip or a rendering sequence.

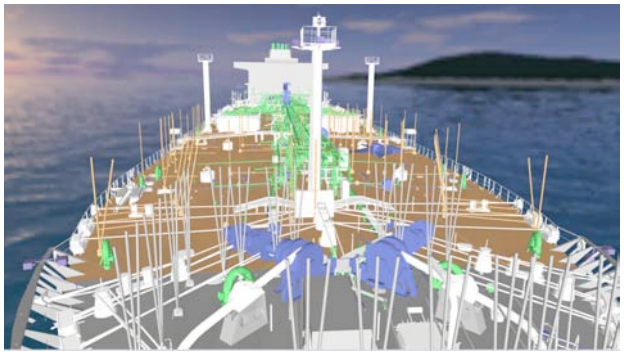


Figure 2: **Double Eagle Tanker:** We compute a cache-oblivious layout of the tanker with 82M triangles and more than 127K different objects. This model has an irregular distribution of primitives. We use our layout to reduce vertex cache misses and to improve the frame rate for interactive view-dependent rendering by a factor of two; we achieve a throughput of 47M tri/sec (at 35 fps) on an NVIDIA GeForce 6800 GPU.

Hoppe [1999] casts the triangle reordering as a discrete optimization problem with a cost function relying on a specific vertex buffer size. If a triangle mesh is computed on the fly using view-dependent simplification or other geometric operations, the rendering sequences need to be recomputed to maintain high throughput. Other techniques improve the rendering performance of view-dependent algorithms by computing rendering sequences not tailored to a particular cache size [Bogomjakov and Gotsman 2002; Karni et al. 2002]. However, these algorithms have been applied only to relatively small models (e.g., 100K triangles).

Processing Sequences: Isenburg and Gumhold [2003] propose processing sequences as an extension of rendering sequences to large-data processing. A processing sequence represents a mesh as an interleaved ordering of indexed triangles and vertices that can be streamed through main memory [Isenburg and Lindstrom 2004]. However, global mesh access is restricted to a fixed traversal order; only localized random access to the buffered part of the mesh is supported as it streams through memory. This representation is mostly useful for offline applications (e.g., simplification and compression) that can adapt their computations to the fixed ordering.

Space Filling Curves: Many algorithms use space filling curves [Sagan 1994] to compute cache-friendly layouts of volumetric grids or height fields. These layouts are widely used to improve performance of image processing [Velho and Gomes 1991] and terrain or volume visualization [Pascucci and Frank 2001; Lindstrom and Pascucci 2001]. A standard method of constructing a layout is to embed the meshes or geometric objects in a uniform structure that contains the space filling curve. Therefore, these algorithms have been used for objects or meshes with a regular structure (e.g. images and height fields). Methods based on space filling curves do not consider the topological structure of meshes. It is unclear whether these approaches would extend to large CAD environments with an irregular distribution of geometric primitives. Moreover, if an application needs to access the mesh primitives based on connectivity information, space filling curves may not be useful. Algorithms have also been proposed to compute paths on constrained, unstructured graphs as well as to generate triangle strips and finite-element mesh layouts [Heber et al. 2000; Oliker et al. 2002; Bartholdi and Goldsman 2004; Gopi and Eppstein 2004].

Sparse Matrix Reordering: There is considerable research on converting sparse matrices into banded ones to improve the performance of various matrix operations [Diaz et al. 2002]. Common graph and matrix reordering algorithms attempt to minimize one of three measures: bandwidth (maximum edge length), profile (sum of maximum per-vertex edge length), and wavefront (maximum front size, as in stream processing). These measures are closely connected with MLA and layouts for streaming, and generally are more applicable to stream layout than cache-oblivious mesh layout.

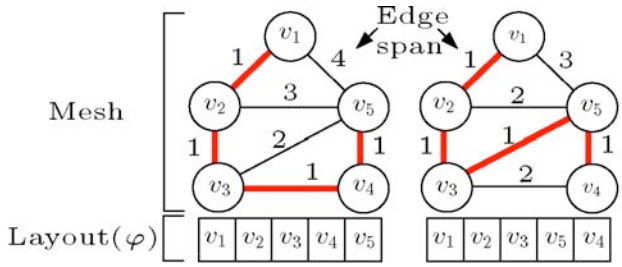


Figure 3: **Vertex layout for a mesh:** A mesh consisting of 5 vertices is shown with two different orderings obtained using a local permutation of v_4 and v_5. We highlight the span of each edge based on the layout. The ordering shown on the right minimizes cache misses according to our cache-oblivious metric.

3 Mesh Layout and Cache Misses

In this section, we introduce some of the terminology used in the rest of the paper and give an overview of memory hierarchies. We represent a mesh as a graph and extend our approach to layouts of multi-resolution and bounding volume hierarchies of a mesh.

3.1 Memory Hierarchy and Caches

Most modern computers use hierarchies of memory levels, where each level of memory serves as a *cache* for the next level. Memory hierarchies have two main characteristics. First, higher levels are larger in size and farther from the processor, and they have slower access times. Second, data is moved in large blocks between different memory levels. The mesh layout is initially stored in the highest memory level, typically the disk. The portion of the layout accessed by the application is transferred in large blocks into the next lower level, such as main memory. A transfer is performed whenever there is a cache miss between two adjacent levels of the memory hierarchy. The number of cache misses is dependent on the layout of the original mesh in memory and the access pattern of the application.

3.2 Mesh Layout

A mesh layout is a linear sequence of vertices and triangles of the mesh. We construct a graph in which each vertex represents a data element of the mesh. An edge exists between two vertices of the graph if their representative data elements are likely to be accessed in succession by an application at runtime.

For single-resolution mesh layout, we map mesh vertices and edges to graph vertices and edges. A vertex layout of an undirected graph $G = (V, E)$ is a one-to-one mapping of vertices to positions, $\varphi : V \rightarrow \{1, \ldots, n\}$, where $|V| = n$. Our goal is to find a mapping, φ, that minimizes the number of cache misses during accesses to the mesh.

A mesh layout is composed of two layouts: a vertex layout and a triangle layout. While a triangle layout can be constructed as a vertex layout of the dual graph, because the triangles and their vertices are often accessed together we ensure that the triangle layout is "compatible" with the vertex layout, e.g. by ordering triangles on their minimum or maximum vertex index (cf. [Isenburg and Lindstrom 2004]). In the rest of the paper, we use the term layout to refer to a vertex layout for the sake of clarity.

3.3 Layouts of Multiresolution Meshes and Hierarchies

In this section, we show that our graph-based formulation can be used to compute cache-coherent layouts of hierarchical representations. Hierarchical data structures are widely used to speed up computations on large meshes. Two types of hierarchies are used for geometric processing and interactive visualization: bounding volume hierarchies (BVHs) and multi-resolution hierarchies (MRHs). The BVHs use simple bounding shapes (e.g. spheres, AABBs, OBBs) to enclose a group of triangles in a hierarchical manner. MRHs are used to generate a simplification or approximation of the original model based on an error metric; these include vertex hierarchies (VHs) used for view-dependent rendering, and hierarchies that are defined using subdivision rules.

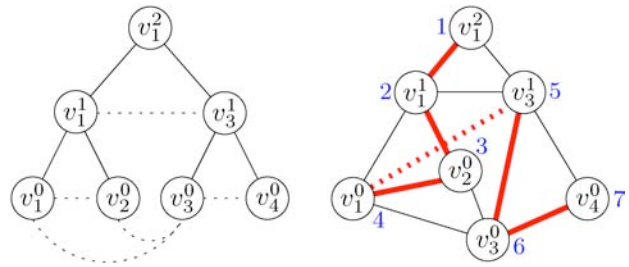


Figure 4: **A layout of a vertex hierarchy:** A vertex hierarchy is shown on the left. Each node of the vertex hierarchy represents a leaf or intermediate level vertex. A parent node, v_1^1, is constructed by merging two child nodes, v_1^0 and v_2^0. Solid lines between the nodes represent connectivity access and dotted lines represent the spatial locality between the nodes at the same level. Its corresponding graph and a layout of the vertices (with a position in the layout shown in blue) are shown on the right.

Terminology: We define $v_i = v_i^0$ as the ith vertex at the leaf level of the hierarchy, and v_i^k as a vertex at the kth level. v_i^k is a parent of v_i^{k-1} and v_{i+1}^{k-1}. In the case of a BVH, v_i^k denotes a bounding volume. In the case of a vertex hierarchy, v_i^k denotes a vertex generated by decimation operations. An example of a vertex hierarchy is shown in Fig. 4.

In order to compute a layout of a hierarchy, we construct a graph that captures cache-coherent access patterns to the hierarchy. We add extra edges to our graph that capture the spatial locality and parent-child relationships within the hierarchy.

1. **C nnectivity between parent-children n des:** Once a node of a hierarchy is accessed, it is highly likely that its parent or child nodes would be accessed soon. For example, a vertex-split of a node in the VH activates its child nodes and an edge-collapse of two sibling nodes activates their parent node.
2. **Spatial l cality between vertices at the same level:** Whenever a node is accessed, other nodes in close proximity are also highly likely to be accessed thereafter. For example, collisions or contacts between two objects occur in small localized regions of a mesh. Therefore, if a node of a BVH is activated, other nearby nodes are either colliding or are in close proximity and may be accessed soon.

Graph Representation: We take these localities into account and compute an undirected graph for MRHs and BVHs. For a BVH, we represent each BV with a separate vertex in the graph. The edges in our graph include edges between parent vertices and their children, and edges between nearby vertices at each level of the BVH. Edges are created between nearby vertices when their Euclidean distance falls below a given threshold. Fig. 4 shows the graph as well as its layout for the given vertex hierarchy. More details on connectivity and spatial localities of BVHs are also available [Yoon and Manocha 2005].

4 Cache-Oblivious Layouts

In this section we present a novel algorithm for computing a cache-coherent layout of a mesh. We make no assumptions about cache parameters and compute the layout in a cache-oblivious manner.

4.1 Terminology

We use the following terminology in the rest of the paper. The *edge span* of the edge between v_i and v_j in a layout is the absolute difference of the vertex indices, $|i - j|$ (see Fig. 3). We use E_l to denote the set that consists of all the edges of edge span l, where $l \in [1, n-1]$. The *edge span distribution* of a layout is the histogram of spans of all the edges in the layout. The *cache miss ratio* is the ratio of the number of cache misses to the number of accesses. The *cache miss ratio function (M F)*, p_l, is a function that relates the cache miss ratio to an edge span, l. The CMRF always lies within the interval $[0, 1]$; it is exactly 0 when there are no cache misses, and equals 1 when every access results in a cache miss. We alter the

Figure 5: **Puget Sound contour line:** This image shows a contour line (in black) extracted from an unstructured terrain model of the Puget Sound. The terrain is simplified down to 143M triangles. We extracted the largest component (223K edges) of the level set at 500 meters of elevation. Our cache-oblivious layouts improve the performance of the isocontour extraction algorithm by more than an order of magnitude.

layouts using a *local permutation* that reorders a small subset of the vertices. The local permutation changes the edge span of edges that are incident to the affected vertices (see Fig. 3).

4.2 Metrics for Cache Misses

We first define a metric for estimating the cache misses for a given layout. One well known metric for the graph layout problem is the minimum linear arrangement (MLA), which minimizes the sum of edge spans [Diaz et al. 2002]. Heuristics for the NP-hard MLA problem, such as spectral sequencing, have been used to compute mesh layouts for rendering and processing sequences [Bogomjakov and Gotsman 2002; Isenburg and Lindstrom 2004]. We have empirically observed that metrics used to estimate MLA may not minimize cache misses for general applications (See Fig. 6). This is mostly because MLA results in a front-advancing sweep over the mesh along a dominant direction that tends to minimize the length of the front. On a rectilinear grid, for example, MLA roughly corresponds to a row-by-row layout, which has poor worst-case performance when accessing the grid column by column. We present an alternate metric based on the edge span distribution and the CMRF that captures the locality for various access patterns and results in layouts with a more "space filling" quality. Contrary to MLA, our layouts are not biased towards a particular traversal direction.

Cache-coherent Access Pattern: If we know the runtime access pattern of a given application a priori and the CMRFs, we can compute the exact number of cache misses. However, we make no assumptions about the application and instead use a probabilistic model to estimate the number of cache misses. Our model approximates the edge span distribution of the runtime access pattern of the vertices with the edge span distribution of the layout. Based on this model, we define the expected number of cache misses of the layout as:

$$ECM = \sum_{i=1}^{n-1} |E_i| p_i \tag{1}$$

where $|E_i|$ is the cardinality of E_i and is a function of the layout, φ.

4.3 Assumptions

Our goal is to compute a layout, φ, that minimizes the expected number of cache misses for all possible cache parameters. We present a metric that is used to check whether a local permutation would reduce cache misses. We make two assumptions with respect to CMRFs: invariance and monotonicity.

Invariance: We assume that the CMRF of a layout is invariant before and after a local permutation. Since a local permutation affects only a small region of a mesh, the changes in CMRF due to a local permutation are very small.

Monotonicity: We assume that the CMRF is a monotonically non-decreasing function of edge span. As we access vertices that are farther away from the current vertex (i.e. the edge spans increase), the probability of having a cache miss increases, until eventually leveling off at 1.

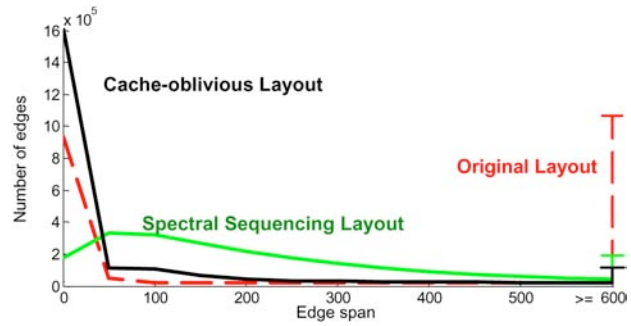


Figure 6: **Edge span distributions:** The edge span histogram of the dragon model with 871K triangles and 437K vertices. We show the histogram of the original model representation (red), spectral sequencing (green), and our cache-oblivious metric (black). In the original layout, a large number of edges have edge spans greater than 600. Intuitively, our cache-oblivious metric favors edges that have small edge spans. Therefore, our layouts reduce cache misses.

4.4 Cache-oblivious Metric

Our cache-oblivious metric is used to decide whether a local permutation decreases the expected number of cache misses, which due to the invariance of p_i is true if the following inequality holds:

$$\sum_{i=1}^{n-1} (|E_i| + \Delta|E_i|) p_i < \sum_{i=1}^{n-1} |E_i| p_i \Leftrightarrow \sum_{j=1}^{m} \Delta|E_{l(j)}| p_{l(j)} < 0 \tag{2}$$

Here $\Delta|E_i|$ is the signed change in the number of edges with edge span i after a local permutation and $n-1$ is maximum edge span for a mesh with n vertices. Furthermore, we let m denote the number of sets (among $E_1, E_2, \ldots, E_{n-1}$) whose cardinality changes because of the permutation, and let $l(j)$ denote the edge span associated with the jth such set, with $l(j) < l(j+1)$ and $m \ll n-1$.

Constant Edge Property: The total number of edges in a layout is the same before and after the local permutation. Hence

$$\sum_{j=1}^{m} \Delta|E_{l(j)}| = 0 \tag{3}$$

Parameterization of cache miss ratio: We parameterize each cache miss ratio, $p_{l(j)}$, by introducing a parametric variable, x_j, which due to the monotonicity of $p_{l(j)}$ is monotonically non-decreasing with j. This is represented as:

$$p_{l(j)} = x_j p_{l(1)} \tag{4}$$

where $1 \leq j \leq m$ and

$$1 = x_1 \leq x_2 \leq \cdots \leq x_{m-1} \leq x_m \leq \frac{1}{p_{l(1)}} \tag{5}$$

$p_{l(1)}$ is the cache miss ratio of the first edge, and $0 \leq p_{l(1)} \leq 1$.

The leftmost constraint of Eq. (5) is obvious because $p_{l(1)} = x_1 p_{l(1)}$. The rightmost constraint is computed from $p_{l(m)} = x_m p_{l(1)} \leq 1$, because all the cache miss values are less than or equal to 1.

By substituting the parameterization of cache miss ratios shown in Eq. (4) into Eq. (2) and canceling the constant $p_{l(1)}$, we have:

$$\sum_{j=1}^{m} \Delta|E_{l(j)}| x_j < 0. \tag{6}$$

This is our exact cache-oblivious metric.

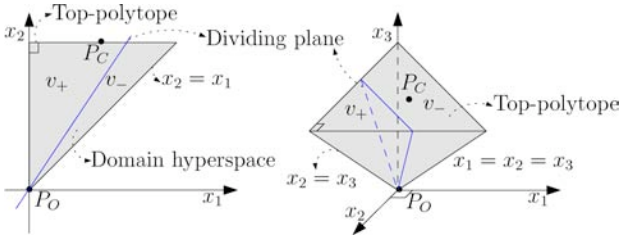


Figure 7: **Geometric volume computation:** The left figure shows a 2D geometric view of Eq. (6). The 3D version is shown in the right figure.

4.5 Geometric Formulation

We reduce the computation of the expression in Eq. (6) to a geometric volume computation in an m dimensional hyperspace. Geometrically, the parameterization domain represented in Eq. (5) defines a closed hyperspace in $\mathbb{R}^m$. We refer to this hyperspace as the *domain*. Eq. (6) defines a closed subspace within the domain of Eq. (5). Moreover, a dividing hyperplane defining this closed subspace passes through the point, $\{1, 1, \ldots, 1\} = P_O \in \mathbb{R}^m$, of the domain according to the constant edge property highlighted in Eq. (3). We also define the *top polytope* of the domain as the polytope intersecting the rightmost constraints of Eq. (5) with the open hyperspace defined by the other constraints of Eq. (5). Moreover, we define V_+ to be the volume of the subspace represented in Eq. (6) and V_- to be the volume of its complement within the closed domain. These geometric concepts in 2 and 3 dimensions are illustrated in Fig. 7.

Volume Computation: Intuitively speaking, the volume V_+ corresponds to the set of cache configurations parameterized by $\{x_j\}$ for which we expect a reduction in cache misses. Since we assume all configurations to be equally likely, we probabilistically reduce the number of cache misses by accepting a local permutation whenever V_+ is larger than V_-.

Complexity of Volume Computation: The computation of the volume of a convex polytope defined by $m + 1$ hyperplanes in m dimensions is a hard problem. The complexity of exact volume computation is $O(m^{m+1})$ [Lasserre and Zeron 2001] and an approximate algorithm of complexity $O(m^5)$ is presented in [Kannan et al. 1997]. In our application, each local permutation involves around 20–50 edges and these algorithms can be rather slow.

4.6 Fast and Approximate Metric

Given the complexity of exact volume computation, we use an approximate metric to check whether a local permutation would reduce the expected number of cache misses. In particular, we use a single sample point—the centroid of the top-polytope—as an estimate of $\{x_j\}$ and compute an approximate metric with low error.

Note that the dividing hyperplane between V_+ and V_- passes through the point P_O. Therefore, the ratio of V_+ to V_- is equal to the ratio of the $(m-1)$ dimensional areas formed by partitioning the top-polytope by the same dividing hyperplane. For example, in the 2D case, the result of volume comparison computed by substituting a centroid into Eq. (6) is exactly same as the result of the 2D area comparison between V_+ and V_-. This formulation extends to 3D, but it introduces some error. The error is maximized when the dividing plane is parallel to one of the edges of the top-polytope and it is minimized (i.e., exactly zero) when the plane passes through one of its vertices.

We generalize this idea to m dimensions. P_C, the centroid of a top-polytope, is defined as $(\frac{1}{m}, \frac{2}{m}, \ldots, \frac{m-1}{m}, \frac{m}{m}) \times \frac{1}{p_{l(1)}} + P_O$. By substituting P_C into Eq. 6 and canceling the constants, $\frac{1}{p_{l(1)}}$ and P_O, we have:

$$\sum_{j=1}^{m} \Delta|E_{l(j)}|j < 0 \tag{7}$$

If inequality (7) holds, we allow the local permutation. Based on this metric, we compute a layout, φ, that minimizes the number of cache misses.

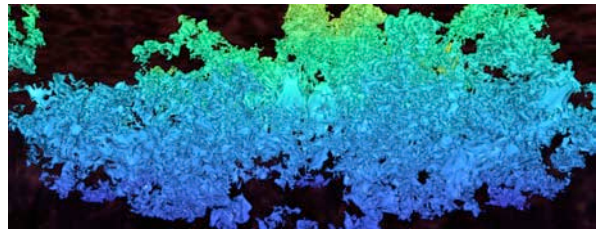


Figure 8: **Isosurface model:** This image shows a complex isosurface (100M triangles) generated from a 3D simulation of turbulent fluids mixing. Our layout reduces the vertex cache misses by more than a factor of four during view-dependent rendering. As a result, we improve the frame rate by 4 times as compared to prior approaches. We achieve a throughput of 90M tri/sec (at 30 fps) on a PC with an NVIDIA GeForce 6800 GPU.

Error Bounds on Approximate Metric: The approximate cache-oblivious metric has a worst case error of 26%, when the dividing hyperplane is parallel to one of the edges of the top-polytope. In practice, the worst case configuration is rare. In our benchmarks, we found that the actual error is typically much less than the worst case bound.

5 Layout Optimization

Given the cache-oblivious metric, our goal is to find the layout, φ, that minimizes the expected number of cache misses, defined in Eq. (1). This is a combinatorial optimization problem for graph layouts [Diaz et al. 2002]. Finding a globally optimal layout is NP-hard [Garey et al. 1976] due to the large number of permutations of the set of vertices. Instead, we use a heuristic based on *multilevel minimi a tion* that performs local permutations to compute a locally optimal layout.

5.1 Multilevel Minimization

Our multilevel algorithm consists of three main steps. First, a series of coarsening operations on the graph are computed. Next, we compute an ordering of vertices of the coarsest graph. Finally, we recursively expand the graph by reversing the coarsening operations and refine the ordering by performing *local permutations*. We will now describe each of these steps in more detail.

Coarsening Step: The goal of the coarsening phase is to cluster vertices in order to reduce the size of the graph while preserving the essential properties needed to compute a good layout. We have tried two approaches: clustering via graph partitioning [Karypis and Kumar 1998] and via streaming edge-collapse [Isenburg and Lindstrom 2004], using only the topological structure of the graph as criterion for collapsing edges. As mentioned above, geometric locality can be preserved by adding additional edges to the graph between spatially close vertices.

Ordering Step: Given the coarsest graph of a handful of vertices, we list all possible orderings of its vertices and compute the costs based on the cache-oblivious metric from Eq. (7). We choose a vertex ordering that has the minimum cost among all possible orderings.

Refinement Step: We reverse the sequence of coarsening operations applied earlier and exhaustively compute the locally optimal permutation of the subset of vertices involved in each corresponding refinement operation.

5.2 Local Permutation

We compute local permutations of the vertices during the ordering and refinement steps. A local permutation affects only a small number of vertices in the layout and changes the edge spans of those edges that are incident to these vertices. Therefore, we can efficiently recompute the cost associated with the metric. Each local permutation involves $k!$ possible orderings for k vertices, which during refinement replace their common parent in the evolving layout. For efficiency we restrict each coarsening operation to merge no more than $k = 5$ vertices at a time, and also limit the number of vertices in the coarsest graph to 5.

Model	Type	Vert. (M)	Tri. (M)	Size (MB)	Layout Comp. (min)
Dragon	s	0.4	0.8	33	0.25
Lucy	s	14.0	28.0	520	8
David	s	28.0	56.0	700	19
Double Eagle	c	77.7	81.7	3,346	56
Isosurface	i	50.5	100.0	2,543	49
Puget Sound	t	67.0	134.0	1,675	58
St. Matthew	s	186.0	372.0	9,611	176
Atlas	s	254.0	507.0	12,422	244

Table 1: **Layout Benchmarks:** Model complexity and time spent on layout computation are shown. Type indicates model type: s for scanned model, i for isosurface, c for CAD model, and t for terrain model. Vert. is the number of vertices and Tri. is the number of triangles of a model. Layout Comp. is time spent on layout computation.

Model	Double Eagle	Isosurface	St. Matthew
PoE	3	5	1
Frame rate	35	30	82
Rendering throughput(million tri./sec.)	47	90	106
Avg. Improvement	2.1	4.5	4.6
ACMR	1.58	0.75	0.72

Table 2: **View-Dependent Rendering** This table highlights the frame rate and rendering throughput for different models. We improve the rendering throughput and frame rates by $2.1 - 4.6$ times. The ACMR was computed with a buffer consisting of 24 vertices.

5.3 Out-of-Core Multilevel Optimization

The multilevel optimization algorithm needs to maintain an ordering of vertices along with a series of coarsening operations. For large meshes composed of hundreds of millions of vertices, it may not be possible to store all this information in main memory. In both of our graph partitioning and edge-collapse approaches, we compute a set of clusters, each containing a subset of vertices. Each cluster represents a subgraph and we compute an inter-cluster ordering among the clusters. We then follow the cluster ordering and compute a layout of all the vertices within each cluster using our multilevel minimization algorithm.

6 Implementation and Performance

In this section we describe our implementation and use cache coherent layouts to improve the performance of three applications: view-dependent rendering of massive models, collision detection between complex models, and isocontour extraction. Moreover, we used different kinds of models including CAD environments, scanned datasets, terrains, and isosurfaces to test the performance of cache coherent layouts. We also compare the performance of our metric with other metrics used for mesh layout.

6.1 Implementation

We have implemented our layout computation and out-of-core view-dependent rendering and collision detection algorithms on a 2.4GHz Pentium-4 PC with 1GB of RAM and a GeForce Ultra FX 6800 GPU with 256MB of video memory.

We use the METIS graph partitioning library [Karypis and Kumar 1998] for coarsening operations to lay out vertex and bounding volume hierarchies. Our current unoptimized implementation of the out-of-core layout computation processes about 30K triangles per sec. In the case of the St. Matthew model, our second largest dataset, layout computation takes about 2.6 hours.

Memory-mapped I/O: Our system runs on Windows XP and uses the operating system's virtual memory through memory mapped files [Lindstrom and Pascucci 2001]. Windows XP can map only up to 2GB of user-addressable space. We overcome this limitation by mapping a small portion of the file at a time and remapping when data is required from outside this range.

6.2 View-dependent rendering

View-dependent rendering and simplification are frequently used for interactive display of massive models. These algorithms precompute a multiresolution hierarchy of a large model (e.g. a vertex hierarchy). At runtime, a dynamic simplification of the model is computed by incrementally traversing the hierarchy until the desired pixels of error (PoE) tolerance in image space is met. Current view-dependent rendering algorithms are unable to achieve high polygon rendering throughput on the GPU for massive models composed of tens or hundreds of millions of triangles. It is not possible to compute rendering sequences at interactive rates for such massive models.

PoE	0.75	1	4	20
COL	0.71	0.72	0.73	0.74
SL	2.85	2.85	2.92	2.96

Table 3: **ACMR vs. PoE:** ACMRs are computed as we increase the PoE, i.e. use a more drastic simplification. The ACMRs of cache-oblivious layouts (COL) are still low even when a higher PoE is selected.

We use a clustered hierarchy of progressive meshes (CHPM) representation [Yoon et al. 2004b] for view-dependent refinement along with occlusion culling and out-of-core data management. The CHPM-based refinement algorithm is very fast and most of the frame time is spent in rendering the simplified model. We precompute a cache-oblivious layout (COL) of the CHPM and use it to reduce the cache misses for the vertex cache on the GPU. We computed layouts for three massive models including a CAD environment of a tanker with 127K separate objects (Fig. 2), a scanned model of St. Matthew (Fig. 1) and an isosurface model (Fig. 8). The details of these models are summarized in Table 1. We measured the performance of our algorithm along paths through the models. These paths are shown in the accompanying video.

6.2.1 Results

Table 2 highlights the benefit of COL over the simplification layout (SL), whose vertex layout and triangle layout are computed by the underlying simplification algorithm. We are able to increase the rendering throughput by a factor of of 2.1-4.6 times by precomputing a COL of the CHPM of each model. We obtain a rendering throughout of 106M triangles per second on average, with a peak performance of 145M triangles per second.

Average Cache Miss Ratio (ACMR): The ACMR is defined by the ratio of the number of accessed vertices to the number of rendered triangles for a particular vertex cache size [Hoppe 1999]. If the number of triangles in the model is roughly twice the number of vertices (e.g. the St. Matthew and isosurface models), then the ACMR is within the interval $[0.5, 3]$. Therefore, the theoretical upper bound on cache miss reduction is a factor of 6. For a cache of 24 vertices, we improve the ACMR by a factor of 3.95 and get a 4.5 times speedup in the rendering throughput. On the other hand, if the number of vertices in the model is roughly the same as the number of triangles, as in the tanker model, then the ACMR is within the interval $[1, 3]$ and the upper bound on cache miss reduction is 3 times. For this model, we improve the ACMR by a factor of 1.89 and the rendering throughput by a factor of 2.1. To verify the cache-oblivious nature of our layouts, we also simulated a FIFO vertex cache of configurable size and measured the ACMR as a function of cache size (Fig. 10). Table 3 shows the ACMR achieved by varying the PoE in the St. Matthew model.

6.3 Collision Detection

Many collision detection algorithms use bounding volume hierarchies to accelerate the interference computations [Lin and Manocha 2003]. We use cache-oblivious layouts to improve the performance of collision detection algorithms. In particular, we compute layouts of OBB-trees [Gottschalk et al. 1996] and use them to accelerate collision queries within a dynamic simulator. The collision detection algorithm traverses the bounding volume hierarchy of each model and checks for overlap between the OBBs and triangle pairs. We have tested the performance of our collision detection algorithm in a rigid body simulation where 20 dragons (800K triangles each) drop on the Lucy model (28M triangles). The details of these models are shown in Table 1. Fig. 9 shows a snapshot from our simulation.

We compared the performance of our cache-oblivious layout with the RAPID library [Gottschalk et al. 1996]. The OBBs are precomputed and stored in memory-mapped files and only the ordering of the hierarchy is modified. We compared our cache-oblivious layout with a depth-first layout (DFL) of OBB-trees. The DFL is computed

Figure 9: **Dynamic Simulation:** Dragons consisting of 800K triangles are dropping on the Lucy model consisting of 28M triangles. We obtain 2 times improvement by using COL on average.

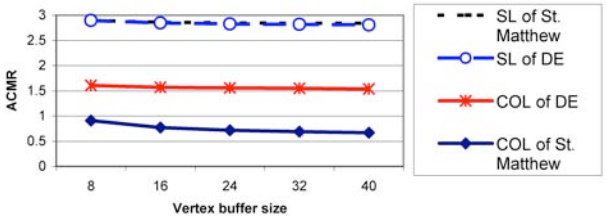


Figure 10: **ACMR vs. cache size:** ACMRs of cache-oblivious layout (COL) and simplification layout (SL) of the St. Matthew and double eagle tanker are shown. As the cache size increases, the improvement of COL becomes larger, and is 3.95 at a cache size of 24 in the St. Matthew model. Note that the lower bound on ACMR is 0.5 in St. Matthew and 1 in the double eagle tanker. The two SL curves almost overlap.

by traversing the hierarchy from its root node in a depth-first order. We chose DFL because it preserves the spatial locality within the bounding volume hierarchy.

6.3.1 Results

We are able to achieve 2 times improvement in performance over the depth-first layout on average. This is mainly due to reduced cache misses, including main memory page faults. We observe more than 2 times improvement whenever there are more broad contact regions. Such contacts trigger a higher number of page faults and in such situations we obtain a higher benefit from cache-oblivious layouts. The query times of collision detection during the dynamic simulation are shown in Fig. 11.

6.4 Isocontour Extraction

The problem of extracting an isocontour from an unstructured dataset frequently arises in geographic information systems and scientific visualization. We use an algorithm based on seeds sets [van Kreveld et al. 1997] to extract the isocontour of a single-resolution mesh. The running time of this algorithm is dominated by the traversal of the triangles intersecting the contour itself.

We use this algorithm to extract isocontours of a large terrain (Fig. 5) and compute equivalent geometric queries such as extracting ridge lines of a terrain[1] and cross-sections of large geometric models.

6.4.1 Comparison with other layouts

We compare the the performance of the isocontouring algorithm on four models, each stored in five different layouts. In addition to our cache-oblivious layout, we also store the meshes in geometric $X/Y/Z$ orders (vertices sorted by their position along the corresponding coordinate axis) and in spectral sequencing order [Diaz et al. 2002]. We use edge-collapse as the coarsening step for computing cache-oblivious layouts and store all meshes in a streaming format [Isenburg and Lindstrom 2004], which allows us to quickly compute the on-disk mesh data structure in a preprocess.

Table 4 reports the time in seconds to compute an isocontour and a ridge line for the terrain models and to compute cross-sections of the other 3D models. The tests have been performed on a 1.3GHz Itanium Linux PC with 2GB of main memory. We take advantage of

[1] For extracting a ridge line the seed point is a saddle and the propagation goes upward to the closest maxima instead of following an isocontour.

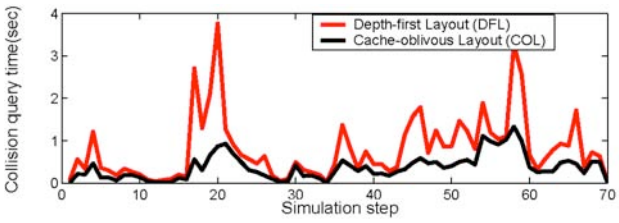


Figure 11: **Performance of Collision Detection:** Average query times for collision detection between the Lucy model and the dragon model with COL and DFL are shown. We obtain 2 times improvement in the query time on average.

Model	Puget Sound		Lucy	David	Atlas
Out. edg.	223K (Contour)	14K (Ridge)	17K (Section)	22K (Section)	38K (Section)
Cac. Obl.	026 (000.5)	003 (000.03)	03.3 (.04)	05.9 (.057)	010 (000.09)
Geom. X	232 (227.8)	001 (000.04)	01.2 (.04)	00.2 (.051)	015 (000.09)
Geom. Y	218 (215.5)	195 (185.10)	39.1 (.09)	60.7 (.103)	419 (379.78)
Geom. Z	011 (000.6)	135 (113.81)	26.1 (.09)	45.5 (.102)	443 (382.60)
Spec. Seq.	150 (127.3)	023 (000.04)	21.0 (.06)	43.1 (.068)	088 (000.10)

Table 4: **Isocontouring.** Time in seconds (on a 1.3GHz linux PC with 2GB of memory) for extracting an isocontour (or equivalent geometric queries) for several models stored each in five different mesh layouts: cache-oblivious, with vertices sorted by X/Y/Z geometric coordinate, and spectral sequencing. In parentheses we report the time for second immediate re-computation of the same contour when all the cache levels in the memory hierarchy have been loaded. In all cases, the performance of our cache-oblivious layout is comparable to the one optimized for the particular geometric query. This demonstrates the benefit of our layout for general applications.

the 64-bit architecture and memory map the entire model. We do not perform any explicit paging. This way we ensure that we do not bias the results in favor of any particular layout.

The empirical data shows that our cache-oblivious layout minimizes the worst case cost of generic coherent traversals. The three layouts that are sorted by geometric direction along the X, Y, and Z axis show that the worst case performance is at least one order of magnitude slower than the best case, which is achieved by the layout that happens to be perfectly aligned along the query direction. The spectral sequencing layout also does not perform well since the geometric query is unlikely to follow its streaming order. Our cache-oblivious layout consistently exhibits good performance.

The running times reported in parentheses in Table 4 are for a second immediate re-computation of the same contour, ridge line, or cross-section. They demonstrate the performance when all the cache levels have been loaded by the first computation. In this case our cache-oblivious layout is always as fast as the the best case and can be two times to several orders of magnitude faster than the worst case. More importantly, this test demonstrates the cache-oblivious nature of the approach since performance advantages at different scales are achieved both when disk paging is necessary and when only internal memory and L2 caches are involved. In case of the Puget Sound terrain model, our cache-oblivious layout is the only layout that takes advantage of loaded cache levels for both the queries (i.e., isocontour and ridge line extraction).

7 Analysis and Limitations

We present a novel metric based on edge span distribution and CMRF to determine whether a local permutation on a layout reduces the expected number of cache misses. In practice, our algorithm computes layouts for which a high fraction of edges have very small edge spans. At the same time, a small number of edges in the layout can have a very large edge span, as shown in Fig. 6. This distribution of edge spans improves the performance because edges with small edge span increase the probability of a cache hit, while the actual length of very high-span edges has little impact on the likelihood of a cache hit.

Our multilevel minimization algorithm is efficient and produces reasonably good results for our applications. Moreover, our minimization algorithm maps very well to out-of-core computations and is able to handle very large graphs and meshes with hundreds of millions of triangles. We have applied our cache-oblivious layouts to

models with irregular distribution of geometric primitives or irregular structure, for which prior algorithms based on space-filling curves may not work well.

Limitations: Our metric and layout computation algorithm has several limitations. The assumptions we make about invariance and monotonicity of CMRFs may not hold for all applications, and our minimization algorithm does not necessarily compute a globally optimal solution. Our cache-oblivious layouts result in good improvements primarily in applications where the running time is dominated by data access.

8 Conclusion and Future Work

We have presented a novel approach for computing cache-oblivious layouts of large meshes. We make no assumptions about the runtime access pattern and we compute an ordering that results in high locality. We show that our formulation can be extended to compute layouts of bounding volume and multiresolution hierarchies of large meshes. We use a probabilistic model to minimize the number of cache misses. Our preliminary results indicate that our metric works well in practice for reducing cache misses. Furthermore, we computed cache-oblivious layouts of different kinds of geometric datasets including scanned models, isosurfaces, terrain, and CAD environments with irregular distributions of primitives. We used our layouts to improve the performance of view-dependent rendering, collision detection and isocontour extraction by $2 - 20$ times without any modification of the algorithm or runtime applications.

There are many avenues for future work. In this paper, we have only considered cache-oblivious access and we can obtain further improvement in performance by taking into account cache parameters to design an improved metric. We would like to use our layout to improve the performance of algorithms for processing and manipulation of large meshes including simplification, compression, parameterization, smoothing, isosurface extraction, shadow generation, and approximate collision detection [Yoon et al. 2004a]. We would also like to use our graph-based formulation to compute cache-coherent layouts for other kinds of datasets, including point primitives and unstructured volumetric grids. Finally, we would like to combine data layouts with application specific techniques to increase program locality to further improve the cache access pattern and efficiency.

Acknowledgments

This work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, DARPA/RDECOM Contract N61339-04-C-0043 and Intel. Some of the work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48. The St. Matthew, Lucy, and Atlas models are courtesy of the Digital Michelangelo Project at Stanford University. The isosurface model is courtesy of the LLNL ASCI VIEWS Visualization project and the Double Eagle tanker is courtesy of Newport News Shipbuilding. We would like to thank Brandon Lloyd, Brian Salomon, Avneesh Sud, Martin Isenburg, Dawoon Jung, Elise London, and the members of UNC Walkthrough and Gamma group for their feedback on an earlier draft of the paper and technical discussions.

References

ARGE, L., BRODAL, G., AND FAGERBERG, R. 2004. Cache oblivious data structures. *Handbook on Data Structures and Applications*.

BARTHOLDI, J., AND GOLDSMAN, P. 2004. Multiresolution indexing of triangulated irregular networks. In *IEEE Transaction on Visualization and Computer Graphics*, 484–495.

BOGOMJAKOV, A., AND GOTSMAN, C. 2002. Universal rendering sequences for transparent vertex caching of progressive meshes. In *Computer Graphics Forum*, 137–148.

CIGNONI, P., MONTANI, C., ROCCHINI, C., AND SCOPIGNO, R. 2003. External memory management and simplification of huge meshes. In *IEEE Transaction on Visualization and Computer Graphics*, 525–537.

COLEMAN, S., AND MCKINLEY, K. 1995. Tile size selection using cache organization and data layout. *SIGPLAN Conference on Programming Language Design and Implementation*, 279–290.

CORRÊA, W. T., KLOSOWSKI, J. T., AND SILVA, C. T. 2003. Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of PVG 2003 (6th IEEE Symposium on Parallel and Large-Data Visualization and Graphics)*, 1–8.

DEERING, M. F. 1995. Geometry compression. In *SIGGRAPH 95 Conference Proceedings*, Addison Wesley, R. Cook, Ed., Annual Conference Series, ACM SIGGRAPH, 13–20. held in Los Angeles, California, 06-11 August 1995.

DIAZ, J., PETIT, J., AND SERNA, M. 2002. A survey on graph layout problems. *ACM Computing Surveys 34*, 313–356.

FRANQUESA-NIUBO, M., AND BRUNET, P. 2003. Collision prediction using mktrees. *Proc. CEIG*, 217–232.

FRIGO, M., LEISERSON, C., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. *Symposium on Foundations of Computer Science*, 285–297.

GAREY, M. R., JOHNSON, D., AND STOCKMEYER, L. 1976. Some simplified np-complete graph problems. *Theoretical Computer Science 1*, 237–267.

GOPI, M., AND EPPSTEIN, D. 2004. Single-strip triangulation of manifolds with arbitrary topology. In *EUROGRAPHICS*, 371–379.

GOTTSCHALK, S., LIN, M., AND MANOCHA, D. 1996. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, 171–180.

HEBER, G., BISWAS, R., AND GAO, G. 2000. Self-avoiding walks over adaptive unstructured grids. In *Concurrency: Practice and Experience*, 85–109.

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. *Proc. of ACM SIGGRAPH*, 269–276.

ISENBURG, M., AND GUMHOLD, S. 2003. Out-of-core compression for gigantic polygon meshes. In *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, vol. 22, 935–942.

ISENBURG, M., AND LINDSTROM, P. 2004. Streaming meshes. Tech. Rep. UCRL-CONF-201992, LLNL.

KANNAN, R., LOVASZ, L., AND SIMONOVITS, M. 1997. Random walks and an $O(n^5)$ time algorithm for volumes of convex sets. *Random Structures and Algorithms*, 1–50.

KARNI, Z., BOGOMJAKOV, A., AND GOTSMAN, C. 2002. Efficient compression and rendering of multi-resolution meshes. In *IEEE Visualization*, 347–354.

KARYPIS, G., AND KUMAR, V. 1998. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 96–129.

LASSERRE, J. B., AND ZERON, E. S. 2001. A laplace transform algorithm for the volume of a convex polytope. *Journal of the ACM*, 1126–1140.

LIN, M., AND MANOCHA, D. 2003. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*.

LINDSTROM, P., AND PASCUCCI, V. 2001. Visualization of large terrains made easy. *IEEE Visualization*, 363–370.

OLIKER, L., LI, X., HUSBANDS, P., AND BISWAS, R. 2002. Effects of ordering strategies and programming paradigms on sparse matrix computations. In *SIAM Review*, 373–393.

PASCUCCI, V., AND FRANK, R. J. 2001. Global static indexing for real-time exploration of very large regular grids. *Super Computing*, 225–241.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. Qsplat: A multiresolution point rendering system for large meshes. *Proc. of ACM SIGGRAPH*, 343–352.

SAGAN, H. 1994. *Space-Filling Curves*. Springer-Verlag.

SEN, S., CHATTERJEE, S., AND DUMIR, N. 2002. Towards a theory of cache-efficient algorithms. *Journal of the ACM 49*, 828–858.

SILVA, C., CHIANG, Y.-J., CORREA, W., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics. *Visualization'02 Course Notes*.

VAN KREVELD, M., VAN OOSTRUM, R., BAJAJ, C., PASCUCCI, V., AND SCHIKORE, D. R. 1997. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, ACM Press, New York, 212–220.

VELHO, L., AND GOMES, J. D. 1991. Digital halftoning with space filling curves. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, T. W. Sederberg, Ed., vol. 25, 81–90.

VITTER, J. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 209–271.

WILSON, A., LARSEN, E., MANOCHA, D., AND LIN, M. C. 1999. Partitioning and handling massive models for interactive collision detection. *Computer Graphics Forum (Proc. of Eurographics) 18*, 3, 319–329.

YOON, S.-E., AND MANOCHA, D. 2005. Cache-Oblivious Layouts of Bounding Volume Hierarchies. Tech. rep., University of North Carolina-Chapel Hill.

YOON, S., SALOMON, B., LIN, M. C., AND MANOCHA, D. 2004. Fast collision detection between massive models using dynamic simplification. *Eurographics Symposium on Geometry Processing*, 136–146.

YOON, S.-E., SALOMON, B., GAYLE, R., AND MANOCHA, D. 2004. Quick-VDR: Interactive View-dependent Rendering of Massive Models. *IEEE Visualization*, 131–138.

© [2006] IEEE. Reprinted, with permission, from:

Mesh Layouts for Block-Based Caches

Sung-Eui Yoon and Peter Lindstrom

IEEE Transactions on Visualization and Computer Graphics
Volume: 12, Issue: 5
On page(s): 1213-1220

Mesh Layouts for Block-Based Caches

Sung-Eui Yoon, *Member, IEEE*, and Peter Lindstrom, *Member, IEEE*

Abstract—Current computer architectures employ caching to improve the performance of a wide variety of applications. One of the main characteristics of such cache schemes is the use of block fetching whenever an uncached data element is accessed. To maximize the benefit of the block fetching mechanism, we present novel cache-aware and cache-oblivious layouts of surface and volume meshes that improve the performance of interactive visualization and geometric processing algorithms. Based on a general I/O model, we derive new cache-aware and cache-oblivious metrics that have high correlations with the number of cache misses when accessing a mesh. In addition to guiding the layout process, our metrics can be used to quantify the quality of a layout, e.g. for comparing different layouts of the same mesh and for determining whether a given layout is amenable to significant improvement. We show that layouts of unstructured meshes optimized for our metrics result in improvements over conventional layouts in the performance of visualization applications such as isosurface extraction and view-dependent rendering. Moreover, we improve upon recent cache-oblivious mesh layouts in terms of performance, applicability, and accuracy.

Index Terms—Mesh and graph layouts, cache-aware and cache-oblivious layouts, metrics for cache coherence, data locality.

1 Introduction

Many geometric algorithms utilize the computational power of CPUs and GPUs for interactive visualization and other tasks. A major trend over the last few decades has been the widening gap between proces sor speed and main memory access speed. As a result, system ar chitectures increasingly use caches and memory hierarchies to avoid memory latency. The access times of different levels of a memory hierarchy typically vary by orders of magnitude. In some cases, the running time of a program is as much a function of its cache access pattern and efficiency as it is of operation count [10].

One of the main characteristics of memory hierarchies is the use of block fetching whenever there is a cache miss. Block fetching schemes assume that there is high spatial coherence of data accesses that allow repeated cache hits. Therefore, to maximize the benefit of block fetch ing, it is important to organize and access the data in a cache coherent manner. There are two standard techniques for minimizing the number of cache misses: computation reordering and data layout optimization. Computation reordering is performed to improve data access locality, e.g. using compiler optimizations or application specific hand tuning. On the other hand, data layout optimization reorders the data in mem ory so that its layout matches the expected access pattern. In this paper, we focus on computing cache coherent layouts of polygonal and poly hedral meshes, in which vertices and cells (e.g. triangles, tetrahedra) are organized as linear sequences of elements.

Many layouts and representations (triangle strips [8], space filling curves [24], stream [14] and cache oblivious [30] layouts) have been proposed for cache coherent access. However, previous layouts have either been specialized for a particular cache [5, 13] or applica tion [14, 17], including graph and sparse matrix computations [6], or are constructive [3,7,8,20,23,30,31] with no measure of global layout quality needed to establish optimality, relative ranking among layouts, and criterion for driving more general optimization strategies. Further more, while prior metrics may be suitable for their intended applica tions, they are not particularly good estimators of layout quality in the context of block based caching.

Main Results: We propose novel metrics and methods to evalu ate and optimize the locality of mesh layouts. Based on a general I/O model, we derive cache aware and cache oblivious metrics that corre

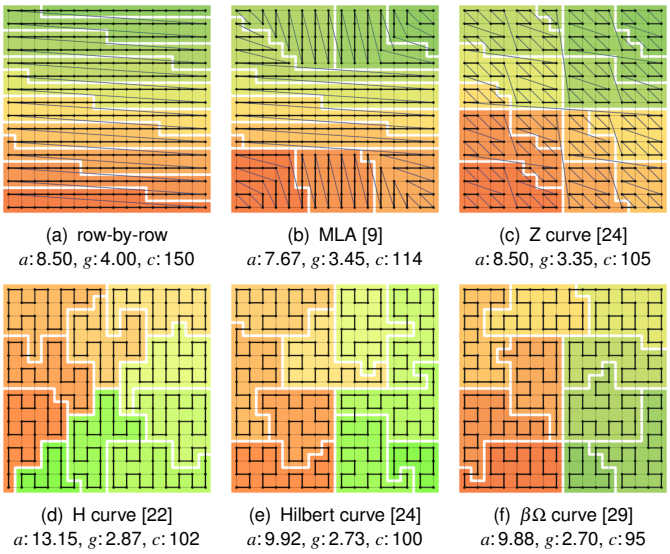

(a) row-by-row a: 8.50, g: 4.00, c: 150
(b) MLA [9] a: 7.67, g: 3.45, c: 114
(c) Z curve [24] a: 8.50, g: 3.35, c: 105
(d) H curve [22] a: 13.15, g: 2.87, c: 102
(e) Hilbert curve [24] a: 9.92, g: 2.73, c: 100
(f) $\beta\Omega$ curve [29] a: 9.88, g: 2.70, c: 95

Fig. 1. **Layouts and Coherence Measures for a** 16×16 **Grid:** a and g correspond to the arithmetic and geometric mean index difference of adjacent vertices; c denotes the *cut*, or number of edges that straddle cache blocks. Each block except the last contains 27 vertices. MLA is known to minimize a, and $\beta\Omega$ is near-optimal with respect to g for grids. Our new cache-oblivious measure, g, correlates better than a with the cut and, hence, the number of cache misses.

late well with the number of cache misses when accessing a mesh in a reasonably coherent, though not specific manner. Using these metrics, we employ a multi level recursive optimization method to efficiently compute cache coherent layouts of massive meshes consisting of hun dreds of millions of vertices. We also observe that recursively con structed layouts, regardless of additional ordering criteria, in general have good locality according to our metric.

Benefits: Our approach offers the following advantages over the current state of the art:

- **Generality:** Our algorithm is applicable to any data set whose expected access pattern can be expressed in a graph structure.
- **Simplicity:** Our metrics are concise and easy to implement.
- **Accuracy:** Our derived metrics correlate well with the observed number of runtime cache misses. Hence our metrics are useful for constructing good layouts.
- **ciency:** Our metrics can quickly quantify the quality of a given layout. If according to the metric a layout already is coher

• *The authors are with the Lawrence Livermore National Laboratory, E mail: {sungeui, pl}@llnl.gov.*

Manuscript received 31 March 2006; accepted 1 August 2006; posted online 6 November 2006.
For information on obtaining reprints of this article, please send e mail to: tvcg@computer.org.

ent, no further work is needed to reorganize it, which saves time and effort when dealing with very large meshes.

- **er rmance:** Computed layouts optimized for our metrics show performance improvements over other layouts.

We apply our cache coherent layouts in two applications: isosur face extraction from tetrahedral meshes and view dependent render ing of polygonal meshes. In order to illustrate the generality of our approach, we compute layouts of several kinds of geometric models, including a CA environment, scanned models, an isosurface, and a tetrahedral mesh. We use these layouts directly without having to mod ify the runtime application. Our layouts reduce the number of cache misses and improve the overall performance.

2 Related Work

Cache efficient algorithms have received considerable attention over last two decades in theoretical computer science and in the compiler literature. These algorithms include models of cache behavior [28] and compiler optimizations based on tiling, strip mining, and loop interchanging all of these algorithms have shown to reduce cache misses [4]. Cache efficient algorithms can be classified as computa tion reordering and data layout optimization.

2.1 Computation Reordering

Computation reordering strives to achieve a cache coherent order of runtime operations in order to improve program locality and reduce the number of cache misses. This is typically performed using compiler optimizations or application specific hand tuning.

At a high level, computation reordering methods can be classi fied as either *cache-aware* or *cache-oblivious*. Cache aware algo rithms utilize knowledge of cache parameters, such as cache block size [28]. On the other hand, cache oblivious algorithms do not as sume any knowledge of cache parameters [10]. There is a consider able amount of literature on developing cache efficient computation reordering algorithms for specific problems and applications, includ ing numerical programs, sorting, geometric computations, matrix mul tiplication, FFT, and graph algorithms. More details are given in recent surveys [2,28]. In visualization and computer graphics, out of core al gorithms are designed to handle massive models using finite memory, typically by limiting access to a small, cached subset of a model. A re cent survey of these algorithms and their applications is given in [25].

2.2 Data Layout Optimization

The order in which data elements are stored can have a major im pact on runtime performance. Therefore, there have been considerable efforts on computing cache coherent layouts of the data to match its anticipated access pattern. The following possibilities have been con sidered.

Graph and Matrix Layouts: Graph and matrix layout problems fall in the class of combinatorial optimization problems. Their main goal is to find a linear layout of a graph or matrix that minimizes a specific objective function. Well known minimization problems in clude linear arrangement (sum of edge lengths, i.e. index differences of adjacent vertices), bandwidth (maximum edge length), profile (sum of maximum per vertex edge length), and wavefront (maximum front size see also [14]). This work has been widely studied and an exten sive survey is available [6]. While potentially improving data coher ence, these layouts are not optimal or necessarily good for block based caching, as we shall see later.

Rendering Sequences: Modern GPUs maintain a small buffer to reuse recently accessed vertices. In order to maximize the benefits of vertex buffers for fast rendering, triangle reordering is necessary. This approach was pioneered by eering [5]. The resulting ordering of triangles is called a triangle strip or a rendering sequence. Hoppe [13] casts the triangle reordering as a discrete optimization problem with a cost function dependent on a specific vertex buffer size. everal techniques improve the rendering performance of view dependent al gorithms by computing rendering sequences not tailored to a particular cache size [3,7,17,30].

Processing Sequences: Isenburg et al. [15] proposed process ing sequences as an extension of rendering sequences to large data processing. A processing sequence represents an indexed mesh as interleaved triangles and vertices that can be streamed through main memory [14]. Global mesh access is restricted to a fixed traversal or der only localized random access to the buffered part of the mesh is supported as it streams through memory. This representation is mostly useful for of ine applications (e.g., simplification and compression) that can adapt their computations to the fixed ordering.

Space-Filling Curves: Many algorithms use space filling curves [24] to compute cache friendly layouts of volumetric grids or height fields. These layouts are widely used to improve the performance of image processing [27] and terrain or volume visualization [20,23]. A standard method of constructing a mesh layout based on space filling curves is to embed the mesh or geometric object in a uniform structure that contains the space filling curve. Gotsman and indenbaum inves tigated the spatial locality of space filling curves [12]. Motivated by searching and sorting applications, Wierum [2] proposed using a log arithmic measure of edge length, resembling one of our new metrics, for analyzing space filling curve layouts of regular grids. Our results, however, indicate that space filling curve embedding does not work well for meshes that have an irregular distribution of geometric primi tives. ecently, oon et al. proposed methods for cache coherent lay out of polygonal meshes [30] and bounding volume hierarchies [31]. These methods are constructive in nature and require making a se quence of binary decisions without relying on a global measure of locality. Hence, these methods depend on a particular optimization framework and cannot be used to judge the quality of a layout.

3 Computation Models

In this section we describe an I/O model for our cache representation. We also propose both a graph representation and probability model to capture the likelihood of accessing mesh elements.

3.1 I/O Model

Most modern computers use hierarchies of memory levels, where each level of memory serves as a *cache* for the next level. Memory hier archies have two main characteristics. First, lower levels are larger in size and farther from the processor, and hence have slower access times. econd, data is moved in blocks of many elements between different memory levels. ata is initially stored in the lowest mem ory level, typically the disk. A transfer is performed whenever there is a cache miss between two adjacent levels of the memory hierarchy. ue to this *block fetching* mechanism, cache misses can be reduced by storing in the same block data elements that are accessed together.

We use a simple *two-level I/O-model* defined by Aggarwal and it ter [1] that captures the two main characteristics of a memory hier archy. This model assumes a fast memory called cache consisting of M blocks and a slower infinite memory. The size of each cache block is B therefore, the total cache size is $M \times B$. ata is transferred between the levels in blocks of consecutive elements.

3.2 Graph Representation

Our layout algorithm requires the specification of a directed graph that represents the anticipated runtime access pattern, and in which each node represents a data element (e.g., mesh vertex or triangle). A di rected arc (i, j) between two nodes indicates an expectation that node j may be accessed immediately after node i. We chose directed graphs to represent access patterns, in contrast to the undirected graphs used in oon et al. [30], as we do not necessarily assume symmetry in node access patterns.

et $G = (N, A)$ be a directed graph, where N is a set of nodes and A is a set of directed arcs. The layout problem reduces to computing the one to one mapping $\varphi : N \to \{1, ..., |N|\}$ of nodes to positions in the layout that minimizes the expected number of cache misses.

We also require probabilities (as weights) for each node and arc that represent the likelihood of accessing them at runtime. We derive these probabilities here based on the graph structure by considering an in finite random walk over the graph. For each directed arc, $(i, j) \in A$,

let $\Pr(j|i)$ denote the probability of accessing node j next, given that i was the previously accessed node. We define P to be a *probability transition matrix* whose (i,j)th element equals $\Pr(j|i)$. Furthermore, let $\Pr(i)$ denote the probability of accessing node i among all nodes N, and let x be the n by 1 column vector whose ith component equals $\Pr(i)$. Given an initial state of node access probabilities, we update each node s probability in x by traversing the node s outgoing arcs, each with probability of access $\Pr(j|i)$. This update is equivalent to premultiplying x by P^T. If applied repeatedly, x will ultimately con verge to a stable equilibrium

$$x = P^T x \tag{1}$$

of node access probabilities, and we are interested in finding this con figuration. It should be clear that a vector x that satisfies this criterion is an eigenvector of P^T with a corresponding eigenvalue equal to one.

Finally, we define $\Pr(i,j)$ to be the probability of accessing a node j from another node i in the equilibrium state (i.e. the probability of accessing the arc (i,j) among all arcs A):

$$\Pr(i,j) = \Pr(i) \times \Pr(j|i) \tag{2}$$

Specialization for Meshes: We specialize the probability com putations for meshes by assuming that given a mesh vertex i we are equally likely to access any of its neighboring vertices via traversal of an out going arc. Therefore, $\Pr(j|i) = \frac{1}{\deg(i)}$ if i and j are adjacent, and equals zero otherwise. Here $\deg(i)$ is the out degree of vertex i (for meshes the in degree equals the out degree). Using eigenanalysis we can then show that $\Pr(i) = \frac{\deg(i)}{|A|}$ (see Appendix A). That is, the probability of accessing a vertex is proportional to its degree. As a re sult, according to q. (2) we have $\Pr(i,j) = \frac{1}{|A|}$. In other words, each edge in the mesh is equally likely to be accessed.

A mesh layout consists of a pair of independent linear sequences of vertices and cells. The vertex layout requires a directed graph, $G = (N, A)$, where N is the set of mesh vertices and A is a set of arcs representing a likely access pattern. For simplicity, we choose A to be the set of mesh edges in our applications. For applications such as col lision detection that require geometric coherence, we may optionally include additional arcs that join spatially nearby elements. The layout of cells can be constructed in a similar manner, e.g. by considering the edges in the dual graph. From here on, we use the term layout to refer to the vertex layout for the sake of clarity.

4 Cache-Aware Layouts

In this section we derive cache aware metrics based on our computa tion models and describe an efficient layout algorithm. Our goal is to compute the expected number of cache misses when accessing a node by traversing a single arc. ince in our framework arcs are equally likely to be accessed, this measure generalizes trivially to any number of accesses. We consider two cases of this problem: the cache consists of exactly one block ($M = 1$), or of multiple blocks ($M > 1$).

4.1 Single Cache Block, $M = 1$

ince the cache can only hold one block, a cache miss occurs whenever a node is accessed that is stored in a block different from the cached block. In other words, a cache miss is observed when we traverse an arc, (i,j), and the block containing node j is different from the block that holds i. Therefore, the expected number of cache misses, $ECM_1^B(\varphi)$, of a layout, φ, for a single block cache with block size B nodes can be computed as:

$$ECM_1^B(\varphi) = \sum_{\substack{(i,j)\in A \\ \varphi^B(i)\neq\varphi^B(j)}} \Pr(i,j) = \frac{1}{|A|} \sum_{(i,j)\in A} S(|\varphi^B(i) - \varphi^B(j)|) \tag{3}$$

where $\varphi^B(i) = \lceil \frac{\varphi(i)}{B} \rceil$ denotes the index of the block in which i resides and $S(x)$ is the unit step function $S(x) = 1$ if $x > 0$ and $S(x) = 0$ other wise. Intuitively speaking, $ECM_1^B(\varphi)$ is the number of arcs whose two nodes are stored in different blocks, i.e. the *cut*, divided by the total number of arcs in the graph.

Layout algorithm: Constructing a layout optimized for $ECM_1^B(\varphi)$ reduces to a k way graph partitioning problem. ach directed arc has a constant weight, $\frac{1}{|A|}$, and we partition the input graph into $k = \lceil \frac{n}{B} \rceil$ different sets of vertices. ince graph partitioning is an P hard problem, we rely on heuristics. One good heuristic is the multi level algorithm implemented in the M TI library [18]. Once the directed graph is partitioned, the ordering among blocks and the order of vertices within each block do not matter in our I/O model.

4.2 Multiple Cache Blocks, $M > 1$

We now assume that the cache holds multiple blocks. As in the single block case, a cache miss can occur only when we traverse an arc, (i,j), that crosses a block boundary, i.e., $\varphi^B(i) = B_i \neq B_j = \varphi^B(j)$. How ever, unlike the single block case, block B_j may already be stored in the cache when we access j. Therefore, to compute the expected num ber of cache misses for a multi block cache, we must also consider the probability, $\Pr_{\text{cached}}(B_j)$, that B_j is cached among the M blocks.

In theory, $\Pr_{\text{cached}}(B_j)$ can be computed by exhaustively generating all possible access patterns for which $B(j)$ is already cached when we access j from i. uch block access patterns take on the form $(B_j, \ldots, B_i, B_j)$, where at most M different blocks are accessed before B_j is accessed the second time. Then, the expected number of cache misses, $ECM_M^B(\varphi)$, for $M > 1$ cache blocks of a layout, φ, is:

$$ECM_M^B(\varphi) = \sum_{\substack{(i,j)\in A \\ \varphi^B(i)\neq\varphi^B(j)}} \Pr(i,j)(1 - \Pr_{\text{cached}}(\varphi^B(j))) \tag{4}$$

Approximation: Unfortunately, generating all possible block ac cess patterns is prohibitively expensive because of its exponential combinatorial nature. Furthermore, we found that it is not feasible to approximate $\Pr_{\text{cached}}(B_j)$ within an error bound without considering a very large number of access patterns. However, we conjecture that there is strong correlation between $ECM_1^B(\varphi)$ and the observed num ber of cache misses, OCM_M^B, for multiple blocks, which $ECM_M^B(\varphi)$ is designed to capture. To support this conjecture, we computed ten dif ferent layouts of a uniform grid (Fig. 4) and measured the number of cache misses incurred in a U based cache during a random walk be tween neighboring nodes. We performed walks long enough until the observed number of cache misses, OCM_1^B, for a single block cache correlated well ($R^2 > 0.$) with our estimate ECM_1^B. In this case, the correlation between ECM_1^B and OCM_M^B, for a multi block cache ($M = 5$ and $M = 5^2$) was observed to be very high ($R^2 = 0.$ 7).

ote that there is, however, a pathological case that suggests that our conjecture is not always valid. To illustrate this, we first compute a spectral layout, $\varphi_{spectral}$, and a cache aware layout, φ_{aware}, opti mized for a single block. We modify φ_{aware} to produce a new lay out, φ_{random}, by performing a series of random swaps of two adjacent nodes from different blocks until φ_{random} has the exact same edge cut as $\varphi_{spectral}$. Although the two layouts, $\varphi_{spectral}$ and φ_{random}, have the same value for ECM_1^B, we observed that φ_{random} results in many fewer cache misses for multi block caches. We attribute this result to the fact that it does not take many local swaps to rapidly boost the edge cut. However, these swaps do not adversely affect locality since they generally increase only the connectivity between already adjacent cache blocks (i.e. blocks spanned by crossing arcs) that are likely to be cached together in a multi block cache.

5 Cache-Oblivious Layouts

A cache oblivious layout allows good performance across a wide range of block and cache sizes, which is important due to the hierarchi cal nature of most cache systems, as well as for portability across plat forms with different cache parameters. In this section we present our cache oblivious metric that measures the expected number of cache misses for any block size B. For a single block cache, we derive two metrics: one with no restriction on B, and one that restricts B to be a power of two. We conclude by brie y discussing multi block caches.

5.1 Single Cache Block, $M = 1$

We now relax the assumption of B being a particular block size and consider metrics optimized for many (even an infinite number of) block sizes. We first assume that the cache holds only a single block.

In ec. 4.1, we derived a cache aware metric ECM_1^B for a single block cache with fixed block size B. We here generalize this metric by considering all possible block sizes B, each with its own likelihood $w(B)$ of being employed in an actual cache. Clearly $w(B)$ is difficult to estimate in practice, but we will consider some reasonable choices be low. We then express our single block cache oblivious metric, ECM_1, in terms of the cache aware metric, ECM_1^B:

$$\begin{aligned} ECM_1(w,\varphi) &= \sum_{B=1}^{t} w(B)\, ECM_1^B(\varphi) \\ &= \frac{1}{|A|} \sum_{(i,j)\in A} \sum_{B=1}^{t} w(B)\, S(|\varphi^B(i) - \varphi^B(j)|) \end{aligned} \tag{5}$$

where t is the maximum block size considered.

Assumptions: For simplicity, we will assume that a layout may start anywhere within a block with uniform probability.[1] Hence we replace the binary step function $S(x)$ with the probability $\Pr_{\text{cross}}(\ell_{ij}, B)$ that an arc (i,j) of length $\ell_{ij} = |\varphi(i) - \varphi(j)|$ crosses a block boundary. We have (see Appendix B):

$$\Pr_{\text{cross}}(\ell, B) = \begin{cases} 1 & \text{if } B \le \ell \\ \frac{\ell}{B} & \text{otherwise} \end{cases} \tag{6}$$

q. (5) then becomes:

$$\begin{aligned} ECM_1(w,\varphi) &= \frac{1}{|A|} \sum_{(i,j)\in A} \left(\sum_{B=1}^{\ell_{ij}} w(B) + \sum_{B=\ell_{ij}+1}^{t} w(B) \frac{\ell_{ij}}{B} \right) \\ &\approx \frac{1}{|A|} \sum_{(i,j)\in A} \left(\int_0^{\ell_{ij}} w(B)\, dB + \int_{\ell_{ij}}^{t} w(B) \frac{\ell_{ij}}{B}\, dB \right) \end{aligned} \tag{7}$$

where we have used integrals to approximate summations in order to simplify the math. One can show that this approximation intro duces a negligible error. Finally, we will attempt to present our metrics in their simplest form, e.g. scaling or addition of constants can be factored out without affecting the relative ranking among lay outs reported by a metric. More generally, we consider ECM and $f(ECM)$ equivalent metrics as long as f is monotonically increasing, i.e. $ECM(\varphi) < ECM(\varphi') \Rightarrow f(ECM(\varphi)) < f(ECM(\varphi'))$ ote that we make such simplifications only to the final metric value of ECM and not to terms like ℓ and w that make up a metric.

5.2 Arithmetic Progression of Block Size

Without further information, perhaps the most natural choice for w is to assume that each block size is equally likely, i.e. $w_a(B) = \frac{1}{t}$. tated differently, we assume that the block size is drawn uniformly from an arithmetic sequence $B \in \{k\}_{k=1}^{t}$. We have:

$$\begin{aligned} ECM_1(w_a,\varphi) &\approx \frac{1}{|A|t} \sum_{(i,j)\in A} \left(\int_0^{\ell_{ij}} dB + \int_{\ell_{ij}}^{t} \frac{\ell_{ij}}{B}\, dB \right) \\ &= \frac{1}{|A|t} \sum_{(i,j)\in A} \ell_{ij}(1 + \log t - \log \ell_{ij}) \end{aligned} \tag{8}$$

As t grows, $1 + \log t - \log \ell_{ij}$ approaches $\log t$. After proper scaling, we arrive at our *arithmetic cache-oblivious metric*, COM_a:

$$COM_a(\varphi) = \frac{1}{|A|} \sum_{(i,j)\in A} \ell_{ij} \tag{ }$$

[1]This is not unrealistic; e.g. a call to `malloc` may return an address not aligned with a memory page or lower level cache block. Furthermore, we can show that this assumption is not needed if the mesh is large enough.

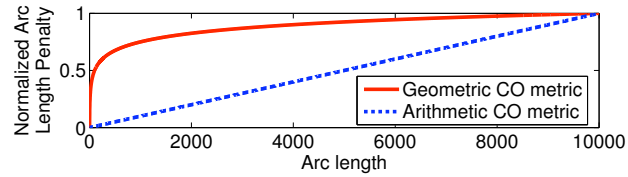


Fig. 2. **Arc Length Dependence of Cache-Oblivious Metrics:** These curves show the per-arc penalty as a function of arc length. Our geometric metric, COM_g, places a large premium on very short arcs while de-emphasizing small differences in long arcs, whereas our arithmetic metric, COM_a, prefers layouts with a more uniform arc length.

ote that COM_a is the arithmetic mean arc length, or equivalently the metric for linear arrangement. Therefore, the optimal layout COL_a for this metric is the well known minimum linear arrangement (M A) [6].

5.3 Geometric Progression of Block Size

The assumption that all block sizes are equally likely and should be weighed uniformly is questionable. First, this diminishes the impor tance of small block sizes since we may include arbitrarily many large block sizes by increasing t. econd, the hierarchical relationship be tween nested cache levels is often geometric, which suggests that we should optimize for cache block sizes at different scales, and not par ticularly for every possible block size. Indeed, most block sizes em ployed in practice are a power of two bytes (e.g., 32B for 1, 64B for 2, 4 B for disk blocks). We thus consider block sizes drawn uni formly from a geometric sequence $B \in \{2^k\}_{k=0}^{\lg t}$:

$$\begin{aligned} ECM_1(w_g,\varphi) &\approx \frac{1}{|A| \lg t} \sum_{(i,j)\in A} \left(\int_0^{\lg \ell_{ij}} dk + \int_{\lg \ell_{ij}}^{\lg t} \frac{\ell_{ij}}{2^k}\, dk \right) \\ &= \frac{1}{|A| \log t} \sum_{(i,j)\in A} \left(\int_1^{\ell_{ij}} \frac{1}{B}\, dB + \int_{\ell_{ij}}^{t} \frac{\ell_{ij}}{B^2}\, dB \right) \\ &= \frac{1}{|A| \log t} \sum_{(i,j)\in A} \left(\log \ell_{ij} + 1 - \frac{\ell_{ij}}{t} \right) \end{aligned} \tag{10}$$

We note that our restriction on block size is equivalent to using a weight function $w_g(B) = \frac{1}{B}$ over all block sizes. As t grows, we reach our *geometric cache-oblivious metric*, COM_g:

$$COM_g(\varphi) = \frac{1}{|A|} \sum_{(i,j)\in A} \log \ell_{ij} = \log \left(\prod_{(i,j)\in A} \ell_{ij}^{\frac{1}{|A|}} \right) \tag{11}$$

The right hand side expression is the logarithm of the geometric mean arc length. ince $\log x$ is monotonic, we may optionally use the geo metric mean directly as cache oblivious metric, as is done in Fig. 1. From here on, we will however include the logarithm when evaluating COM_g. For simplicity of presentation, we here consider only power of two byte blocks and single byte nodes, however one can show that any geometric sequence $B \in \{b^k/m\}_k$ with base b and node size m leads to the same metric COM_g.

5.4 Properties

Fig. 2 shows how the metrics COM_a and COM_g change as a function of arc length for a single arc. This graph shows that COM_g puts a big premium on very short arcs, and does not excessively penalize long arcs. This is justified by the fact that once an arc in a layout is long enough to cause a cache miss, lengthening it will not drastically increase the probability of cache misses.

We note that COM_g and COM_a are instances of the power mean, $M_p = \lim_{q\to p} \frac{1}{|A|} \sum_{(i,j)\in A} \ell_{ij}^q \;^{1/q}$, with $p = 0$ and $p = 1$, respectively. quivalently, our new measure, COM_g, can be viewed as an exten sion of the well known *p-sum* [16] family of graph theoretic measures that includes linear arrangement ($p = 1$), discrete spectral sequencing ($p = 2$), and bandwidth ($p = \infty$) [6]. Although the case $0 < p < 1$ is considered in [21], we are not aware of prior work for which $p = 0$.

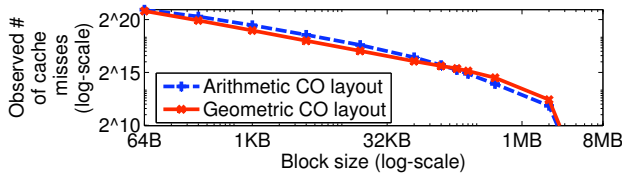


Fig. 3. **Cache Misses vs. Cache Block Size:** We simulated LRU-based caching for several block sizes and measured the number of cache misses during random walks between adjacent vertices of the dragon mesh using layouts optimized for our two metrics. The arithmetic layout results in better performance for block sizes between 196KB and 8MB, or 97% of the full range of block sizes considered here. On the other hand, the geometric layout causes fewer cache misses in 73% of all tested power-of-two-byte block sizes.

5.5 Validation

We derived two cache oblivious metrics for different assumptions on cache block sizes. To determine whether the two metrics correlate with the actual number of cache misses, we performed a validation test on the dragon model. First we computed two different layouts: an arithmetic layout, COL_a, and a geometric layout, COL_g, optimized for our two metrics. Then, we performed random walks on the mesh and measured the number of cache misses for a range of block sizes. As can be seen in Fig. 3, COL_g results in fewer actual cache misses over most (73) of the tested power of two block sizes, although the *range* of block sizes over which COL_a performs better is much wider (7). (This is not readily evident from Fig. 3 due to the logarithmic scale.) Hence, when considering also all possible non power of two block sizes, we expect COL_a to perform better on average. Because cache block sizes employed in practice are often nested powers of two bytes, a layout optimized for COM_g is likely to be useful in practice.

As an additional sanity check, we evaluated our COM_g metric on several layouts of a uniform grid (Fig. 4). We computed $\beta\Omega$ indexing [2], Hilbert, curve [24], H order [22], M A [ ], row by row, and diagonal by diagonal layouts of a 256×256 uniform grid (see also Fig. 1). We measured the actual number of cache misses during random walks for a 4 B single block cache. We found strong correlation, $R^2 = 0.$ 8, between the value of COM_g and the observed number of cache misses for this experiment.

According to COM_g, layouts with increasing space filling structure have better locality. We exhaustively searched for the layout of a 4×4 grid that minimizes COM_g and found the $\beta\Omega$ space filling curve [2] to be the optimum, closely followed by the Hilbert curve, confirming conventional wisdom. For an 8×8 grid, we had to restrict our exhaus tive search to recursive layouts due to combinatorial explosion of the search space. Here again we found $\beta\Omega$ to be optimal for all $2^n \times 2^n$ grids investigated we have not found any other space filling layout with lower COM_g. By considering non recursive layouts produced by an alternative (non exhaustive) optimization method, we have however found layouts slightly better than $\beta\Omega$ for 8×8 and larger grids.

5.6 Layout Algorithms

To construct layouts optimized for COM_a, we may use existing M A layout algorithms. The M A problem is known to be P hard and its decision version is P complete [11]. However its importance in many applications has inspired a wide variety of approximations based on heuristics, including spectral sequencing [16]. A more di rect, multi level algorithm for M A is presented in [1]. ince lay outs minimizing COM_a are not practically useful for our purpose, we do not further consider them here.

To minimize COM_g, we propose using a minor variation on the multi level recursive layout algorithm from [30] that is based on graph partitioning and local permutations. This algorithm recursively par titions a mesh into k (e.g., $k = 4$) vertex sets using M TI [18] and computes the locally best ordering of these k sets among all k permu tations. The process is repeated until each set contains a single vertex. ote that contrary to the layouts in [30], which depend on a partic ular constructive algorithm, our global metrics allow us to apply other optimization methods, e.g. based on simulated annealing, genetic al gorithms, steepest decent, etc. We have, however, found the multi level recursive method to be quite effective for minimizing COM_g, which can be explained by this metric s goal to measure locality at dif ferent scales. That is, optimally partitioning a mesh into k sets amounts to computing a cache aware layout for a block size $\frac{n}{k}$. Indeed, even when not applying any local permutations, i.e. by randomly ordering nodes within each set, we observed that the resulting layouts yield only 5 more cache misses than layouts locally optimized for COM_g.

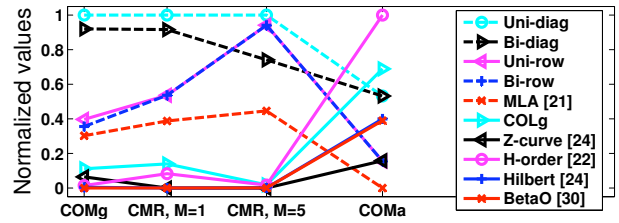


Fig. 4. **Correlation between Cache Misses and Our Metrics:** We computed ten layouts of a 256×256 grid and measured the values of COM_g and COM_a and the number of cache misses incurred during random walks on the grid. We found that COM_g and the number of cache misses for a single-block and multi-block ($M = 5$) cache correlated well, with correlation coefficients $R^2 = 0.98$ and $R^2 = 0.81$, respectively. COM_a, on the other hand, did not predict cache misses well, with $R^2 = -0.19$ and $R^2 = -0.32$, respectively. For this parallel-coordinates plot we linearly mapped each quantity to the interval $[0,1]$. In the row-by-row and diagonal-by-diagonal layouts, *uni-* indicates that we traverse each row/diagonal from left to right; *bi-* indicates that we alternate direction. *CMR* denotes cache miss ratio.

5.7 Multiple Cache Blocks, $M > 1$

We can derive a multi block cache oblivious metric ECM_M based on the corresponding cache aware metric ECM_M^B as we did for the single block case. But since evaluating ECM_M^B is computationally infeasible, we again must resort to using the single block metric ECM_1 as an approximation. As evidenced by Fig. 4, we obtain good correlation ($R^2 = 0.81$) between COM_g and the observed number of cache misses when using a cache with multiple blocks.

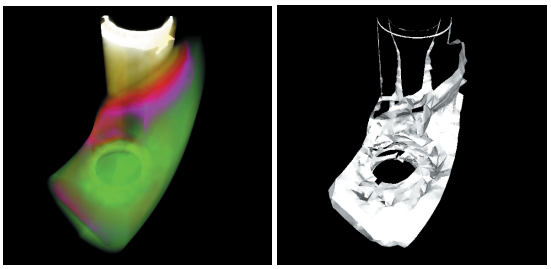
6 Evaluating Layouts

In this section we propose two simple ways of evaluating the quality of layouts using our metrics. If a layout is deemed to be good, it can be used without expensive reordering, which is especially useful for massive meshes with millions of vertices. While our cache aware and geometric cache oblivious metrics allow ranking different layouts of a graph or mesh, it is not obvious how to map the numerical values they report for a single layout to some notion of absolute quality or closeness to optimality. If a tight lower bound for either of these two metrics is known for a graph, we can compare this bound with the met ric value of the layout to determine its quality. Unfortunately, no such bound for general graphs or meshes is known. However, empirical ev idence suggests that for optimized layouts of unstructured triangle and tetrahedral meshes with bounded vertex degrees, COM_g depends only on the average vertex degree and not on mesh size. For ten bench mark triangle meshes, spanning 35 to 880 vertices and optimized for COM_g, we observed the geometric mean edge length to fall in the narrow range 4.48 4.87. While pathological cases can be constructed for which this measure is unbounded with respect to mesh size, we hypothesize that mesh layouts with geometric mean close to the aver age degree are sufficiently near optimal to be quite useful in practice. Future work is needed to investigate this hypothesis in more detail.

An alternative, albeit more expensive, approach to measuring lay out quality, is to compare a given layout, φ, with the optimized layout, φ^*, constructed by our cache aware or cache oblivious layout meth ods. ince the goal of this comparison is primarily to avoid lengthy optimization, we limit the evaluation to a small subgraph, G', extracted from the input graph, G, and optimize only the layout φ^* for G'. We present our algorithm for constructing G' below.

6.1 Algorithm

We compute a small subgraph, $G' = (N', A')$, of an input graph, $G = (N, A)$, with $N' \subseteq N$ and $A' = \{(i, j) \in A : i, j \in N'\}$. Our algo rithm requires at most two parameters: a ratio, $p = \frac{|N'|}{|N|}$, that specifies



(a) Spx tetrahedral mesh (b) Extracted isosurface

Fig. 5. **Isosurface Extraction:** We measured the performance of isosurface extraction from unstructured tetrahedral meshes. Using our cache-oblivious layout of the Spx volume mesh with 140K vertices, we were able to reduce cache misses by 5%–50% compared to other layouts. Moreover, our cache-oblivious layout yields only less than 2% fewer cache misses compared to our cache-aware layout.

the target subgraph size $|N'|$, and for the cache aware case the block size, B. For efficiency, p should be small, e.g., 0.1 1 .

Our evaluation algorithm has two major steps: 1) sampling the input graph and 2) constructing an optimized layout of the sampled subgraph.

1. Sampling the input graph: We first randomly select a start node in the input graph and add it to the subgraph. In order to obtain a subgraph that well re ects the quality of the full layout, φ, and that allows capturing the cache behavior of a probable sequence of successive accesses to the graph, we construct connected subsets of G via breadth first region growing from the start node and add all visited nodes to the subgraph. We also add to the subgraph all the arcs from A that join nodes in N'. We continue growing until the total number of nodes in the subgraph is about k times (e.g., $k = 5$) the block size, B. If B is not specified, as in the cache oblivious case, we simply set B to be 8 B, which is commonly used for large page sizes of virtual memory.

Once the region growing stops, we add for each node $i \in N'$ all other nodes that map to the same cache block as i. We do this to avoid having to account for intra block holes in the layout that might otherwise be unfairly utilized in the optimized layout. This also ensures that we do not accidentally miss cut edges between sampled blocks with respect to the cache aware metric. We do, however, allow for holes due to unsampled blocks since for incoherent layouts there could be arbitrarily many such blocks. We then repeat step 1, randomly selecting a new start node, until the number of nodes $|N'|$ is close to $p|N|$.

2. Constructing an optimized layout of the subgraph: We apply our cache aware or cache oblivious layout algorithm to construct a new layout, φ^*, of the subgraph and evaluate the chosen metric for φ^* and φ. We simply use these numbers to approximate the expected numbers of cache misses of the input layout and the optimized layout for the full graph. If there is big difference between these estimates for the subgraph, it is likely beneficial to compute an improved layout of the full graph using our layout algorithm.

We used this approach to quickly evaluate the original layouts of our benchmark models. ven for the largest meshes, our approximate method takes less than 10 seconds. We found that we are able to predict the metric values of these full layouts within 15 error using subgraphs of only 6 40 vertices, even though the original meshes have as many as tens of millions of vertices.

7 Results

In this section we highlight the performance improvements obtained using our cache coherent layouts in two different applications: isosurface extraction and view dependent rendering. We implemented our layout computation algorithm on a 2.8GHz Pentium 4 PC with 1GB of AM. We used the M TI graph partitioning library [18] to compute our cache aware and cache oblivious layouts. Also, our metric has been integrated into *OpenCCL*, an open source library for the layout computation. Our current unoptimized implementation of the out of core layout computation, based in large part on the method

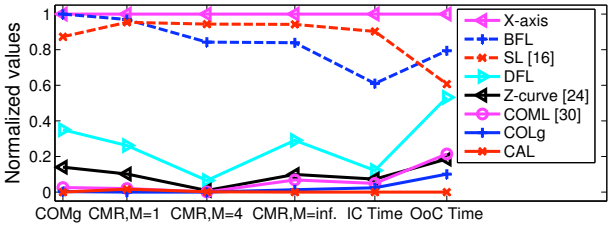


Fig. 6. **Comparison with Different Layouts in Iso-Surface Extraction:** We compared our cache-aware (CAL) and cache-oblivious (COL_g) layouts with breadth-first (BFL), depth-first (DFL), Z-curve, spectral (SL), cache-oblivious mesh (COML), and geometrically sorted layout along the X-axis. We simulated a LRU-based cache with a block size of 4KB and measured cache misses during isosurface extraction from the Spx model shown in Fig. 5. We also measured the out-of-core running time of extracting the surface from disk (OoC) and a second time from memory (IC). Due to the different scales, each quantity is normalized to the unit interval. We observe that our geometric cache-oblivious metric has strong correlation with both cache misses and running time. CMR indicates cache miss ratio.

in [30, 32], processes about 15 triangles per second, which is comparable in speed to other out of core layout methods [14, 30].

Inducing a Layout: In order to reduce the layout computation time, we compute only one of the vertex and triangle layouts and induce the other layout rather than computing the layouts separately. First, we construct a vertex layout since the number of vertices is typically smaller than the number of triangles. Hence, the processing time of a vertex layout is smaller than that of a triangle layout. Then, as we access each vertex of the vertex layout, we sequentially store all triangles incident on the vertex that have not already been added to the triangle layout. We found that using the induced layouts at runtime causes a minor runtime performance loss in our benchmark, less than 5 compared to using layouts that are computed separately.

7.1 Isosurface Extraction

The problem of extracting an isocontour from an unstructured dataset frequently arises in geographic information systems and scientific visualization. Many efficient isosurface extraction methods employ seed sets [26] to grow an isosurface by traversing only the cells intersecting the isosurface. The running time of such an algorithm is dominated by the traversal of the cells intersecting the contour. We efficiently extract an isosurface from a seed cell by making a depth first traversal, thereby accessing the volume mesh in a reasonably cache coherent manner.

7.1.1 Comparison with Other Layouts

We compared the performance of the isosurface extraction algorithm on the px volume mesh (Fig. 5) consisting of 140 vertices and 820 tetrahedra. We stored the volume mesh using eight different layouts (see Fig. 6). We measured cache misses during two invocations of the same isosurface extraction. uring the first extraction, we ensured that no part of the model was cached in main memory therefore, loading the data from disk was the major bottleneck. uring the second extraction of the same isosurface, all the data was already loaded into main memory therefore, 1 and 2 cache misses dominated. As seen in Fig. 6, we observe strong correlations between our geometric cache oblivious metric and both cache misses and running times of the isosurface extraction. Moreover, our cache oblivious layout yields only a slight performance decrease compared to our cache aware layout optimized for a block size of 4 B. Our layouts furthermore result in up to two times speedup over the other layouts.

7.2 View-dependent rendering

iew dependent rendering is frequently used for interactive display of massive models. These algorithms precompute a multiresolution hierarchy of a large model, and at run time dynamically simplify the model as long as the desired pixels of error (Po) tolerance in image space is met. We use the clustered hierarchy of progressive meshes (CHPM) representation from [32] for view dependent rendering. The CHPM based view dependent method is fast and most of the frame time is

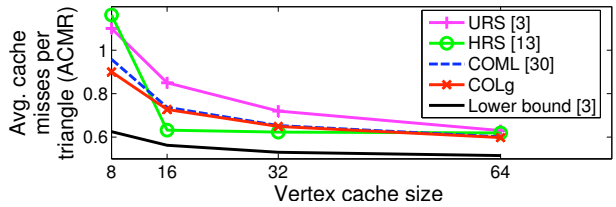


Fig. 7. **View-Dependent Rendering of Bunny Model:** The ACMRs of our cache-oblivious layout (COL_g) are close to the lower bound on ACMR. COL_g consistently outperforms universal rendering sequences (URS), cache-oblivious mesh layout (COML), and Hoppe's rendering sequences (HRS) at cache sizes 8 and 64; HRS is optimized for a cache size in the range 12–16.

spent rendering the simplified model. We precomputed different lay outs and compared their cache performance for three different models, including a CA environment of a power plant consisting of 12 mil lion triangles, the tanford bunny model, and a subset of the ichtmeyer Meshkov isosurface model consisting of 100M triangles.

To compare the cache performance of different layouts during view dependent rendering, we use the *average cache miss ratio* (ACM), which is defined as the ratio of the number of vertex cache misses to the number of rendered triangles for a particular vertex cache size [13]. To verify the cache oblivious nature of our layouts, we also simulated a FIFO vertex cache of configurable size and measured the ACM as a function of cache size.

7.2.1 Comparison with Other Layouts

We compared our cache oblivious layout with *universal rendering sequences* (U) [3], Hoppe s rendering sequences (H) [13], em bedding using a order space filling curve, and oon et al. s cache oblivious mesh layouts (COM) [30]. H is considered a cache aware layout since it is optimized for a given cache size and replace ment policy. On the other hand, the curve, U , and COM are considered cache oblivious layouts since they do not take advantage of any cache parameters.

Fig. 7 shows ACM s of different rendering sequences for the tan ford bunny model. ince the number of triangles in the model is roughly twice the number of vertices, the ACM is within the interval $[0.5, 3]$. Moreover, it is possible to compute a lower bound $0.5 + O(\frac{1}{k})$ on the ACM , where k is the size of vertex cache [3]. As can be seen, the ACM s of our layout are close to the lower bound and con sistently lower than those of U and COM among all tested cache sizes. Although our layout yields more cache misses at cache size 16 than H , which is optimized for this size, our layout shows superior performance at cache sizes 8 and 64. This is greatly due to the cache oblivious nature of our layouts, which achieve good performance over a wide spectrum of cache sizes rather than one particular size. These observed results also correlate with cache miss estimates reported by our COM_g metric, which for the bunny model predict our layout to be 5 , 17 , and 31 better than COM , H , and U , respectively. We observed similar results on the isosurface model.

Fig. 8 shows ACM s for rendering the power plant model, which has a very irregular geometric distribution, using our layout and others, including the curve. ince space filling curves mainly retain locality between mesh elements if their geometric distribution is regular, we would not expect the curve layout to result in good performance on this irregular mesh. As evidenced, our layout consistently yields better performance than the curve and the other layouts. This is correctly predicted by our COM_g metric, which estimates our layout to be 5 , 2 , and 241 better than COM , H , and the curve, respectively.

Finally, we measured ACM s of COL_g and H at a cache size of 32 as we decreased the resolution of the mesh by subsampling ver tices and triangles via edge collapse operations. The relative positions of surviving elements were retained in the simplified meshes. ince our COL_g layout maintains locality at multiple scales, it is likely to be coherent in the simplified mesh. As expected, Fig. shows our layout to be more robust to simplification than H , which is optimized only for the finest mesh resolution.

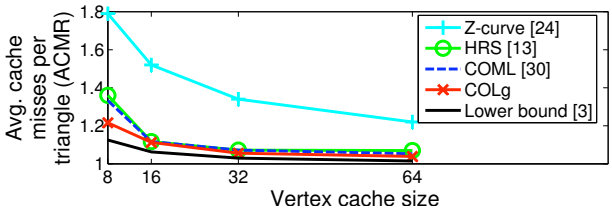


Fig. 8. **View-Dependent Rendering of Power Plant Model:** Our new cache-oblivious layout (COL_g) consistently performs better than the Z-curve, Hoppe's rendering sequences (HRS), and Yoon et al.'s cache-oblivious mesh layout (COML) on the power plant model, which has an irregular geometric distribution.

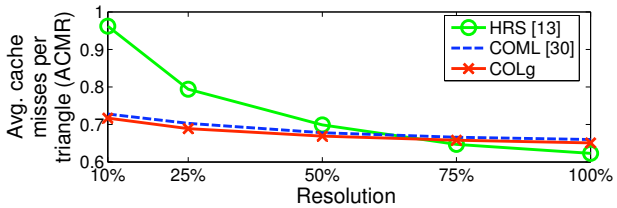


Fig. 9. **ACMRs of Different Resolutions:** These curves correspond to ACMRs for our cache-oblivious layout and Hoppe's rendering sequences over several mesh resolutions for a cache size of 32. The horizontal axis indicates the fraction of triangles relative to the full resolution mesh.

7.3 Comparison with COML

The cache oblivious mesh layouts (COM) proposed by oon et al. [30] bear many similarities with ours, in particular with respect to performance. Whereas we consistently achieve modest performance improvements over COM , our main motivation for extending their work was their lack of a global measure of layout coherence, which manifests itself as a number of limitations. COM makes use of a *local* probabilistic measure of coherence that allows making decisions whether a local permutation of a small number of vertices is likely or not to reduce the number of cache misses. This measure does not ex tend to a whole mesh, and it does not satisfy transitivity. uppose that a local permutation, A, is deemed better than another local permutation, B, and B is better than another a permutation, C. However, according to the COM measure, A is not necessarily better than C. For these two reasons, the COM measure cannot be used to evaluate and com pare the quality of layouts. On the other hand, our new measure is *global* and transitive, and furthermore correlates with expected cache misses. And because this measure is global and easy to compute, it can be easily integrated into layout computation frameworks other than the one presented here. Finally, we started from a general I/O model from which both cache aware and cache oblivious measures emerged. It is not obvious that COM lends itself to computing cache aware layouts since it fundamentally does not incorporate cache parameters.

7.4 Limitations

While in many respects an improvement over COM , our new ap proach still has several limitations. Our layout quality estimation tech nique considers only a subset of a mesh, and may not be indicative of quality for unsampled portions. The greedy and recursive multi level method we use for mesh layout is not likely to find a true optimum, and may not even compute a layout better than the input layout. Moreover, our multi level method relies heavily on the M TI graph partitioning algorithm, which itself is based on heuristics. Therefore, the partition ing results may be far from optimal, as evidenced for example when applied to a uniform grid. Here the cache oblivious layouts produced with M TI and our metric result in up to 60 more cache misses than achieved by space filling curves, which in a sense provide op timal partitions of such grids. Our layouts furthermore help perfor mance primarily in applications where the running time is dominated by data access. Finally, we require the specification of a graph to rep resent a typical access pattern. Whereas the connectivity structure of a mesh can often be used to define this graph, certain applications may need a different set of nodes, arcs, or access probability assignments than those automatically generated by our method.

8 Conclusion and Future Work

We have presented novel cache aware and cache oblivious metrics for evaluating the layout quality of unstructured graphs and meshes. We based our metrics on a two level I/O model. Our cache aware metric is reduced to counting the number of arcs whose nodes are stored in different cache blocks. For the cache oblivious case, we derived two different metrics based on different assumptions on what cache block sizes may be used for caching. When applied equally to all possible block sizes, our cache oblivious metric reduces to the graph theoretic metric for minimum linear arrangement. When only power of two block sizes are considered, our cache oblivious metric is a function of logarithmic arc length. quivalently, these metrics correspond to arithmetic and geometric mean arc length. We show that there is good correlation between our metrics and the number of observed cache misses for two different visualization applications. Moreover, we im prove the performance of these applications by 5 100 over several other common layouts.

There are many avenues for future work. In addition to address ing some of the limitations of our current approach, we are working on mesh compression schemes to further reduce expensive I/O access time. One major challenge is to design a compression method that both preserves the mesh layout and supports random access so that a mesh can be accessed using a coherent but not predetermined traver sal. Another challenge would be to extend our current approach to support maintaining coherent layouts of non static mesh connectivity, e.g. due to animation, simplification, or refinement. There is consid erable room for exciting theoretical work on the properties of our new, simple cache oblivious metric, such as proving what layouts are op timal for 2 and 3 grids, and whether our metric produces similar space filling optimal layouts for unstructured meshes. Finally, we expect our metric to have uses in applications other than visualization, such as acceleration of shortest path and other graph computations.

A Eigenanalysis for Meshes

et x be a column vector with ith component $x_i = \Pr(i) = \frac{\deg(i)}{|A|}$. Then, x is an eigenvector of the probability transition matrix, P^T, since:

$$(P^T x)_i = \sum_{j \in N} \Pr(j|i)\Pr(j) = \sum_{j\,:\,(j,i)\in A} \frac{1}{\deg(j)} \frac{\deg(j)}{|A|} = \sum_{j\,:\,(j,i)\in A} \frac{1}{|A|} = \frac{\deg(i)}{|A|} = x_i$$

Therefore, x_i is the stationary probability $\Pr(i)$ that node i is accessed.

B Expected Edge Cut

Consider an edge of length ℓ. The two nodes of the edge are always stored in different blocks when $B \leq \ell$, irrespective of where within a block the layout starts. ow consider the case $B > \ell$. There are ℓ different positions for which the edge crosses a block boundary, and B different positions at which the layout may start. Therefore, the probability $Pr_{cross}(\ell, B)$ that the edge crosses a block boundary is $\frac{\ell}{B}$.

Acknowledgments

We would like to thank Ajith Mascarenhas for his isosurfacing codes Martin Isenburg for his out of core mesh decomposition code Fabio Bernardon, oao Comba, and Claudio ilva for their unstructured tetrahedra rendering program and inesh Manocha and the anony mous reviewers for their useful feedback. The bunny and dragon mod els are courtesy of tanford University. The spx model is courtesy of Bruno otrosso (lectricite de France). The power plant model is courtesy of an anonymous donor. This work was supported by the OCA project (05 018) and was performed un der the auspices of the U. . epartment of nergy by the University of California, awrence ivermore ational aboratory under Contract o. W 7405 ng 48.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *ommunications of A M*, 31:1116–1127, 1988.
[2] L. Arge, G. Brodal, and R. Fagerberg. Cache Oblivious Data Structures. *andboo on ata tructures and Applications*, 2004.
[3] A. Bogomjakov and C. Gotsman. Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes. *omputer raphics Forum*, 137–148. 2002.
[4] S. Coleman and K. McKinley. Tile Size Selection using Cache Organization and Data Layout. *LAN onference on rogramming Language esign and mplementation*, 279–290, 1995.
[5] M. F. Deering. Geometry Compression. *A M A* , 13–20. 1995.
[6] J. Diaz, J. Petit, and M. Serna. A Survey of Graph Layout Problems. *A M omputing urveys*, 34(3):313–356, 2002.
[7] P. Diaz-Gutierrez, A. Bhushan, M. Gopi, and R. Pajarola. Constrained Strip Generation and Management for Efficient Interactive 3D Rendering. *omputer raphics nternational*, 115–121. 2005.
[8] F. Evans, S. S. Skiena, and A. Varshney. Optimizing Triangle Strips for Fast Rendering. *EEE isuali ation*, 319–326. 1996.
[9] P. Fishburn, P. Tetali, and P. Winkler. Optimal linear arrangement of a rectangular grid. *iscrete Mathematics*, 213(1):123–139, 2000.
[10] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *Foundations of omputer cience*, 285–297. 1999.
[11] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical omputer cience 1*, 237–267, 1976.
[12] C. Gotsman and M. Lindenbaum. On the metric properties of discrete space-filling curves. *EEE Transactions on mage rocessing*, 5(5):794–797, 1996.
[13] H. Hoppe. Optimization of mesh locality for transparent vertex caching. *A M A* , 269–276, 1999.
[14] M. Isenburg and P. Lindstrom. Streaming Meshes. *EEE isuali ation*, 231–238, 2005.
[15] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large Mesh Simplification using Processing Sequences. *EEE isuali ation*, 465–472, 2003.
[16] M. Juvan and B. Mohar. Optimal linear labelings and eigenvalues of graphs. *iscrete Applied Mathematics*, 36(2):153–168, 1992.
[17] Z. Karni, A. Bogomjakov, and C. Gotsman. Efficient compression and rendering of multi-resolution meshes. *EEE isuali ation*, 347–54. 2002.
[18] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *ournal of arallel and istributed omputing*, 96–129, 1998.
[19] Y. Koren and D. Harel. A Multi-scale Algorithm for the Linear Arrangement Problem. *Lecture Notes n omputer cience*, 2573:296–309, 2002.
[20] P. Lindstrom and V. Pascucci. Visualization of Large Terrains Made Easy. *EEE isuali ation*, 363–370, 2001.
[21] G. Mitchison and R. Durbin. Optimal numberings of an $N \times N$ array. *AM ournal on iscrete and Algebraic Methods*, 7(4):571–582, 1986.
[22] R. Niedermeier, K. Reinhardt, and P. Sanders. Towards optimal locality in mesh-indexings. *iscrete Applied Mathematics*, 117(1):211–237, 2002.
[23] V. Pascucci and R. J. Frank. Global Static Indexing for Real-time Exploration of Very Large Regular Grids. *upercomputing*. 2001.
[24] H. Sagan. *pace Filling urves*. Springer-Verlag, 1994.
[25] C. Silva, Y.-J. Chiang, W. Correa, J. El-Sana, and P. Lindstrom. Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics. *EEE isuali ation ourse Notes*. 2002.
[26] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour Trees and Small Seed Sets for Isosurface Traversal. *A M ym posium on omputational eometry*, 212–220. 1997.
[27] L. Velho and J. de Miranda Gomes. Digital halftoning with space filling curves. *A M A* , 81–90. 1991.
[28] J. Vitter. External Memory Algorithms and Data Structures: Dealing with MASSIVE Data. *A M omputing urveys*, 209–271, 2001.
[29] J.-M. Wierum. Logarithmic Path-Length in Space-Filling Curves. *1 th anadian onference on omputational eometry*, 22–26. 2002.
[30] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-Oblivious Mesh Layouts. *A M A* , 886–893, 2005.
[31] S.-E. Yoon and D. Manocha. Cache-Efficient Layouts of Bounding Volume Hierarchies. *Eurographics*. 2006. To appear.
[32] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-VDR: Interactive View-Dependent Rendering of Massive Models. *EEE isuali a tion*, 131–138, 2004.

Random-Accessible Compressed Triangle Meshes
Sung-eui Yoon and Peter Lindstrom
Lawrence Livermore National Laboratory, April, 2007
Technical report, UCRL-JRNL-229656

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Random-Accessible Compressed Triangle Meshes

Sung-Eui Yoon, *Member, IEEE*, and Peter Lindstrom, *Member, IEEE*

Abstract—With the exponential growth in size of geometric data, it is becoming increasingly important to make effective use of multilevel caches, limited disk storage, and bandwidth. As a result, recent work in the visualization community has focused either on designing sequential access compression schemes or on producing cache-coherent layouts of (uncompressed) meshes for random access. Unfortunately combining these two strategies has proven challenging as they fundamentally assume conflicting modes of data access.

In this paper, we propose a novel order-preserving compression method that supports transparent random access to compressed triangle meshes. Our decompression method selectively fetches from disk, decodes, and caches in memory requested parts of a mesh. We also provide a general mesh access API for seamless mesh traversal and incidence queries. While the method imposes no particular mesh layout, it is particularly suitable for cache-oblivious layouts, which minimize the number of decompression I/O requests and provide high cache utilization during access to decompressed, in-memory portions of the mesh. Moreover, the transparency of our scheme enables improved performance without the need for application code changes. We achieve compression rates on the order of 29:1 and significantly improved I/O performance due to reduced data transfer. Using two common applications as benchmarks, we demonstrate 2–6 times overall speedup compared to using uncompressed meshes.

Index Terms—Mesh compression, random access, cache-coherent layouts, mesh data structures, external memory algorithms.

1 Introduction

Among the key challenges in visualization is how to effectively manage and display large geometric data sets from scientific simulation, computer-aided design, and remote sensing. Today's unstructured meshes measure hundreds of millions of elements and require gigabytes of storage, often greatly exceeding available memory and rendering resources.

A compounding factor to this problem is the increasing mismatch between processing performance and the rate at which data can be fed to the CPU and GPU, which is limited by latency and bandwidth [36]. As a result, multilevel caching schemes are commonly employed, with successively smaller but faster caches that provide reduced latency. Such caching schemes are effective as long as there is a reasonably close match between data organization and access patterns. Since access patterns on meshes are usually localized, recent work in the visualization community has focused on organizing meshes and other data in cache-coherent order [4, 9, 14, 37, 38].

Data compression is a complementary approach to reducing bandwidth requirements. Whereas mesh compression has traditionally been used to reduce on-disk storage or transmission time over slow networks, recent work has explored the possibility of trading underutilized computing power for higher effective disk bandwidth through on-line compression [5, 24].

For large data sets, the access pattern of the application also significantly influences its performance. The concept of *windowed stream processing* was recently proposed for I/O-efficient access to large compressed meshes [16]. However this approach requires restructuring the computation to match the data layout, which is not always possible or even desirable, e.g. when direct access to small subsets of the data is needed. The main competing approach is to keep a raw on-disk mesh data structure that supports random access [7, 8, 15, 23, 30, 38]. However such data structures usually require a significant amount of disk space: up to 40 times more space than compressed meshes [16]. The added bandwidth requirements of such verbose representations often negate the benefits of organizing the meshes to support random access. To address these issues, new mesh compression schemes are needed that support random access to data layouts suitable for the anticipated access pattern.

Unfortunately, combining compression with coherent data layout is nontrivial. Conventional mesh compression schemes [1, 27, 32] maximize the compression ratio by reordering the data as a canonical permutation, which destroys any layout designed for cache coherence. Streaming mesh compression [17] avoids such reordering by also encoding the layout, but restricts decompression to sequential access: to access an element late in the stream, the entire stream up to that element must be decoded, which can be prohibitive for large files. Recognizing this shortcoming, new schemes have emerged that support selective access to small pieces of the compressed mesh [6, 20]. However, these methods are mainly designed for rendering applications; they do not preserve the mesh layout nor provide a general mesh access mechanism for various geometric applications.

Main Results: In this paper we present a new order-preserving triangle mesh compression algorithm that supports random access to the underlying compressed meshes. Our method selectively fetches, decompresses, and caches the requested parts of the mesh in main memory, possibly after paging out data not recently accessed. The compressor preserves the original, possibly cache-coherent triangle layout, and hence allows optimizing the layout for different modes of access—even for sequential streaming computations. Our decompressor provides direct access to individual vertices and triangles via their global index in their respective layouts, and exposes to the visualization application a conventional mesh data structure API for transparent access to individual mesh elements and their neighborhoods. Although we do not maintain the entire mesh and full neighboring information in main memory, we ensure that correct connectivity is constructed for all mesh elements requested by the application. Using layouts with good locality, we achieve compression ratios approaching 30:1 and speedups as high as 6:1 in out-of-core visualization applications, compared to accessing the same uncompressed external memory data structure. In particular, we find that cache-oblivious mesh layouts [37, 38] result in good compression, high disk cache utilization and thus a small memory footprint, high decompression throughput, and good locality for lower-level in-memory caching.

2 Related Work

In this section we review related work on general mesh compression methods and other compression techniques that support random access for various data representations, including video and meshes.

2.1 Mesh compression

Mesh compression has been well researched over the last decade and excellent surveys are available [2, 12]. At a high level, connectivity compressors may be classified as vertex-based (or valence-based) [1, 32], edge-based [18, 22], or face-based [13, 27], depending

• *The authors are with the Lawrence Livermore National Laboratory, E-mail: {sungeui, pl}@llnl.gov.*

on the mesh element type that drives the compression. Our order-preserving method belongs to the class of face-based compressors.

Most previous mesh compression schemes were mainly designed to achieve maximum compression as they were targeted for archival use or transmission. They achieved this goal by encoding vertices, edges, and faces in a particular order agreed upon by encoder and decoder such that the mesh layout itself would not have to be transmitted. Because many applications are oblivious to the ordering of mesh elements, such reordering is often acceptable.

Recently, Isenburg et al. [17] introduced a streaming compression scheme for triangle meshes built on top of their streaming mesh representation [16]. This compression method efficiently handles massive models by directly encoding mesh elements in the order in which they arrive, which obviates having to first create a complete uncompressed mesh data structure to support traversal of the mesh in the designated order. Because the input order may be locally incoherent and not easily compressible, they allow reordering mesh elements within a small window to improve compression. Our work is built on top of this streaming compression method in order to both preserve the input order of triangles and to achieve high compression and decompression throughput. Note that streaming and order-preserving compression are not synonymous (neither implies the other), and it is primarily the latter quality that we desire.

2.2 Random Accessibility

Most prior approaches to mesh compression do not directly provide random access to the compressed data. To access a particular element, the mesh must first be sequentially decompressed to an uncompressed format (e.g. an indexed mesh format like PLY, or a mesh data structure such as half-edge) that supports random access.

Multimedia compression: Random access is one of the key components of the MPEG video compression format that allows users to browse video in a non-sequential fashion [10]. This is achieved by periodically inserting "intra pictures" as access points in the compressed stream, which allows bootstrapping the decompressor. An intra picture is compressed in isolation from other frames, and subsequent frames are compressed by predicting the motion in between these intra pictures. Similar coding schemes exist for audio compression.

Mesh and multi-resolution compression: Recently, Choe et al. [6] proposed a random-accessible mesh compression technique. They perform a chartification of the mesh and compress each chart separately. Chart boundaries receive special treatment so as to avoid duplicate encoding. Random access to the compressed mesh is achieved by independently decompressing a requested chart, i.e. without having to decompress the whole mesh. There have been a few multi-resolution compression methods that support random access. Gobbetti et al. [11] proposed a compressed adaptive mesh representation of regular grids for terrain rendering. They decompose the regular grid into a set of chunks and apply wavelet-based lossy compression to each chunk. Kim et al. [20] introduced a random access compression technique for general multi-resolution triangle meshes based on their earlier multi-resolution data structure [21].

Although these techniques provide random access to compressed meshes, they are mainly targeted for selective access and refinement in rendering applications, and do not provide a general mesh traversal mechanism. On the other hand, our method supports transparent random access and provides the connectivity information needed by many mesh processing applications.

Cache coherence: Cache-oblivious layouts of polygonal meshes and bounding volume hierarchies have recently been introduced [37–39]. Contrary to cache-aware layouts, e.g. [14], a cache-oblivious layout is not optimized for a particular size cache, but exhibits good average-case performance across multiple cache levels and access patterns. By maintaining cache-oblivious layouts of triangles and vertices in our compressed meshes, we achieve high cache utilization both to compressed data fetched from disk and to uncompressed in-memory data. We also find that cache-oblivious layouts compress well using our order-preserving method, and result in improved application performance compared to other layouts, as we will see in Sec. 6.

3 Overview

In this section we briefly discuss some of the challenges of dealing with massive models and present an overview of our approach.

3.1 Dealing with massive models

Applications such as isosurface extraction and geodesic distance computation require random access to geometry and connectivity information. Such applications access vertices and triangles at runtime in an order that generally differs from the order in which the mesh is stored. For large meshes, the amount of geometry and connectivity information accessed may approach giga- or even terabytes of data. As a consequence, large meshes are usually stored on disk, or may even be fetched over a network. Since accessing remote data in an arbitrary order can be very expensive, data access time often becomes the major computational bottleneck in geometric applications.

Out-of-core data structures and algorithms: There have been extensive research efforts to design out-of-core data structures and algorithms to handle massive models larger than main memory [31]. These techniques aim at loading only the data necessary to perform local computations, and at minimizing the number of I/O accesses. However, as the gap between processing time and data access time increases, the time spent loading even the necessary data on demand becomes very expensive.

Cache coherence: Since cache misses in various memory levels (e.g., L1/L2 and main memory) are quite expensive compared to the computational processing time, research has focused on reorganizing the data access pattern of applications (e.g. [3, 16, 34]) and on computing data layouts (e.g. [37–39]) to minimize the number of cache misses for coherent but unspecified access patterns. In particular, data layout optimization can achieve high cache coherence without having to modify the algorithm or access pattern of the target application, whereas computational reordering usually requires complete algorithm and data structure re-design.

3.2 Our Approach

We propose a novel compression and decompression method that supports transparent and random access to compressed meshes suitable for many geometric applications. Whereas we support truly "random" access to any element of the mesh, we exploit the fact that most geometric applications access the mesh in a spatially coherent manner, e.g., by walking the mesh from an element to one of its neighbors. However, we neither assume nor impose any particular access pattern. At a high level, our method has two major components: (1) cluster-based order-preserving mesh compression, and (2) a runtime decompression framework that transparently supports random access.

Cluster-based order-preserving compression: We compress a mesh by sequentially accessing and grouping triangles and vertices in the order they appear in the input mesh. Although our method does not require a specific layout of a mesh, we propose to use cache-oblivious layouts since they have exhibited superior cache utilization in a number of applications [37]. We also find that these cache-oblivious layouts result in the best compression and runtime performance.

In order to provide random access to the compressed mesh, we group vertices and triangles into a set of *clusters*. Each cluster consists of a fixed number of consecutive triangles (e.g. a few thousand) as well as the vertices first referenced by those triangles. The beginnings of these clusters serve as access points in our method, and each atomic I/O request operates at cluster granularity.

Runtime decompression framework: Our decompression framework provides efficient, but transparent and random access to applications through a common mesh access API that to the application appears as though the entire mesh were memory resident. Because we provide a complete system for loading, decompressing, caching, and paging-out of data, applications benefit directly from the

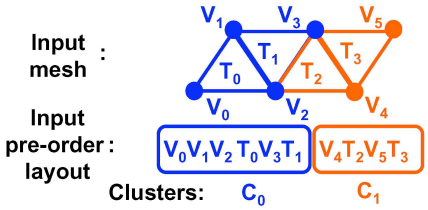


Fig. 1. **Clusters in Vertex-compact Pre-order Layout:** The interleaved sequence of vertices and triangles is *pre-order vertex-compact* [16]. V_i denotes the geometry of the ith vertex; T_i denotes the three vertex indices of the ith triangle. The blue and red boxes indicate the decomposition of the mesh into clusters consisting of two consecutive triangles.

fast I/O and reduced memory footprint provided by our decompressor without having to make any application code changes. We assume that applications access mesh vertices and triangles via an index through our mesh access APIs. When an element is requested, we efficiently locate the cluster containing it, decompress the cluster, and store the uncompressed data in main memory. As the compressed stream does not explicitly encode full connectivity information, we dynamically derive such data on the fly during decompression and link together adjacent in-memory elements in the mesh.

4 Triangle Mesh Compression

In this section we describe our cluster-based order-preserving compression method. We first review the streaming, sequential access compression scheme that our order-preserving method is built upon. Then, we describe the extensions necessary to support random access.

4.1 Streaming Mesh Compression

Isenburg and Lindstrom [17] proposed a streaming compression method for triangle meshes represented in a streaming format [16]. This method sequentially compresses an input mesh of nearly arbitrary size in an I/O efficient manner. There are two major components of this method that our compressor also utilizes.

Input data format: The streaming mesh compressor operates on *vertex-compact* layouts in *pre-order* format (Fig. 1). In a pre-order format, each vertex must appear in the stream before all triangles that reference it. If the layout is also vertex-compact, the first triangle that follows a vertex is guaranteed to reference that vertex, and hence this reference need not be explicitly coded. Vertex-compactness ensures that vertex and triangle layouts are interleaved, and that vertices referenced by a triangle are buffered earlier than they are necessary. The requirement that the input be vertex-compact and pre-order is not particularly restrictive as all triangle layouts have a "compatible" (though not unique) vertex-compact pre-order vertex layout. Note that all face-based compressors naturally produce vertex-compact pre-order layouts.

Finalization: The other key feature of the streaming mesh compressor is the use of *finalization* information. Finalization of a vertex indicates that the last reference to the vertex (by a triangle) has occurred and that no subsequent triangles in the stream will reference it. Hence, during compression we may safely limit references to the set of *active* vertices that have been introduced but not yet finalized. Typically the active vertices are only a small fraction of all vertices, which aids in efficient coding of vertex references. Finalization is usually known to mesh writing applications, and can easily be incorporated with most mesh formats. See [16] for how to compute such information in case it is not readily available.

Our random access compression method uses the pre-order format as input and preserves the order during compression. It also exploits and encodes finalization information. Because of this, our compressed meshes can be used also for streaming computations with little overhead, although our method is mainly designed for random data access.

4.2 Cluster-based Order-Preserving Compression

Our compression method reads intermixed sequences of vertices and triangles in a pre-order streaming file format with finalization information. During compression, we implicitly decompose vertices and triangles into a set of *clusters* (Fig. 1). We define a cluster as a fixed number of consecutive triangles (e.g., 4K triangles). All vertices that are introduced by a triangle in a cluster C are also assigned to C. As a consequence, although each cluster has a fixed number of triangles, the the number of vertices per cluster may not be same (though the variation in vertex count is usually low).

We compress the triangles and vertices assigned to a cluster only based on information collected from the cluster, not from any other clusters. Therefore, at runtime we can decompress the cluster independently without decompressing any other clusters, which provides random access to the mesh at the granularity of clusters. Note that not all the geometry of vertices referenced by the triangles in a cluster reside in the same cluster, nor are all triangles incident on a vertex stored in the same cluster. We will resolve this "stitching" problem in our runtime decompression framework.

Terminology: For a mesh element e such as a vertex or a triangle, we represent its index as $Idx(e)$. Let $C(e)$ denote the cluster containing e, and let C_i indicate the cluster whose index is i, with C_0 being the first cluster. If a triangle of C_j references a vertex stored in C_i, we say that C_j references C_i. Let $R_{dest}(C)$ be the set of (destination) clusters referenced by the cluster C and $R_{src}(C)$ be the set of (source) clusters referencing C. Whenever C_j references C_i, we have: $C_i \in R_{dest}(C_j)$ and $C_j \in R_{src}(C_i)$.

Lemmas related to Clusters: We highlight the following lemmas, which are easily derived from our definition of clusters given vertex-compact pre-order input. These lemmas will be used to show the correctness of our method later.

Lemma 1: Existence of Triangle: For a vertex v introduced in a cluster $C(v)$, at least one triangle contained in $C(v)$ references v.

Lemma 2: Clusters of Adjacent Triangles: The triangles incident on a vertex v are either in the same cluster, $C(v)$, or in $R_{src}(C(v))$.

4.3 Encoding Compression Operators

For each cluster, our compressor sequentially reads vertices and triangles from the pre-order layout of a mesh. For each triangle t, we first determine the compression operator associated with t, which tells how t is connected to the set of already compressed triangles within the cluster.

As in [17], we use five different compression operators (or configurations): START, ADD, JOIN, FILL, and END. START indicates that t shares no edge with the already compressed triangles in the cluster; in ADD and JOIN, there is one shared edge; in FILL and END there are two respectively three shared edges. For example, the sequence of compression operators for the mesh shown in Fig. 1 is "START, ADD, ADD, ADD." For START, ADD, and JOIN cases, we also determine how many new vertices the triangle t introduces and encode their geometry. The number of introduced vertices ranges from 0 to 3 for START, equals 1 for ADD, and 0 for JOIN. These compression operators can be easily determined by maintaining a *half-edge* data structure. For each coded triangle, three half-edges are created. Once a vertex is finalized, the half-edges incident on the vertex may be deallocated.

Compression side: We compute these sequences of compression operators based on the half-edge data maintained from the first triangle of the input to the current triangle under the compression. This means that we do not deallocate the existing half-edge data when we transit from one cluster to the next during compression. The main reason for this is to avoid any duplicate storing of vertex geometry in the compressed mesh, since otherwise we would not know whether a vertex first referenced in a cluster was introduced here or by a triangle in some earlier encoded cluster. We therefore encode the compression operator in the context of global information of all the encoded triangles and vertices.

Decompression side: On the decompression side, however, we do not need to maintain global information of all the encoded triangles and vertices. Given a decoded compression operator, we can deduce the vertex indices associated with the triangle. Some of these vertex indices may refer to geometry information stored in another cluster. If an

(a) Cache-oblivious mesh layout (b) Z-order (c) Spectral layout (d) Depth-first layout (e) Breadth-first layout

Fig. 2. **Clusters with Layouts**: This figure highlights clusters induced by different triangle layouts of a simplified Puget Sound terrain consisting of 512K triangles. Each cluster, here outlined by black boundaries, is defined by 8K consecutive triangles. The cluster colors smoothly vary with the sequential layout from red to yellow to green, and the brightness alternates between each consecutive pair of clusters. The cache-oblivious mesh layout has high spatial coherence, leading to well-shaped clusters with short boundaries and few inter-cluster references. As a result, it achieves the highest compression ratio and best runtime performance on our benchmarks.

application requires geometry (as opposed to only connectivity) information for such vertices, our runtime framework will determine which cluster has that information and decode and return the requested data to the application. When on the other hand geometry is not requested, this lazy decoding scheme saves time.

4.4 Encoding Mesh Elements

Once the compression operator for a triangle is encoded, we need to encode the vertices, and possibly their geometry, referenced by the triangle. For each of these vertices, there are two cases: (1) the vertex is referenced for the first time and its geometry is encoded, or (2) the vertex has already been introduced.

If a vertex, v, is introduced by a triangle t, we encode v's geometry and attributes (e.g., color). Note that in this case we do not need to encode its vertex index since it necessarily must equal the current global vertex count. This vertex count is made available to the decompressor by storing with each cluster C the global index of C's first vertex, which is maintained incrementally during compression.

In the other case, when vertex v was introduced in an earlier cluster, we are interested only in encoding its vertex index $Idx(v)$ since v's geometry has already been encoded. In order to effectively encode its index, we make use of three layers of buffering: (1) a cache holding the three vertex indices of the previous triangle in the current cluster, $C(v)$, (2) the active (unfinalized) vertices of $C(v)$, and (3) the vertices among the clusters $R_{dest}(C(v))$ referenced by $C(v)$. The first and second layers are for *in-cluster* vertices stored in the current cluster, and the third layer is for *out-of-cluster* vertices stored in some other cluster.

If vertex v was also referenced by the previous triangle t_p, we encode which of the three vertices in the cache v corresponds to. Otherwise, we check whether v is an active vertex of the current cluster, and if so encode its position in a dynamic vector (containing active vertices) maintained by both encoder and decoder. Note that the set of active in-cluster vertices is usually much larger than the three vertices stored in the cache, but is also smaller than the set of vertices stored in the current cluster.

Finally, if v is not among the active vertices, we conclude that it is in another cluster among the set $R_{dest}(C(v))$. In this case, we decompose its index into a pair index (i,o) where i is the global cluster index for C_i and o is an offset within C_i. Instead of directly encoding the pair (i,o), we map the global index i into a local index j within the set $R_{dest}(C(v))$. This is beneficial as the number of clusters in $R_{dest}(C(v))$ is usually much smaller, e.g. 4 on average for cache-oblivious layouts, than the total number of clusters.

After finishing compressing the mesh, we have accumulated cluster-specific information, such as the cluster's position in the compressed file, its first vertex index, and the variable-size sets R_{dest} and R_{src}, that must be written as header information. This information can be stored in a separate file, or may be appended together with a file offset to the end of the compressed file. The decompressor is initialized with this header information to allow any cluster to be decompressed.

Memory requirement and time complexity: The memory usage of the data structures needed to perform the operations described above is very small because the data is limited to the set of active vertices and the elements of the current cluster (a typical cluster size is on the order of a few thousand triangles). The time complexity of encoding end decoding a triangle is constant. This is made possible by using a hash table of active vertices to map a global index $Idx(v)$ to the cluster-offset pair (i,o).

Half-edge based coding: We further improve the compression ratio by encoding some of the vertex indices based on existing half-edge information around vertices, as proposed in [17]. We encode the index of an in-cluster vertex by specifying which of the set of half-edges it is associated with. For example, when the compression operator is ADD, we may encode two active vertices by the single half-edge that joins them, and to which the triangle being encoded is attached. Recall that we deallocate all the existing half-edge information when we encounter a new cluster in the decompressor. Therefore, the compressor, too, needs to consider only those half-edges created in the current cluster. This half-edge based coding requires a small amount of computational overhead, such as traversing half-edges for the vertices of a triangle. However, we can achieve a higher compression ratio since the number of half-edges around a vertex is typically small (e.g., 6).

Geometry prediction: We use the parallelogram rule [32] to predict the geometry of a newly introduced vertex v_3 of a triangle t. To perform the parallelogram prediction in an ADD operation, we require geometry information for the three vertices of an adjacent triangle that was already compressed, and that shares v_1 and v_2 with t. However, some of these vertices may be stored in another cluster and, thus, their geometry information may not be available when we compress or decompress the current cluster. In this case, we simply use v_1 or v_2 as prediction, if available, or otherwise the third "opposite" vertex. When no nearby vertex is available, we use the bounding box center as prediction.

Arithmetic coding: We use context-based arithmetic coding to encode compression operators, vertex indices, and geometry. Particularly, we use as context the previous compression operator to encode the following operator. We also re-initialize all the probability tables with a uniform distribution as we begin compressing a new cluster. This enables us to independently decompress each cluster. However, this periodic initialization may hurt the compression ratio. To minimize the impact of initialization, we employ a fast adaptive probability modeler [29].

5 Runtime Mesh Access Framework

In this section we present our runtime decompression and caching method that provides transparent random access for a wide variety of applications.

5.1 In-Core Mesh Representation

When the application requests geometry or connectivity information of a mesh element, our underlying decompression framework determines the cluster containing the requested mesh element and decompresses the cluster into an in-core partial mesh representation.

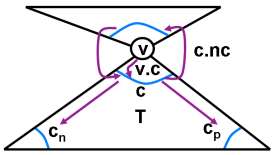


Fig. 3. **Corner Representation:** The figure illustrates our corner representation. Each triangle has three corners (shown as blue curves). Given a corner c of a triangle t, we provide access to the next (c_n) or previous (c_p) corner in t, as well as the next corner (in no particular order), $c.nc$, incident to vertex v. These corner pointers form circular linked lists around vertices and triangles. From v, we provide access to one of its incident corners, $v.c$, and from each corner c is a pointer, $c.v$, to the corresponding vertex v. Similar pointers implicitly exist between corners and triangles. All these operations are supported by our APIs, described in Sec. 5.2.

To support general mesh access mechanisms for various applications, we represent our decompressed in-core mesh in a corner table data structure, loosely based on the one proposed by Rossignac [27]. A *corner* is assigned to each wedge of a triangle and conceptually connects a vertex v and a triangle t incident to v (see Fig. 3). Detailed vertex and triangle data structures are shown in Listing 1.

```
struct {
  struct {
    float X, Y, Z; // Geometry of the vertex
  } Coords;
  unsigned c;      // A corner incident on the vertex
} Vertex;

struct {
  struct {
    unsigned v;    // Vertex index
    unsigned nc;   // Next corner incident on the vertex
  } Corner [3];
} Triangle;
```

Listing 1. **In-core Structure of Vertex and Triangle:** These two structures represent in-core geometry and connectivity information based on a corner table.

Each vertex v has its geometry information and a corner index $v.c$ indicating one of corners of the triangles incident to v. Each triangle is represented as three corners. A corner c consists of an index $c.v$ to a vertex v that c is incident to, and a next corner index $c.nc$ that points to a corner of another triangle incident to v. Note that we can compute all the incident triangles to a vertex v by traversing all the next corner indices $c.nc$ incident to v. This corner representation is illustrated in Fig. 3.

For our in-core mesh representation, we conceptually create two separate contiguous corner table arrays of vertices and triangles for the whole mesh. Suppose that a triangle t has index i. Then, the three corners of t have indices $3i$, $3i+1$, and $3i+2$ given our virtually contiguous corner table array of triangles.

Construction: The corner table information can be incrementally computed in a constant time per triangle as we sequentially decompress the global vertex indices of each triangle. For each vertex of a triangle, we compute a corresponding corner index from the triangle index. Then, we connect the corner of the triangle into the circular corner list maintained with the vertex. If there is no incident corner to the vertex, we initialize its corner pointer with the new corner.

5.2 Mesh Access API

We provide the following atomic mesh access API to support transparent and random access to the compressed mesh:

1. Coords **GetVertex** (unsigned v_{Idx}): Given a vertex index, v_{Idx}, this function returns the requested vertex coordinates (see Listing 1).
2. Triangle **GetTriangle** (unsigned t_{Idx}): Similar to GetVertex but for triangles.
3. unsigned **GetVertexCorner** (unsigned v_{Idx}): Given a vertex index, v_{Idx} of a vertex v, the function returns an index, $v.c$, to a corner incident to v.

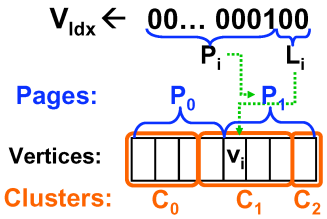


Fig. 4. **Page-based Data Access:** The figure illustrates an overview of our data access mechanism where page size is 4 mesh elements. Given a vertex index, i, page and local indices P_i and L_i are computed by simple bit operations. Then, we load clusters, C_1 and C_2, overlapped with the requested page P_1 and return the requested the vertex v_i to the application.

4. unsigned **GetNextCorner** (unsigned c_{Idx}): Given a corner index, c_{Idx}, of a corner c in a triangle, provide the next corner, $c.nc$, of another triangle incident to the vertex of c.
5. unsigned **GetTriIdxfromCorner** (unsigned c_{Idx}: Compute a triangle index t_{Idx} from a corner index c_{Idx}. Given our virtually contiguous triangle array, t_{Idx} is computed as $c/3$.
6. unsigned **GetNextCornerInTriangle** (unsigned c_{Idx}): This function returns the next corner index c_n from c_{Idx} in the same triangle. c_n can be computed using modular arithmetic.
7. unsigned **GetCornerVertex** (unsigned c_{Idx}): Given a corner index c_{Idx} of a corner c, return the index of the vertex incident to the corner. To do that, we first access the triangle t of the corner c by using **GetTriIdxfromCorner()**. Then, the requested vertex is stored in the kth corner of t, where $k = c_{Idx} \mod 3$.

Based on these atomic APIs, we can also provide more various and advanced mesh traversal APIs. For example, if we want to compute all the triangles incident to a vertex v, we can compute those triangles by sequentially accessing all the corners (by calling "GetNextCorner ()") starting from the corner $v.c$ of the vertex computed from "GetVertexCorner ()". Moreover, we implement our benchmark application, iso-contour extraction, with our mesh access APIs.

5.3 Page-based Data Access

Given a mesh element index including vertex, triangle, and corner index, we have to first compute a cluster containing the requested mesh element. Since each cluster can have different number of vertices although clusters can have same number of triangles, computing a cluster index given a vertex index can take logarithmic time complexity.

To provide fast decoding mechanism from a vertex index to a cluster index, we propose to use another layer of *page table* on top of clusters and mesh elements as shown in Fig. 4. We decompose virtually contiguous two arrays of vertices and triangles to a set of pages, each of which holds power-of-two mesh elements (e.g., 1k vertices). Each page has two data: (1) cluster indices whose clusters overlap with the page and (2) a flag variable indicating the status of the page. The flag of each page consists of 2bit and can have four states in two categories: (1) mesh elements of all the clusters associated the page are "Loaded or Not loaded" and (2) connectivity is "Partially or Fully connected". Also, each cluster has same flag variable, which can have these same four states with respect to the cluster. All the flags of pages and clusters are initially set as "Not loaded".

The main benefit of having this page structure on top of layers of clusters and mesh elements is that we can compute page index from mesh element index based on a few bit operations since each page has power of two mesh elements. Then, since each page has cluster indices whose clusters overlap with the page, we can compute a superset of clusters, one of which contains the requested mesh element. Also, to have the small average number (e.g., very close to 1) of clusters overlapped with each cluster, we set the size of pages (e.g., 1K triangles) to be smaller than cluster size.

Memory management: We also provide memory management based on page tables to deal with massive models whose uncompressed data cannot fit into main memory. If an application set an maximum allowance on in-core main memory, we only maintain pages that

can fit into the maximum allowance. Also, when we need a new page and there is no available main memory, we have to unload a page and its related clusters. To perform this task, we also maintain LRU-list of pages. Please note that maintaining LRU-list every time to access a mesh element can be very expensive. Instead, we update LRU-list only when a page is newly loaded.

5.4 Accessing Mesh Elements

Suppose that an application calls "GetVertex $(\cdot)$" requiring geometry of a vertex or "GetTriangle $(\cdot)$" requiring vertex indices of a triangle by passing a mesh element index (e.g, vertex index or triangle index). We first compute a page index given a mesh element index. Then, if the page has "Not loaded" state, we make sure that all the clusters associated with the page are loaded by loading unloaded clusters and decompressing them into our in-core mesh representations. Then, we set the flag of the page as "Loaded". If we access a page with "Loaded" state next time, we do not need to load any of clusters of the page and directly return requested data from our in-core mesh representations. Therefore, we can very efficiently access uncompressed data from in-core mesh representation.

5.5 On-demand Connectivity Construction

We dynamically construct corner information to provide mesh traversal mechanism requested through our atomic APIs including "GetVertexCorner$(\cdot)$" and "GetNextCorner$(\cdot)$". As explained earlier, there are two states related to connectivity information of a cluster or a page and we will explain them in detail.

Partial connectivity: Suppose that an application call "GetVertexCorner$(\cdot)$" by passing a vertex index. We first compute a page index from the requested vertex index. Then, for each cluster associated with the page, we sequentially access each corner of triangles stored in our in-core mesh representations and attempt to connect the corner to its incident vertex. However, some of those vertices may not be already loaded; therefore, we cannot connect them. In this case, we put corners indicating out-cluster vertices into into *SkippedCornerList* of the cluster. Then, we set the state of a cluster to be "Partially connected". After we process all the clusters of the page, it is guaranteed that the requested vertex will have a corner information due to Lemma of incident triangle introduced in Sec. 4.2.

Full connectivity: Now suppose that an application call "GetNextCorner $(\cdot)$" by passing a corner index in order to compute one of adjacent triangles given a vertex v incident to the requested corner. To provide a correct result to the application, all the corners incident to the vertex v have to be loaded and connected into a virtual circular corner list of the vertex v. We achieve this property by performing the following three steps. We first load clusters of $R_{src}(C(v))$, clusters referring to a cluster $C(v)$ and build partial connectivity for $R_{src}(C(v))$. Then, according to Lemma 2 introduced in Sec. 4.2, it is guaranteed that all the adjacent triangles to the vertex v are loaded and decompressed into our in-core mesh representation. Second, we access each corner from *SkippedCornerList* of the cluster $C(v)$ and connect it if its incident vertex is stored in one of clusters of $R_{src}(C(v))$. If a corner is connected, then, the corner is removed from *SkippedCornerList*. Finally, we set the flag of $C(v)$ as "Fully connected". At next time when we access a vertex or a corner, its page or cluster has a flag of "Fully connected", then, we simply return the requested corner from our in-core mesh representation.

Unloading a cluster: When we unload a cluster to give memory space to a newly accessed page, we have to put extra care to guarantee correct connectivity information. If we unload a cluster C, application may not get correct connectivity information when it accesses connectivity information of clusters of $R_{dest}(C)$ since vertices stored in $R_{dest}(C)$ can be referred by some triangles of the cluster C. Therefore, we set the flag of clusters of $R_{dest}(C)$ as "Partially connected".

5.6 Cache coherent layout

We also would like to provide high cache-coherence during accessing in-core mesh elements by various geometric applications. In order to achieve this goal, we use cache-oblivious mesh layouts [37, 38] of vertices and triangles of a mesh. To compute such a layout, we first compute a cache-oblivious layout of triangles by using OpenCCL library [40]. Then, we compute an pre-order layout of vertices and triangles by traversing triangles in the computed order. Although we do not separately compute cache-oblivious layout of vertices, the vertex layout induced from the cache-oblivious layout of triangles is empirically observed to maintain good quality of cache-coherence [37]. An example of clusters computed from the cache-oblivious mesh layout of a model is shown in Fig. 2.

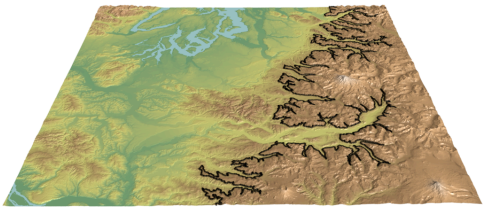


Fig. 5. **Puget Sound Iso-Contour:** This image shows a contour line (in black) extracted from an unstructured terrain model of the Puget Sound. The terrain is simplified down to 143 million triangles. We extracted an iso-contour (passing 286K triangles) of the level set at 720 meters of elevation. Our compression method achieves a compression ratio of 29 to 1 compared to uncompressed original mesh. Moreover, we are able to achieve runtime performance of the iso-contouring 2.4 times on average compared to using uncompressed meshes.

6 Results

In this section we highlight results obtained by our method. To demonstrate the benefits of our method, we implement two applications, iso-contour extraction and a layout re-computation on compressed meshes by using our general mesh access APIs shown in Sec. 5.2. We choose these two applications since both of them are performed by traversing meshes in a random order different from the stored order in an original layout of a mesh. Iso-contour extraction typically accesses a small region of the mesh. On the other hand, for the layout re-computation, we have to traverse the entire mesh.

We have implemented our compressor, decompressor, and our applications on a Intel Pentium 4 mobile laptop machine with a 2.1Ghz CPU and 2GB main memory. We also implemented our compression method to directly work on a streaming mesh format [16]. Converting any file format to streaming mesh file format with finalization information is very simple and can be very efficiently computed [16].

6.1 Compression Results

We test our compression method with different benchmark models. These include a large terrain model of Puget Sound area consisting of 134 million triangles (Fig. 5), DOE turbulence surface model consisting 102 million triangles, and St. Matthew model consisting of 128 million triangles. The details of these models are shown in Table 1. We also quantize each geometry of vertices into 16bit given a bounding volume of a model.

We measure the processing time of our compressor. Our compressor can process 380K triangles per sec on average on our mobile laptop. For example, it takes 6 min to compress Puget Sound terrain model consisting of 134 million triangles.

We also compare the file sizes of our compressed meshes with those of original uncompressed meshes, which is stored in structures shown in Listing 1. We are able to achieve a compression ratio of up to 29 to 1 compared to that of original mesh. Also, our method spend 10.75 bit per triangle (bpt), 14.32 bpt, 11.57 bpt for Puget Sound model, DOE model and St. Matthew model respectively. Therefore, if a cluster has 4K triangles–the size of un-compressed in-core data of the cluster is 150KB, the size of a compressed file of the cluster in disk is 5.3KB on average in Puget Sound model.

6.2 Iso-contouring

The problem of extracting an iso-contour given a specific iso-value from an unstructured dataset frequently arises in geographic information systems and scientific visualization. Many efficient iso-contour

Model	Triangles (M)	Uncompressed mesh size (MB)	Compressed mesh size (MB)	Compression ratio
Terrain model	134	5243	178	29.4
DOE model	102	4013	177	22.6
St. Matthew	128	4899	179	27.3

Table 1. **Benchmark Models:** Model complexity, file sizes of uncompressed and compressed meshes with cluster size of 4K triangles, and compression ratios are shown.

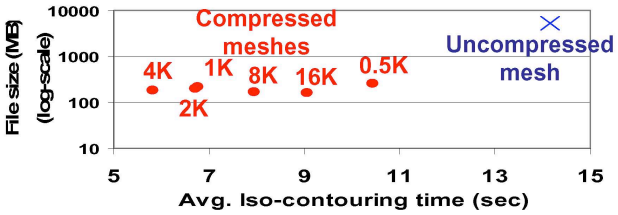


Fig. 6. **File sizes and Iso-contouring time VS. Cluster sizes:** The figure shows the iso-contouring time (shown in X-axis with a log scale) and the file sizes (shown in Y-axis) as a function of cluster size with our compression method. Also, the figure shows these values of original uncompressed Puget Sound model. We are able to achieve a compression ratio of 29 to 1 and 2.4 time performance improvement on average in iso-contouring application when setting cluster to have 4K triangles. Also, we are able to achieve similar performance when we set cluster sizes of 2K and $1K$.

extraction methods employ seed sets [33] to grow an iso-contour by traversing only the triangles intersecting the iso-contour. The running time of such an algorithm is typically dominated by the traversal of the triangles intersecting the contour. We efficiently extract an iso-contour from a seed triangle by traversing the iso-contour in a depth-first order, thereby accessing a mesh in a reasonably cache-coherent manner, but in a different order from the stored layout in the compressed mesh.

We measure the runtime performance of iso-contouring application during performing 20 iso-contour extractions with different iso-values on the Puget Sound terrain model. We use a cache-oblivious mesh layout for the input layout of the terrain model and set cluster size to be 4K triangles. In this configuration, we are able to achieve 2.4 times improvement on average and up to 6.4 times improvement in one iso-contour extraction compared to running the application with the uncompressed meshes stored in the same cache-oblivious layout. The main performance improvement is caused by drastic reductions on expensive disk I/O accesses during traversing the mesh.

Performance VS. Cluster size: We measure the performance of iso-contouring and file sizes of the compressed files as the function of cluster sizes. In general, the file sizes of our compressed meshes are getting smaller as the cluster sizes is getting bigger since there are less likely to encode expensive out-of-cluster vertices. Also, we find that compression with cluster sizes of 4K triangles gives good compression ratio, which is 16% lower than that of our compression method with the maximum cluster size (i.e., is the number of triangles of a mesh). Also, the runtime performance of applications can vary as the function of cluster sizes. In the iso-contouring application, we find that cluster size of 4K triangles shows the best performance among the tested cluster sizes. However, clusters size from 1K to 6K triangles shows reasonably good performance compared to the best runtime performance. More detail results are shown in Fig. 6.

Working set size: We also measure working sets, which are loaded data during iso-contouring, as a granularity of 4KB, which is the page size of the tested system. Working set of our method with cluster size of 4K triangles is 4 times bigger than that of using original uncompressed meshes. The main reason why our approach has higher working set sizes are two folds: (1) our method access mesh elements in the granularity of a cluster by decompressing whole clusters although only one triangle is required by an application and (2) when a cluster C is accessed, our method requires to load all the cluster of $R_{src}(C)$, clusters referring to the cluster C in order to provide correct connectivity information to the application.

Performance with Different Layouts: We also measure iso-contouring time with different layouts of Puget Sound terrain model.

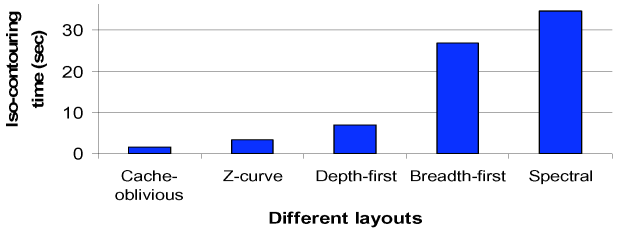


Fig. 7. **Performance VS. Layout:** The figure shows the runtime iso-contouring time (shown in Y-axis) observed by using our compressed meshes with different layouts on Puget Sound model. Cache-oblivious mesh layout shows much better performance than others. Particularly, cache-oblivious mesh layout shows 2times, 4.3 times , 16.7 times, and 21.7 times performance improvement over Z-curve, depth-first, breadth-first, and spectral layouts.

We compare the performance of iso-contouring with cache-oblivious mesh layout (COML) [37, 38], Z-curve [28], depth-first (DFL), breadth-first (BFL), and spectral (SL) layouts [19] . Please note that Z-curve is also cache-oblivious layouts, which work well mainly for regular grids. We are able to observe the best performance with COML, which shows 2 times, 4.3 times, 16.7 times, and 21.7 times performance improvement over Z-curve, DFL, BFL, and SL respectively (see Fig. 7).

We also measure compressed file sizes of different layouts: 178MB with COML, 261MB with Z-curve, 287MB with DFL , 312MB with SL, and 461MB with BFL. The cache-oblivious mesh layout (COML) shows the best compression ratio among the tested layouts. Also, please note that the performance improvement of COML over other layouts are much bigger than that of file sizes. The main reason why COML layout achieves such high runtime performance is due to its spatial coherence between elements in the layout as shown in Fig. 2. Higher spatial coherence is likely to require loading of less clusters given access patterns of applications. To quantitatively verify this, we measure the working set size during iso-contouring with different layouts. During extracting an iso-line consisting of 33K edges, COML requires 118MB working set size. However, Z-curves, DFL, BFL, and SL require 292MB, 196MB, 1.4GB, and 2.4GB respectively.

Comparison with streaming computations: We also compare our method supporting random accesses with streaming computation method [16] working on the streaming compression method [17]. Particularly, we measure the running time of extracting an iso-lines with a certain X-value on Puget Sound terrain model. Therefore, the extracted iso-lines are parallel to X-axis. Since streaming computation method does not allow random accesses on the compressed mesh, we have to access entire mesh by sequentially traversing vertices and triangles in the stored order of the mesh to extract the iso-lines. On the other hand, our compression method can provide random accesses and, thus, can access small regions of the mesh intersecting with the iso-line. Therefore, we are able to observe 45 times performance improvement on this particular iso-contour extraction compared to using streaming compression method.

However, for various geometric processing applications (e.g., smoothing and computing vertex normal) suitable for sequential accesses on meshes, streaming computation framework may perform better than our random access framework. Our compressed meshes also supports very efficient streaming computation since our compression method provides finalization. We also compare the performance of our method during sequentially accessing the whole compressed mesh with that of streaming compression method, which is optimized particularly for the streaming computation. We find that our method has 33% performance overhead over streaming compression method. Since our method efficiently supports both random accesses and streaming accesses on the compressed meshes, we believe that our method has strong advantage over the streaming compression method.

6.3 Layout Re-computation

We implement a layout re-computation method from an input layout of a mesh. Particularly, we traverse the input mesh in a breadth-first order

and compute a breath-first triangle layout of the mesh. The different characteristic of this application over iso-contour extraction is that we have to traverse the entire mesh to compute a breadth-first layout of the mesh. We measure the performance of the application by using both our compressed mesh and un-compressed original mesh of Puget Sound model. Our method shows 6.7 times performance improvement over that of using the un-compressed original mesh Moreover, we are able to observe similar performance improvement with different cluster sizes ranging from 2K to 16K triangles.

7 Conclusion and Future Work

We have proposed a novel compression and decompression method supporting transparent random access on the compressed meshes. In order to efficiently provide mesh traversal mechanism, our method dynamically constructs connectivity information necessary for traversing triangular meshes and provide general mesh traversal APIs. Also, our method can preserve an input layout of a mesh. Particularly, cache-coherent mesh layouts showed highest runtime performance and compression ratio among tested layouts. Moreover, our method enables to improve the performance of applications without modifying application's algorithm and implementation.

Our method works well with our benchmark models and our tested application. However, it has certain limitations. First of all, the performance of our method in a certain application can vary as the function of cluster size. Therefore, there is no guarantee that we improve the performance with any cluster sizes. However, in practice, we have found that cluster sizes ranging from 2K to 8K works well.

There are many interesting avenues for future work. In addition to addressing limitations of our current method, we would like to extend our method to efficiently handle models requiring frequent modifications. Such scenarios include various geometric processing (e.g., simplification and filtering), dealing with dynamic models, and interactive editing. Also, we would like to generalize current method to work with other polygonal representations such as tetrahedron and hexahedron meshes. Also, we believe that our current method works well with parallel computations and would like to apply it to an application running on an parallel environment. Finally, we would like to design random accessible compressed hierarchies to improve the performance of ray tracing and collision detection between massive models.

Acknowledgments

We would like to especially thank Martin Isenburg for sharing his streaming compression mesh codes with us.

References

[1] P. Alliez and M. Desbrun. Valence-Driven Connectivity Encoding for 3D Meshes. *Eurographics'01 Proceedings*, 480–489. 2001.

[2] P. Alliez and C. Gotsman. *Recent advances in compression of 3D meshes*, 3–26. Springer, 2005.

[3] L. Arge, G. Brodal, and R. Fagerberg. Cache Oblivious Data Structures. *Handbook on Data Structures and Applications*, 2004.

[4] A. Bogomjakov and C. Gotsman. Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes. *Graphics Interface'01 Proceedings*, 81–90. 2001.

[5] M. Burtscher and P. Ratanawoabhan. High Throughput Compression of Double-Precision Floating-Point Data. *IEEE Data Compression Conference*, 293–302. 2007.

[6] S. Choe, J. Kim, H. Lee, S. Lee, and H.-P. Seidel. Mesh compression with random accessibility. *Proc. of 5th Israel-Korea Bi-National conference*, 81–86. 2004.

[7] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External Memory Management and Simplification of Huge Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2003.

[8] C. DeCoro and R. Pajarola. XFastMesh: Fast View-Dependent Meshing from External Memory. *Visualization'02 Proceedings*, 363–370. 2002.

[9] P. Diaz-Gutierrez, A. Bhushan, M. Gopi, and R. Pajaola. Single-strips for fast interactive rendering. *The Visual Computer*, 22(6):372–386, 2006.

[10] D. Gall. MPEG: a video compression standard for multimedia applications. *Eurographics Symposium on Geometry Processing*, 46–58. 1991.

[11] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli. C-BDAM - Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering, 2006.

[12] C. Gotsman, S. Gumhold, and L. Kobbelt. *Simplification and Compression of 3D Meshes*, 319–361. Springer, 2002.

[13] S. Gumhold and W. Strasser. Real Time Compression of Triangle Mesh Connectivity. *SIGGRAPH'98 Proceedings*, 133–140. 1998.

[14] H. Hoppe. Optimization of Mesh Locality for Transparent Vertex Caching. *SIGGRAPH 99 Proceedings*, 269–276. 1999.

[15] M. Isenburg and S. Gumhold. Out-of-Core Compression for Gigantic Polygon Meshes. *SIGGRAPH 2003 Proceedings*, 935–942. 2003.

[16] M. Isenburg and P. Lindstrom. Streaming Meshes. *Visualization '05 Proceedings*, 231–238. 2005.

[17] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming Compression of Triangle Meshes. *Eurographics Symposium on Geometry Processing*, 111–118. 2005.

[18] M. Isenburg and J. Snoeyink. Face Fixer: Compressing Polygon Meshes with Properties. *SIGGRAPH'00 Proceedings*, 263–270. 2000.

[19] M. Juvan and B. Mohar. Optimal linear labelings and eigenvalues of graphs. *Discrete Applied Mathematics*, 36(2):153–168, 1992.

[20] J. Kim, S. Choe, and S. Lee. Multiresolution Random Accessible Mesh Compression. *Computer Graphics Forum (Eurographics 2006)*, 25(3):323–332, 2006.

[21] J. Kim and S. Lee. Truly selective refinement of progression meshes. *Proc. Graphics Interface*, 101–110. 2001.

[22] J. Li and C. C. Kuo. A dual graph approach to 3D triangular mesh compression. *Proceedings of ICIP'98*, 891–894. 1998.

[23] P. Lindstrom. Out-of-Core Construction and Visualization of Multiresolution Surfaces. *ACM SIGGRAPH Symposium on Interactive 3D Graphics*, 93–102. 2003.

[24] P. Lindstrom and M. Isenburg. Fast and Efficient Compression of Floating-Point Data. *Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.

[25] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitized message. *Video and Data Recording Conference*. 1979.

[26] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic Coding Revisited. *ACM Transactions on Information Systems*, 16(3):256–294, 1998.

[27] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[28] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[29] M. Schindler. Range Encoder version 1.3, 2000. URL `http://www.compressconsult.com/rangecoder/`.

[30] E. Shaffer and M. Garland. A Multiresolution Representation for Massive Meshes. *IEEE Transaction on Visualization and Computer Graphics*, 11(2):139–148, 2005.

[31] C. Silva, Y.-J. Chiang, W. Correa, J. El-Sana, and P. Lindstrom. Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics. *IEEE Visualization Course Notes*. 2002.

[32] C. Touma and C. Gotsman. Triangle mesh compression. *Graphics Interface'98 Proceedings*, 26–34. 1998.

[33] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour Trees and Small Seed Sets for Isosurface Traversal. *ACM Symposium on Computational Geometry*, 212–220. 1997.

[34] J. Vitter. External Memory Algorithms and Data Structures: Dealing with MASSIVE Data. *ACM Computing Surveys*, 209–271, 2001.

[35] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, 1987.

[36] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[37] S.-E. Yoon and P. Lindstrom. Mesh Layouts for Block-Based Caches. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization)*, 12(5), 2006.

[38] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-Oblivious Mesh Layouts. *ACM SIGGRAPH*, 886–893, 2005.

[39] S.-E. Yoon and D. Manocha. Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum (Eurographics)*, 25:507–516, 2006.

[40] S.-E. Yoon, D. Manocha, P. Lindstrom, and V. Pascucci. OpenCCL, 2005. URL `http://gamma.cs.unc.edu/OpenCCL`.

Ray-Strips: A Compact Mesh Representation for Interactive Ray Tracing

Technical Report TR07-003

Christian Lauterbach*
University of North Carolina at Chapel Hill

Sung-Eui Yoon†
Lawrence Livermore National Laboratories

Dinesh Manocha‡
University of North Carolina at Chapel Hill

Abstract

We present a novel hierarchical representation, Ray-Strips, for interactive ray tracing of complex triangle meshes. Many optimized algorithms for ray tracing store each triangle in the input model explicitly. Instead, a Ray-Strip takes advantage of mesh connectivity for both compact storage as well as efficient traversal and ray intersections. As a result, we considerably reduce the memory overhead of both original model and acceleration structure, thereby making it suitable for representing massive data sets. At the same time, the representation is orthogonal to geometry compression approaches and does not reduce precision. We also present efficient algorithms for single ray and ray packet traversal using Ray-Strips. Furthermore, we demonstrate an additional benefit of our representation: utilizing the SIMD capabilities of current CPUs for incoherent ray packets and single rays. Finally, we demonstrate the benefit of Ray-Strips on models with tens of thousands to tens of millions of triangles. In practice, our approach can reduce the storage overhead of interactive ray tracing algorithms by 5 times.

1 Introduction

Ray tracing has recently emerged as an alternative to rasterization for interactive rendering. Ray tracing has been studied extensively for almost three decades due to its ability to simulate the physical process of light propagation and to generate rendering effects like transparency, shadows and indirect illumination. There has been renewed interest in real-time ray tracing due to the exponential growth of processing power and recent hardware trends of using multiple cores, since ray tracing algorithms are embarrassingly parallel and easily map to multi-core and multi-processor systems.

One of the main benefits of ray tracing is that its asymptotic complexity increases as a logarithmic function of model size, for a given image resolution. This makes ray tracing an attractive approach for rendering large data sets composed of tens or hundreds of millions of primitives. However, the model representation and acceleration data structures used to ray trace such massive models can take tens of giga-bytes. This large storage overhead can affect the performance of out-of-core ray tracers [YLM06]. As a result, interactive ray tracing of massive models remains a challenging problem.

Recent algorithmic improvements such as ray coherence techniques [WBWS01; RSH05] are able to exploit coherent hierarchy traversal behavior of packets or groups of rays and improve the performance of ray tracing. These algorithms have been shown to work well with architectural models, where most visible triangles cover large areas in the image space in terms of number of pixels. As a result, groups of rays maintain high spatial coherence between them in terms of hierarchy traversal behavior. However, these methods may not work well with highly complex and detailed models since coherence decreases with rising model complexity and small triangles. In particular, recent advances in 3D scanning and scientific simulations have greatly increased the model complexity and frequently result in models composed of hundreds of millions of triangles. We need better representations to handle such models.

Most ray tracing algorithms also assume that input models are 'triangle soup', i.e. all triangles are considered unordered and independent. However, many meshes used in computer games, CAD, scanned data, and scientific simulations have well-defined connectivity between the triangles. This property has been exploited to compute a sequential order of triangles and using that order for higher rasterization throughput. This includes representations based on triangle strips or rendering sequences [Dee95; Hop99], which are designed to reduce the memory requirement and transformation cost during rendering triangles. Unfortunately, such compact and efficient representations developed for improving the rasterization performance are not directly applicable to ray tracing. This is mainly caused by the fact that triangle strips or rendering sequences are mainly designed to be accessed sequentially. On the other hand, current ray tracing algorithms access underlying data representations including triangles in a random manner. Therefore, most optimized ray tracing methods store each triangle and its geometry information separately and in practice require a higher memory footprint for ray tracing massive models than comparable rasterizers.

Main Contributions: In this paper, we address the problem of efficiently representing triangle meshes for fast ray tracing. We present a novel mesh and hierarchy representation, *Ray-Strips*, for interactive ray tracing. A Ray-Strip includes a list of vertices, which implicitly represents a list of triangles like a triangle strip. Moreover, our representation also includes a light-weight hierarchy defined on the list. Therefore, our Ray-Strips allows efficient tree traversal as well as efficient ray intersection tests between a ray and the triangles stored in the Ray-Strip. We also present techniques to improve the SIMD utilization for incoherent ray packets based on Ray-Strips. We also apply our methods to different benchmarks including complex scanned and CAD and simulation data sets. The model complexity varies from tens of thousands of triangles to tens of millions.

As compared to prior approaches, our representation gives us the following advantages:

- **Memory efficiency:** Ray-Strips require much less memory for both the hierarchy as well as the geometric primitives by compactly storing both of them. Compared to previous approaches, Ray-Strips are able to reduce memory requirement of running ray tracing by up to 5 times. This can considerably improve the performance of out-of-core ray tracers and makes it possible to handle massive models on machines with limited disk space or memory (e.g. laptops).

- **SIMD utilization:** Ray-Strips enable us to efficiently detect and handle incoherent packets by allowing to select the use of data parallelism in SIMD units, since at any point is is possible to either handle multiple rays or multiple primitive intersections in parallel. This also allows improved SIMD performance for tracing single rays that do not benefit from packet tracing methods.

*e-mail: cl@cs.unc.edu
†e-mail:sungeui@llnl.gov
‡e-mail:dm@cs.unc.edu

Organization: The rest of the paper is organized as follows: In section 2, we briefly survey related work on mesh representations and ray tracing. We discuss issues in ray tracing massive models in section 3 and give an overview of our approach. Section 4 presents our mesh representation in detail and section 5 describes efficient traversal and intersection algorithms based on our representation. We discuss our implementation and highlight the performance on different benchmarks in Section 6.

2 Previous work

In this section we give a brief overview on previous work on ray tracing of massive models and and mesh representations.

Coherent ray tracing: Current interactive ray tracers use packets of rays to amortize hierarchy traversal and primitive intersection cost over multiple rays by assuming there is strong coherence among packets of rays. The packet traversal algorithm was first introduced for kd-trees [WBWS01] and has been applied to different structures such as bounding volume hierarchies [WBS06; LSTM06] and s-kd-trees [WK06]. Recently, Reshetov et al. [RSH05] proposed a multi-level ray tracing method, which can split ray packets and adapts ray packet size to the geometric complexity.

Out-of-core techniques: One of the major challenges when dealing with massive models is the high memory footprint during rendering. To address this issue, many out-of-core techniques have been proposed in visualization [CESL*03], rasterization[CKS03; GM05]. Particularly, in ray tracing, Wald *et al*. [WDS04] improved the performance of ray tracing by employing an explicit memory handling scheme and using precomputed light-field-like representations of the scene to hide the latency incurred during the loading process. Pharr *et al*. [PKGH97] proposed an algorithm to optimize memory coherence in ray tracing by reordering rays so that they access the primitives in a scene in a coherent manner. They mainly showed their performance improvement for offline ray tracing. Also, DeMarle *et al*. [DGP04] described a simplified version of reordering in the context of an interactive renderer. Our approach is complimentary to these algorithms and can be combined with them.

Simplification: Levels-of-detail (LODs) have been widely used to accelerate rasterization of large polygonal data sets [LRC*02] and have been applied to improve the performance of ray tracing [CLF*03; WS03; SMD*06; YLM06]. However, these LOD techniques may introduce visual artifacts in the final images and their representations can require bigger disk storage than those of non-LOD based methods.On the other hand, our method reduces memory and disk requirements without introducing any visual artifacts.

Mesh and Hierarchical Representations: There has considerable work on computing compact mesh representations for triangulated models, especially for GPU rasterization. Since modern GPUs maintain a small buffer to reuse recently accessed vertices, triangle ordering can also improve frame rates. This approach was pioneered by Deering [Dee95]. The resulting ordering of triangles is called a *triangle strip* or *a rendering sequence*. There has been considerable work on designing improved algorithms to generate these sequences [Hop99; DGBGP05; NS06] and extending them to view-dependent rendering[ESAV99; KBG02; YLPM05].

In the ray tracing literature, there is work on rendering and improved representations of subdivision meshes [KDS98; MTF03; CLF*03; SMD*06] and tessellated versions of surfaces such as Bézier patches [BWS04]. Amanatides and Choi [AC97] present an edge-based ray-mesh intersection method for regular meshes using Plücker coordinates. All of these have the advantage that connectivity between vertices is implicit and therefore does not need to be stored. In contrast, there is relatively little work on efficiently handling triangular meshes. Galin and Akkouche [GA05] use a triangle fan decomposition of the model and show an efficient method for intersecting a ray against fans. However, it is not clear whether their decomposition method will work for various kinds of triangulated models.

Compression: There also have been tremendous research efforts to compactly represent meshes and good surveys are available [AG05]. Most previous mesh compression schemes were mainly designed to achieve maximum compression as they were targeted for archival use or transmission. However, it is not clear whether these compressions algorithms can be directly used to improve the performance of ray tracing algorithms, as they may not provide random access to the mesh primitives.

Coherent layouts: Since the stored order of triangles and vertices can affect the performance of applications, coherent layouts minimizing the number of cache misses have been researched. These includes space filling curves [Sag94; LP01], cache-aware layouts [Hav00], cache-oblivious mesh layouts [YLPM05]. Our algorithm is complimentary to all the layout techniques and can potentially be combined with some of them to achieve improved performance.

3 Overview

In this section we briefly explain the two main issues that arise during ray tracing of complex models: low ray coherence and high memory requirements. Then, we present an overview of our approach that is designed to address both of those issues. We also assume that the basic input primitives are triangulated models for the rest of the paper.

3.1 Ray Coherence

Ray coherence techniques such as ray packet traversal and intersection [WBWS01] and multi-level ray tracing [RSH05] can significantly improved the performance of ray tracing. These approaches simultaneously perform intersection tests for rays in a group against a hierarchy (e.g. kd-tree or bounding volume hierarchy) and geometric primitives of a model. Assuming that the rays are spatially coherent, the traversed path of rays in the group during hierarchy traversal will be similar for all the rays. Moreover, almost always the SIMD functionality of current CPUs is used for hierarchy traversal and intersections of multiple rays in parallel. Therefore these ray coherence techniques can improve the performance of ray tracing by reducing overall traversal steps.

One major disadvantage of ray packet traversal and intersection is its behavior for incoherent packets, where all rays in a group are forced to traverse different parts of the hierarchy. For complex models, this negates all the reduction of traversal steps gained by packet tracing. It will also result in a low utilization ratio of SIMD vector resources and can even make packet traversal slower than actually tracing single rays due to the overhead of preparing data for SIMD usage. In general, this is more likely to happen as the number of primitives in the model increases (see Fig. 1 for illustration). For ray tracing, this means that a large part of the computational resources is not fully utilized since the SIMD vector units in modern CPUs are typically used to trace several rays in parallel. Moreover, improving the utilization ratio of SIMD units is becoming more important for future CPU or many-core architectures with wider SIMD instructions [ARC*06].

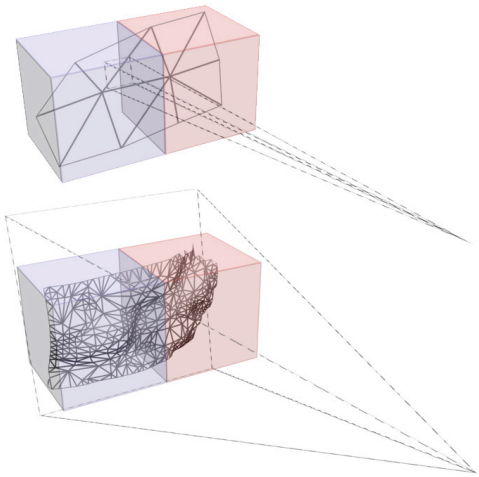


Figure 1: **Ray coherence for packets of rays:** *Consider a group of rays organized as a ray packet (visualized here by the enclosing frustum). In the upper image, the complexity of the geometry proportional to the frustum size is low, so rays will likely traverse the same part of the hierarchy and will hit the same or close primitives. In the lower image, the geometric complexity is very high relative to the frustum size. Traversal and intersection overhead are high since rays will traverse very different paths in the tree.*

3.2 Memory Requirements

Ray tracing algorithms typically need two major data structures: a hierarchical acceleration data structure such as kd-trees or bounding volume hierarchies (BVHs) and a geometric representation of the primitives associated with the hierarchy. The acceleration structure is used to improve the performance of ray tracing by allowing a quick search for the primitives that a ray or group of rays can potentially intersect. However, the acceleration structure adds a significant memory overhead in addition to storing the actual primitives of an input mesh. For example, given a model consisting of n triangles a performance-optimized BVH as used in recent systems [WBS06; LSTM06] needs $2n-1$ BVH nodes and each node requires 32 bytes.

Ray tracers commonly store triangles individually along with their geometric information. One of the most memory efficient, but still usable triangle representations for ray tracing is a simple list of three indices per triangle that share vertices. However, most optimized implementations require additional information such as the normal, a projection plane or other precomputed information needed by efficient triangle intersection algorithms and therefore, use of shared vertices may not be possible. For example, the system described in [Wal04] uses 48 bytes per triangle for storage. Therefore, these hierarchical acceleration data structures and geometric representations for interactive ray tracing can require a large amount of memory and disk space. As an example, for the Stanford Lucy model consisting of approximately 14 million vertices and 28 million triangles, the minimal indexed triangle list including vertices requires a total of 482 MB of data, and the optimized triangle representation $1,284$ MB. A BVH for the model adds another $1,712$ MB.

Compared to efficient mesh representations such as triangle strips for rasterization, data representations for ray tracing take much more memory space. Moreover, previous research [WDS04; YLM06] has shown that the performance of ray tracing reduces drastically when the acceleration data structures and the primitives do not fit into the main memory. Considering that one of the attractions of ray tracing is its logarithmic performance with regard to the number of input primitives, it is important that we design compact memory representations for interactive ray tracing.

3.3 Our Approach

To address those major issues for ray tracing massive models, we introduce a novel mesh representation called Ray-Strips. This representation is similar to triangle strips used to accelerate rasterization. However, this representation is designed such that it is amenable to ray tracing while maintaining compactness. Each Ray-Strip represents a mesh that is a subset of the overall input model, and all of them are organized in a high-level scene hierarchy, where each leaf node references a Ray-Strip. In addition, Ray-Strips also represent a balanced hierarchy from the triangles encoded by the strips; therefore, Ray-Strips itself represent the low-level scene hierarchy.

The choice of a two-level hierarchy stems from the observation that the high levels of the hierarchy of an optimized tree structure such as one computed with the surface-area heuristic (SAH) [Hav00] have the largest effect on overall ray tracing performance. Assuming that the underlying geometric primitives are connected and close, the tree structure closer to the leaf nodes in the hierarchy is likely to be evenly distributed in space and therefore can be split evenly without introducing 'bad' split decisions (e.g. as measured by the SAH metric). This means that at some point it is possible to switch to a balanced hierarchy (by splitting at the object median) which requires less memory because no child pointers are necessary.

3.3.1 High-level hierarchy

Our representation consists of a high-level hierarchy and a strip representation for the triangle mesh. The high-level hierarchy is built on the original geometric primitives but only up to a certain triangle count or other subdivision criterion. Therefore, each leaf of the high-level hierarchy references a subset of triangle primitives. This hierarchy can be any kind of acceleration structure such as a kd-tree, BVH, grids or spatial kd-trees. Note that the size of the tree is significantly smaller and the bounding boxes of the tree are large since each node of the tree contains many triangle primitives. Therefore we can maintain higher ray coherence during traversing the high-level hierarchy. We use a BVH of axis-aligned bounding boxes and construct the hierarchy using the SAH for best performance. We perform single ray and ray packet traversal on our high-level hierarchy in the same manner as described in [WBS06; LSTM06].

3.3.2 Ray-Strips

Our low-level hierarchy represents a triangle mesh and is intended to both reduce the memory footprint of the geometric primitives in the mesh as well as to allow efficient traversal and intersection. Ray-Strips are based on triangle strips, but also contain an object hierarchy on the strip that is similar to spatial kd-trees and allows traversal for efficient culling. This representation is described in more detail in section 4. One major aspect of this representation is that at any step during the traversal it is directly possible to intersect with all the triangles contained in the current sub-tree instead of traversing further. Later, we show that this representation also allows intersection against multiple primitives in parallel at any time during traversal. This is in contrast to prior SIMD algorithms that intersect multiple rays with a single triangle. The traversal and intersection algorithms are described in section 5.

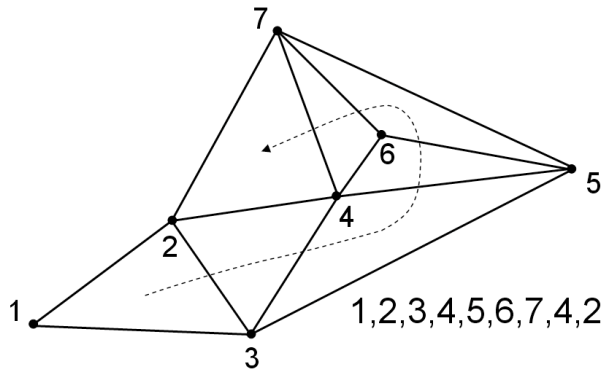


Figure 2: **Example triangle strip:** *A simple mesh of 7 triangles that can be represented by a triangle strip of length 9.*

4 Ray-Strip Representation

In this section we explain our Ray-Strip representation and its relationship with our two levels of hierarchy. We will also discuss the construction of Ray-Strips.

4.1 Representation

As mentioned in the previous section, the Ray-Strip representation is based on triangle strips that define a set of connected triangle as a list of vertices, where connectivity is given implicitly by the order of vertices. Fig. 2 shows an example of a simple mesh represented as a triangle strip. However, even though a triangle strip is an efficient way to represent a triangle mesh, intersecting a ray with each triangle in the strip can be rather slow for ray tracing. Therefore, we generate a memory-efficient spatial kd-tree hierarchy on top of the strip to allow traversal for determining the visible triangles in Ray-Strip quickly.

Given n triangles in the mesh, our Ray-Strip consists of a list of $n+2$ vertex indices to represent the triangles. We recursively split the strip at the median edge into two parts of equal number of triangles (or a difference of one, similar to an AVL tree) in order to define a balanced tree on the strip. An example of this operation is shown in Fig. 3: The red edge is the one that defines the split and will be a part of the two resulting sub-strips. To use the balanced tree on the strip as an acceleration data structure for ray tracing, we also need to know the spatial bounds of the nodes during traversal. We avoid having to store the full bounding box by selecting the axis of the least overlap between children and then store the maximum bound for the left and the minimum bound for the right child. Effectively, we store the split axis as well as the coordinates for the two bounds, which is very similar to spatial kd-trees [WK06]. Currently, we split down to sub-strips of length 4, i.e. representing 2 triangles, which has performance advantages for SIMD usage and will be discussed in detail later. Because we always perform median splits, this hierarchy is balanced and can therefore be stored in an array without the need for child pointers. In addition, each node also directly corresponds to an consecutive interval of in the vertex list of the triangle strip. Therefore, it is not necessary to actually store leaf nodes as the traversal algorithm can detect whenever it has reached a sub-strip of sufficient size. Because the spatial kd-tree is an object hierarchy, the number of nodes in the tree is also known: for n triangles, we need a total of $\lceil n/2 \rceil - 1$ nodes for the hierarchy. Note that in order to store the tree in a flat array of that size as well, we need to ensure that each new level of the tree starts on the leftmost subtree of the previous tree level. We achieve this by slightly modifying the median split rule to take this into account.

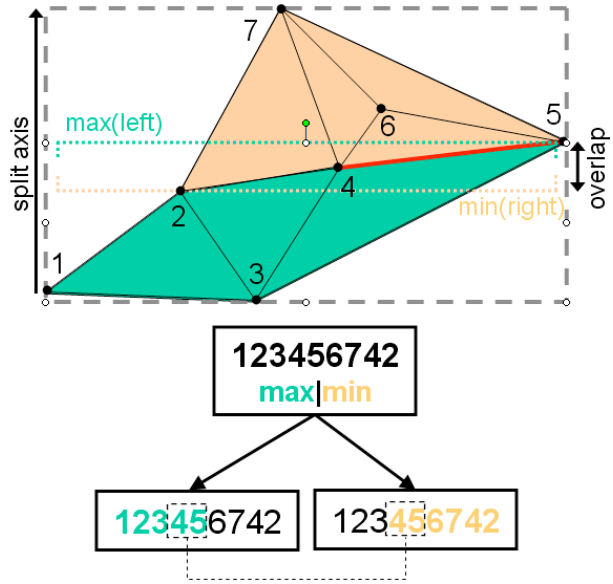


Figure 3: **Strip hierarchy:** *We subdivide a strip at the center edge (shown in red) and then record the minimum and maximum bounds on one axis for the resulting two children. This defines a balanced spatial kd-tree on the mesh such that only the two bounds need to be stored. The interval in the strip is defined implicitly (also note that intervals overlap since both children reference the center edge 4-5.)*

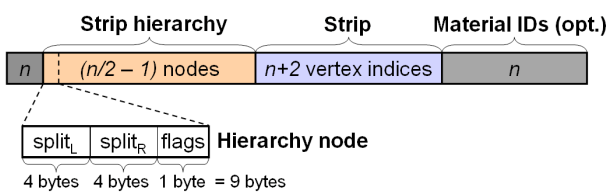


Figure 4: **Ray-Strip structure:** *The data structure for storing a Ray-Strip consists of the header recording the number of triangles n, the hierarchy defined on the strip as well as the triangle strip representing the geometry. We also allow optional per-triangle information, such as material references in this case. Note that the size of the Ray-Strip is known a priori and only depends on n.*

However, there is one issue: the spatial kd-tree always assumes that the left sub-strip in a split is the one on the left with regard to the split axis as well. This can be true, but in general will only sometimes be the case. Therefore, we also need an additional flag per split that indicates whether the split is in the normal split order or in reverse. Overall, a node in the hierarchy needs to store two split coordinates, the split axis and the reverse flag. We encode the latter two into one byte for simplicity, although technically they only need 3 bits (2 for the axis and one for the flag). Fig 4 shows the actual memory representation of a Ray-Strip: a header stores the number of triangles in the strip, then all the hierarchy nodes are stored and finally the actual triangle strip as a sequence of vertex indices. We also store additional per-triangle information, which currently is used for saving a material reference (as a 16-bit integer) for shading purposes, but other applications are possible. If materials are not used or per-vertex colors are used, then this can be left out.

4.2 Strip generation:

The decomposition of a model into triangle strips is a well-studied problem called stripification and several algorithms are known in the literature [Hop99; DGBGP05; NS06]. For a given input model, we can subdivide the model into triangle strips and build the high-level hierarchy on those strips. Unfortunately, typical stripification algorithms tend to generate very long and thin triangle strips. However, these triangle strips optimized for high GPU vertex cache utilization may not be good partitions of the model for ray tracing since they do not take any spatial relationships such as spatial distribution of geometric primitives or overlap into account, which have a decisive effect on the run-time performance of ray tracing. Therefore, our approach for finding suitable triangle strips is as follows: we first run the mesh subdivision algorithm to decompose an input mesh into small chunks of the mesh, each of which has a group of primitives below a threshold count. Then, we build the high-level hierarchy from the chunks of the mesh.

As a next step, we attempt to build a triangle strip from each chunk of the mesh. For strip computation, we use the Stripe library [ESV96] for computing a stripified version of the mesh. Note that even for a connected mesh, the existence of a triangle strip fully spanning a mesh is not guaranteed. Even worse, the task of finding such a decomposition has been shown to be NP-complete [EMX02]. Therefore, if we are not able to build a strip for a chunk, we partition the chunk into sub-chunks. By doing this partitioning, we create new child nodes for a node containing the chunk in the high-level hierarchy.

5 Ray Tracing using Ray-Strips

In this section, we present our algorithm for fast ray tracing using Ray-Strips. It includes two operations: hierarchy traversal and triangle intersection. We present efficient algorithms for single rays as well as ray packets.

5.1 Hierarchy Traversal

The Ray-Strip hierarchy is essentially a spatial kd-tree and can be traversed in a very similar manner (see [WK06], also illustrated in Fig. 5). Starting at the root node, the bounding box of the mesh is split in the axis stored in the node at both stored coordinates. Given a ray, the distances $split_L$ and $split_R$ to both split planes can be tested to determine whether it hits the left, the right or both children, where the order is given by the ray's direction.

If the ray intersects both the children, the near one – as determined by the ray's direction – is traversed first and the far one is pushed on a stack. The children are processed in the same manner by updating the bounding box from the parent with the child's respective split plane. Since the hierarchy is subdivided according to fixed rules and the number of triangles in the mesh is known, the traversal just needs to keep track of how many triangles are left in the current sub-tree to know when it encounters a leaf. This is done by updating the interval in the triangle strip representing the triangles whenever the traversal jumps to another node. As mentioned above, this also involves the reverse flag for the node to find out which side of the interval corresponds to the left and right children, respectively. The pseudo-code implementing this traversal is shown in 1

5.2 Intersection Computation

At every step in the traversal, the ray tracer can decide to intersect with the triangles contained in the node, which are defined by the current interval in the triangle strip sequence. The naïve approach would be to test every triangle by itself using a standard triangle intersection algorithm, but that would ignore information available

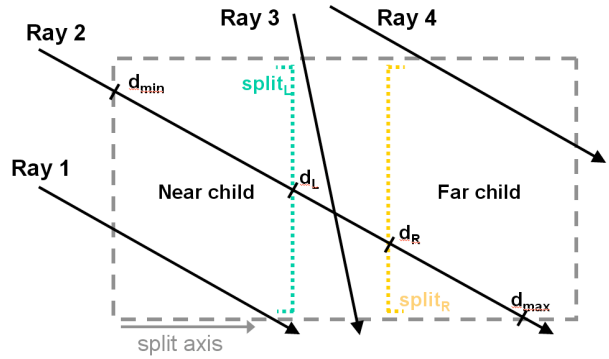


Figure 5: **Traversing a node:** *Given an inner node of the mesh hierarchy, there are two split planes associated with it. For each ray, if the distance to the split plane (d_L or d_R) is inside the interval defined by the distances, (d_{min}, d_{max}) where the ray enters and exits the box, then that child needs to be traversed. For ray 1, only the near child is traversed, for ray 2 both need to be traversed and ray 4 only traverses the far child. Note the special case for ray 3 which traverses neither of the two children.)*

Algorithm 1 Traversal of the Ray-Strip hierarchy

```
node ← root
left ← 0, right ← #tris
[min, max] = ray.intersectWithBoundingBox()
while node != NULL do
  while !node.isLeaf() do
    dist1 ← ray.distTo(node.axis, node.split1)
    dist2 ← ray.distTo(node.axis, node.split2)
    if dist1 < min then
      if dist2 > max then
        [node,left,right,min,max] ← stack.pop()
      else
        [node,left,right] ← node.farChild(ray, node.reverseFlag)
      end if
      continue
    else if dist2 ≤ max then
      stack.push(node.farChild(ray, node.reverseFlag), max(dist2, min), max)
    end if
    max ← min(dist1, max);
    [node,left,right] ← node.nearChild(ray, node.reverseFlag)
  end while
  intersectMesh(left, right)
  [node,left,right,min,max] ← stack.pop()
end while
```

as part of the strip representation. Instead, we take advantage of the connectivity information. Specifically, we use an edge-based intersection that tests the ray for containment using Plücker coordinates[Sho98], which allows us to test the orientation of a ray relative to an edge. Given three edges defining a triangle in a consistent order (clockwise/counter-clockwise), the ray intersects the triangle if and only if the signs of the Plücker edge tests match. Given an interval in the triangle strip, we can compute all the edge tests directly and then test each consecutive set of three edge results for intersection with the respective triangle. Since there are shared edges, this is more efficient than testing each of the triangles. In particular, we can easily test 4 edges at the same time by using SIMD instructions on current CPUs. This is particularly effective when tracing just one ray because we use the available data parallelism.

5.3 Ray Packets

Both the traversal and the intersection algorithms described above can be extended to handle ray packets. For traversal, the algorithm given above changes in that a sub-tree is traversed if the bounding box is intersected by any of the rays. For intersection, the edge tests have to be performed for each ray in the packet. However, if the rays in the packet share the same origin, the Plücker intersection can be optimized to reuse the coordinate computation, which is presented in detail in [Ben05].

The important problem for ray packets on massive models is that rays can be very incoherent at the lower levels of the hierarchy due to small triangles and the geometric detail. This leads to traversal and intersection steps where just one or very few rays are 'active', i.e. intersect the current sub-tree. Ray-Strips allow us to detect those cases and switch to parallel edge intersection for each active ray whenever appropriate. Thus, when the number of active rays becomes smaller than a certain threshold, we switch to single ray intersection for all the active rays. We get improved performance because the intersection of a single ray with Ray-Strips takes advantage of data parallelism and can quickly test one ray against a large set of triangles.

6 Implementation and Results

We now present results from our system that uses Ray-Strips and compare it to prior methods. We also analyze the performance of our method and discuss its limitations.

6.1 Results and Comparison

We have implemented the Ray-Strip representation and traversal methods on a Intel Xeon system at 2.5 GHz and 2 GB of RAM. Although the system has multiple cores, we only use one thread for rendering. We use the Intel SSE instruction set which allowed to perform traversal and intersection of 4 rays (a 2×2 ray packet) simultaneously similar to other interactive ray tracing implementations. All performance numbers are given in frames per second at a resolution of 1024×1024 pixels.

We test our methods with several benchmark scenes of varied complexity and properties, ranging from 67 thousand to 30 million triangles (see Fig. 6.)

Memory complexity: Table 1 summarizes the results for memory reduction when using Ray-Strips for our benchmark scenes. We compare the results to a current BVH implementation as used in interactive ray tracing [WBS06; LSTM06] that stores one triangle per node for performance. To compare the geometry representation, the table also shows memory cost for storing the triangles both as a minimal list of 3 vertex indices and one material index (14 bytes per triangle, plus global vertex list), as well as the performance optimized triangle representation used in [Wal04] (48 bytes / triangle). To better compare the results, we also split up the total memory footprint into the memory taken by hierarchy and actual geometry (triangles and vertices) for all approaches. The results show that Ray-Strips reduce the memory cost for all models significantly, for both hierarchy as well as geometry. Note that the impact of Ray-Strips is slightly lower for architectural and CAD scenes such as the power plant. This is due to lesser mesh connectivity compared to the meshes of 3D scanned and the iso-surface models.

We also compare Ray-Strips to several other hierarchies that were previously used to reduce the cost for storing the acceleration structure. Table 2 shows results for a subset of scenes from Table 1. kd-trees and spatial kd-trees [WK06] are both relatively lightweight structures that are optimized for fast ray tracing of static scenes. Previous approaches also include compressed BVH structures such as limited precision BVHs [Mah05] as well as lightweight BVHs [CSE06]. Both methods quantize the coordinates of the axis-aligned bounding boxes and are therefore able to reduce the memory cost for storing each node. They also store multiple references to triangles in each leaf node in order to use less nodes overall. As the results show, Ray-Strips represent the hierarchy much more efficiently than both kd-trees and spatial kd-trees, but may take somewhat more space than the compressed representations. However, reversing the compression during traversal slows down the actual ray tracing process. In addition, bounding boxes are enlarged due to the conservative quantization which causes additional, false intersections and therefore leads to an overhead of traversed nodes. Therefore, rendering using the compressed structures is significantly slower than using Ray-Strips. In addition, neither of the approaches above reduces the memory footprint for storing the geometry.

Model	Triangles	Ray-Strip (fps)	BVH (fps)
Bunny	67k	0.35	1.19
Buddha	1.2M	0.43	1.00
Thai	10M	0.48	0.88
Iso-surface	10M	0.30	0.70
Power plant	12.7M	0.89	0.53
Lucy	30M	0.51	0.02

Table 3: **Results: Rendering performance for single rays.** *This table shows rendering performance using single rays for a standard BVH implementation and the same implementation running on Ray-Strips. The resolution for rendering was* 1024^2 *using one core, and only primary rays were used.*

Rendering performance: The performance results of using Ray-Strips are presented for single rays in Table 3. Although the pure rendering performance is still somewhat less than a standard BVH (also due to a less optimized implementation compared to the BVH code), the behavior of Ray-Strips for complex models shows that the SIMD utilization and memory complexity has effects on the frame rates. Note that for the Lucy scene the BVH renderer had to operate out of core and therefore was slower than would otherwise be expected.

Construction: Performing the stripification of the input model adds some overhead to the construction of the hierarchy, which is shown in Table 4. Almost all the additional time is spent in the stripification library, which is not optimized for speed yet. If necessary, a faster library can be used instead. However, since the main application for Ray-Strips is rendering complex models, the construction is just performed as a preprocessing step and the hierarchies can be loaded from disk for runtime rendering. Note that the CAD models in general only have much smaller triangle strips since there is a lower degree of connectivity in the input data.

6.2 Analysis and Limitations

The results above show that Ray-Strips are an efficient representation for our tested complex benchmarks. In practice, the improvements gained from using Ray-Strips are highly dependent on finding sufficiently long triangle strips to build the hierarchy on. Obviously, this makes our approach unsuitable for models without any mesh connectivity (e.g. without any shared vertices). In that case, the Ray-Strip representation shown in section 4 becomes an indexed triangle list with a standard BVH (just with the addition on the 2-byte header storing the triangle count) and performance gains are lost. However, the memory overhead added is only very small and neither will there be a performance loss compared to using an indexed triangle list to start with.

Our current implementation uses the the stripification library Stripe [ESV96], which can handle very general meshes without many limitations. More current stripification algorithms such as [DGBGP05] assume that the input mesh is a manifold, which is usually not true for the input scenes that we looked at. It could be possible to include newer aspects into the stripification algorithm used by us to further improve the results.

Model	Tris	Memory total (MB) / Reduction			Memory geometry (MB)			Memory hierarchy (MB)	
		Ray-Strip	minimal	optimized	Ray-Strip	minimal	optimized	Ray-Strip	BVH
Bunny	67k	1.5	5.5 (-73%)	7.4 (-80%)	1.2	1.4	3.2	0.3	4.2
Buddha	1.2M	23	87 (-74%)	116 (-80%)	11	20	49	11	66
Thai	10M	215	801 (-72%)	1068 (-79%)	163	190	457	106	610
Power plant	12.7M	395	1075 (-63%)	1361 (-71%)	196	296	583	198	778
Lucy	30M	581	2247 (-74%)	2996 (-81%)	301	535	1284	279	1712
Iso-surface	10M	220	822 (-73%)	1095 (-80%)	110	195	469	108	626

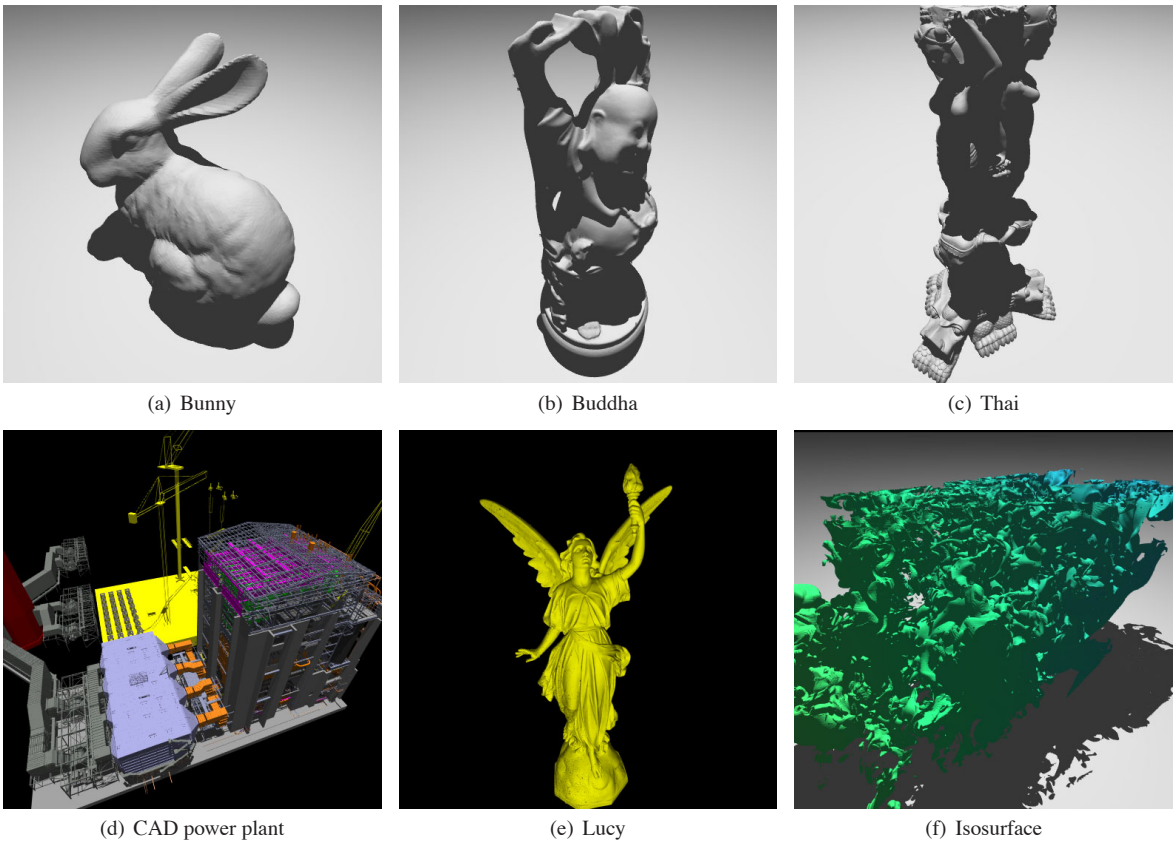
Table 1: **Results: Memory complexity.** *This table shows the memory requirements for our approach as well as the two usual other triangle representations and a BVH using axis-aligned bounding boxes. The total memory cost can be split up into memory requirements for the hierarchy as well as for the actual geometry. The comparison structures are a* minimal *representation (which stores just 3 vertex references and a material pointer per triangle), as well as the triangle representation optimized for intersection speed as used in [Wald 2004]. For Ray-Strips and indexed triangle lists, the size of the global vertex list is included.*

Model	Triangles	Ray-Strip	kd-tree [WK06]	spatial kd-tree [WK06]	LBVH [CSE06]	LPBVH [Mah05]
Bunny	67k	0.3	4.1	0.9	0.5	0.3
Buddha	1.2M	11	29	17	8.3	n/a
Lucy	30M	279	n/a	n/a	214	103

Table 2: **Results: Hierarchy memory comparison:** *We compare the relative memory complexity of several acceleration structures to Ray-Strips (all values in MB, 'n/a' signifies that the model was not used in the respective paper) The spatial kd-tree is an object-level hierarchy similar to our Ray-Strip hierarchy, LBVH and LPBVH are both compressed BVH structures storing multiple primitives at leafs and values were taken for the highest compression ratio in the respective papers, which usually leads to a significant degradation in performance. Note that our approach could use the compression from those approaches for further memory reduction. Neither of the techniques above compresses the geometry data.*

Model	Tris	Construction time (ms)		Avg. strip length (triangles)	#strips
		Ray-Strip (s)	BVH (s)		
Bunny	67k	0.4	0.03	19.29	4260
Buddha	1.2M	5.5	0.7	29.58	46591
Thai	10M	48.1	7.1	24.95	502577
Iso-surface	10M	49.3	6.3	24.69	519468
Power plant	12.7M	68.9	29.6	7.75	1890760
Lucy	30M	1764	149.6	31.01	1122074

Table 4: **Results: Construction statistics** *The construction time for both plain BVHs (using surface-area heuristic splits) as well as for Ray-Strips is shown. Our approach adds is much slower here, which is mainly due to the stripification process. The timings for Lucy are particularly slow because the model does not fit into memory during construction. We also show the number of strips and the average strip length. Note that for architectural scenes, the stripification can only find shorter strips due to limited connectivity.*



(a) Bunny (b) Buddha (c) Thai

(d) CAD power plant (e) Lucy (f) Isosurface

Figure 6: **Benchmark scenes:** *We test our Ray-Strip representation on a range of complex scenes of different types including scanned, iso-surfaces, and CAD models.*

As presented in section 3 and 4, our system uses a BVH with axis-aligned bounding boxes as the high-level hierarchy, but is important to note that in principle any acceleration structure can be used instead since it is built on the Ray-Strips. We find that a BVH usually provides a reasonable compromise between rendering speed, flexibility and ease of use. It is also easily updateable so that dynamic scenes can be handled efficiently. If maximum performance for a static scene is desired, a kd-tree may be a better choice and might reduce the memory footprint slightly. Note that Ray-Strips can be updated in the same manner as BVHs, but have the limitation that mesh connectivity cannot change in the animation, e.g. objects cannot 'break'.

7 Future Work and Conclusion

We have proposed a novel compact representation, Ray-Strips, for ray tracing triangular meshes that can drastically reduce the memory requirements for ray tracing of massive models. We showed how the new representation can increase both SIMD utilization and rendering speed for complex models compared to a standard acceleration structure, which makes our approach relevant for many current and future hardware architectures.

For future work, we are interested in extending the mesh representation to also include geometry compression and hierarchy compression, which is expected to decrease memory complexity even further. Another promising avenue is integration with a LOD technique such as R-LODs [YLM06] for out-of-core rendering of more complex models. We are also interested in investigating stripification algorithms that are optimized towards generating strips suitable for ray tracing with Ray-Strips. Finally, we plan to investigate the effects of using Ray-Strips on data parallel architectures such as GPUs.

References

AMANATIDES J., CHOI K.: Ray tracing triangular meshes. In *Proceedings of the Eighth Western Computer Graphics Symposium* (1997), pp. 43–52.

ALLIEZ P., GOTSMAN C.: *Recent advances in compression of 3D meshes*. Springer, 2005, pp. 3–26.

ASANOVIC K., R., CATANZARO B., GEBIS J., HUSBANDS P., K., PATTERSON D., PLISHKER W., SHALF J., WILLIAMS S., YELICK K.: *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. Rep. UCB/EECS-2006-183, EECS Dept., Univ. of California, Berkeley, 2006.

BENTHIN C.: *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Computer Graphics Group, Saarland University, 2005.

BENTHIN C., WALD I., SLUSALLEK P.: Interactive ray tracing of free-form surfaces. In *AFRIGRAPH '04: Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa* (New York, NY, USA, 2004), ACM Press, pp. 99–106.

CHIANG Y.-J., EL-SANA J., LINDSTROM P., PAJAROLA R., SILVA C. T.: Out-of-core algorithms for scientific visualization and computer graphics. *IEEE Visualization 2003 Course Notes* (2003).

CORRÊA W. T., KLOSOWSKI J. T., SILVA C. T.: Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of PVG 2003 (6th IEEE Symposium on Parallel and Large-Data Visualization and Graphics)* (2003), pp. 1–8.

CHRISTENSEN P. H., LAUR D. M., FONG J., WOOTEN W. L., BATALI D.: Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *j-CGF* 22, 3 (Sept. 2003), 543–552.

CLINE D., STEELE K., EGBERT P. K.: Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools: JGT* (2006).

DEERING M. F.: Geometry compression. In *ACM SIGGRAPH* (1995), pp. 13–20.

DIAZ-GUTIERREZ P., BHUSHAN A., GOPI M., PAJAROLA R.: Constrained strip generation and management for efficient interactive 3d rendering. In *Computer Graphics International* (2005), pp. 115–121.

DeMarle D. E., Gribble C. P., Parker S. G.: Memory-savvy distributed interactive ray tracing. In *EGPGV* (2004), pp. 93–100.

Estkowski R., Mitchell J. S. B., Xiang X.: Optimal decomposition of polygonal models into triangle strips. In *SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry* (New York, NY, USA, 2002), ACM Press, pp. 254–263.

El-Sana J., Azanli E., Varshney A.: Skip strips: Maintaining triangle strips for view dependent rendering. In *IEEE Visualization '99 Proceedings* (1999), pp. 131 – 138.

Evans F., Skiena S. S., Varshney A.: Optimizing triangle strips for fast rendering. In *IEEE Visualization '96* (1996), Yagel R., Nielson G. M., (Eds.), pp. 319–326.

Galin E., Akkouche S.: Fast processing of triangle meshes using triangle fans. In *SMI '05: Proceedings of the International Conference on Shape Modeling and Applications 2005 (SMI' 05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 328–333.

Gotsman C., Gumhold S., Kobbelt L.: *Simplification and Compression of 3D Meshes*. Springer, 2002, pp. 319–361.

Gobbetti E., Marton F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph. 24*, 3 (2005), 878–885.

Havran V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

Hoppe H.: Optimization of mesh locality for transparent vertex caching. *ACM SIGGRAPH* (1999), 269–276.

Isenburg M., Lindstrom P.: Streaming meshes. *IEEE Visualization* (2005), 231–238.

Karni Z., Bogomjakov A., Gotsman C.: Efficient compression and rendering of multi-resolution meshes. In *IEEE Visualization* (2002), pp. 347–354.

Kobbelt L. P., Daubert K., Seidel H.-P.: Ray tracing of subdivision surfaces. In *Rendering Techniques '98 (Proceedings of the Eurographics Workshop)* (New York, 1998), Springer-Verlag, pp. 69–80.

Lindstrom P., Pascucci V.: Visualization of large terrains made easy. *IEEE Visualization* (2001), 363–370.

Luebke D., Reddy M., Cohen J., Varshney A., Watson B., Huebner R.: *Level of Detail for 3D Graphics*. Morgan-Kaufmann, 2002.

Lauterbach C., S., Tuft D., Manocha D.: *RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs*. Tech. Rep. TR-06-010, Univ. of North Carolina at Chapel Hill, 2006, 2006.

Mahovsky J.: *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, September 2005.

Müller K., Techmann T., Fellner D. W.: Adaptive ray tracing of subdivision surfaces. *Comput. Graph. Forum 22*, 3 (2003), 553–562.

Nehab D., Sander J. B. P. V.: Triangle order optimization for graphics hardware computation culling. *Symposium on Interactive 3D Graphics and Games* (2006).

Pharr M., Kolb C., Gershbein R., Hanrahan P.: Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 101–108.

Reshetov A., Soupikov A., Hurley J.: Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3 (2005), 1176–1185.

Sagan H.: *Space-Filling Curves*. Springer-Verlag, 1994.

Shoemake K.: Pluecker coordinate tutorial. *Ray Tracing News 11*, 1 (1998).

Stoll G., Mark W. R., Djeu P., Wang R., Elhassan I.: *Razor: An Architecture for Dynamic Multiresolution Ray Tracing*. Tech. Rep. TR-06-21, Dept. of CS, Univ. of Texas, 2006.

Wald I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

Wald I., Boulos S., Shirley P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics* (2006).

Wald I., Benthin C., Wagner M., Slusallek P.: Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)* (2001), Chalmers A., Rhyne T.-M., (Eds.), vol. 20, Blackwell Publishers, Oxford, pp. 153–164.

Wald I., Dietrich A., Slusallek P.: An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proceedings of the Eurographics Symposium on Rendering* (2004).

Wächter C., Keller A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006: Eurographics Symposium on Rendering*. (2006).

Wand M., Straber W.: Multi-resolution point-sampled raytracing. In *Graphics Interface* (2003).

Yoon S.-E., Lauterbach C., Manocha D.: R-LODs: Interactive LOD-based Ray Tracing of Massive Models. *The Visual Computer (Pacific Graphics)* (2006). To appear.

Yoon S.-E., Lindstrom P., Pascucci V., Manocha D.: Cache-Oblivious Mesh Layouts. *Proc. of ACM SIGGRAPH* (2005).

Yoon S.-E., Manocha D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. In *Computer Graphics Forum (Eurographics)* (2006). To appear.

6 SPVN: A New Framework for Distributed Interactive Visualization of Large Datasets

SPVN: A New Application Framework for Interactive Visualization of Large Datasets

Wagner T. Corrêa* James T. Klosowski† Christopher J. Morris‡ Thomas M. Jackman§
IBM T. J. Watson Research Center

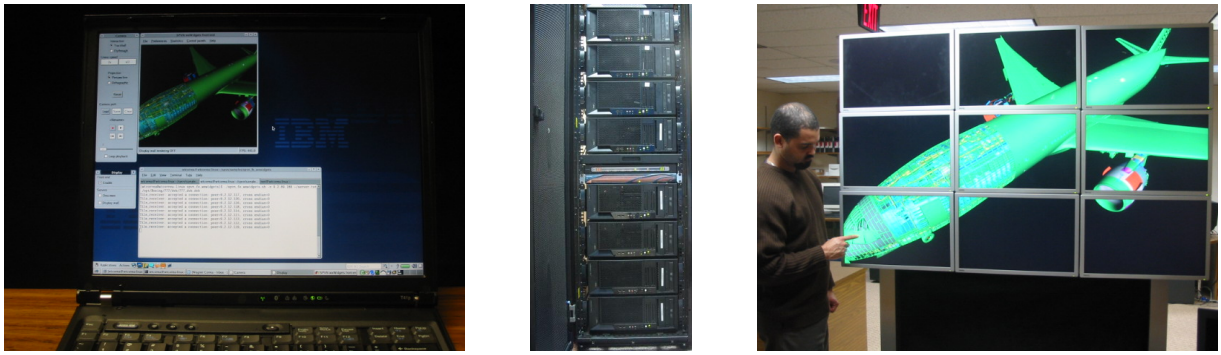


Figure 1: A thin client controls an SPVN application running on a cluster of PCs connected to a display wall.

Abstract

We present SPVN (Scalable Parallel Visual Networking), a new application framework for interactive visualization of large datasets. We designed SPVN with four main goals in mind. First, we wanted to make it easier for developers to write visualization applications that can handle massive datasets and deliver interactive frame rates. SPVN provides developers with efficient implementations of many optimization techniques such as spatialization, simplification, view-frustum culling, occlusion culling, multithreading, and prefetching. Second, we wanted developers to better leverage the performance and scalability of a cluster of inexpensive rendering servers, while insulating them from the complexities of distributed programming. SPVN provides implementations of sort-first rendering with dynamic load balancing, sort-last rendering with depth-order and binary-swap image compositing, and a distributed shared memory mechanism that gives programmers the illusion that all machines have all the data at all times. Third, we wanted to support multiple low-level rendering libraries. SPVN separates modeling from rendering so that the same scene can be rendered by different backends (e.g., OpenGL, DirectX, or ray tracing). Finally, we wanted SPVN to be easy to use and extend. SPVN uses the familiar concept of a scene graph, applies many well-established design patterns (e.g., smart pointers, factories, observers, and visitors), and allows for extensions of shape classes, rendering algorithms, and file formats using registry and plug-in mechanisms. We have used SPVN to develop both remote and immersive visualization applications, and found that SPVN reduces the amount of time and money it takes to write such applications.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; I.3.4 [Computer Graphics]: Graphics Utilities—Application packages

Keywords: distributed visualization, parallel rendering, large datasets, application framework

*e-mail:wtcorrea@us.ibm.com
†e-mail:jklosow@us.ibm.com
‡e-mail:cjmorris@us.ibm.com
§e-mail:tjackman@us.ibm.com

1 Introduction

Visualization of large datasets is an important problem with a wide range of applications, including computer-aided design and engineering, visualization of medical data, modeling and simulation of weapons, modeling and simulation of weather and ecosystems, exploration of oil and gas, and virtual training.

Developing large data visualization applications is very challenging. One of the main challenges is that the size of many interesting datasets overwhelms a single CPU or GPU. For example, the 3D model of the Boeing 777 airplane contains 350 million triangles, which add up to 14 GB of data. Despite the exponential progress in CPU and GPU performance, no single CPU or GPU today could render a dataset that large at interactive frame rates by simply drawing all the data every frame. In addition, our demand for larger datasets grows faster than the improvements in hardware technology. To alleviate this problem, researchers have developed many techniques that either prune or simplify the data that has to be drawn in a frame, but implementing these techniques is not trivial.

Another challenge in large data visualization is parallelization. Even after pruning and simplification, the amount of data left may overwhelm a single CPU or GPU, and a natural approach to handle this problem is to use a parallel machine with multiple CPUs or GPUs. Shared-memory parallel machines are more expensive and more difficult to upgrade than distributed-memory parallel machines, particularly clusters of commodity PCs, but distributed-memory parallel machines are typically harder to program.

A third challenge in large data visualization is dependency on low-level rendering libraries. Unfortunately, many visualization toolkits create an unnecessary coupling between modeling and rendering, and force the application developer to use their choice of rendering library. For example, Jupiter [Bartz et al. 2001], OpenSceneGraph [Burns and Osfield 1998], GTB [Corrêa 2004], OpenSG [Reiners et al. 1999], VTK [Schroeder et al. 1996], OpenGL Performer [SGI 2003], and Open Inventor [TGS 2003] all require OpenGL. For applications that run on Windows, a rendering library based on DirectX might be more suitable, and for applications that require higher photorealism, a rendering library based on ray tracing might be necessary.

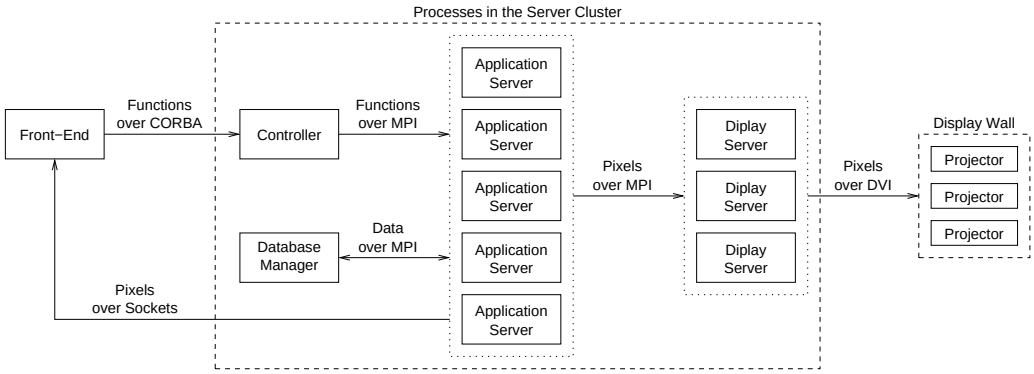


Figure 2: The components of a typical SPVN application.

Finally, a fourth challenge that developers of large data visualization applications face is learning how to use a particular framework and how to extend it to suit their needs.

In this paper, we present **SPVN** (Scalable Parallel Visual Networking), a new application framework for interactive visualization of large datasets that addresses all these challenges. To address the dataset size challenge, SPVN provides developers with efficient implementations of many optimization techniques such as spatialization, simplification, view-frustum culling, occlusion culling, multithreading, and prefetching. To address the parallelization challenge, SPVN focuses on clusters of inexpensive PCs, supporting the sort-first and sort-last parallel rendering strategies [Molnar et al. 1994], and providing a distributed shared memory mechanism that gives programmers the illusion that all machines have all the data at all times. To support multiple low-level rendering libraries, SPVN separates modeling from rendering so that the same scene can be rendered by different back-ends (e.g., OpenGL, DirectX, or ray tracing). Finally, SPVN strives to be easy to use and extend by using the familiar concept of a scene graph, applying many well-established design patterns (e.g., smart pointers, factories, observers, and visitors) [Gamma et al. 1995], and providing registry and plug-in mechanisms for extensions of shape classes, rendering algorithms, and file formats.

2 A Typical SPVN Application

Figure 1 shows a typical application written using SPVN. The basic idea is that the front-end drives the application running in the cluster, and the application servers generate pixels that may be sent to the front-end, to a display wall, or both.

More specifically (Figure 2), the front-end implements the interface with the user, and accesses the cluster controller functions through CORBA interfaces [Henning and Vinoski 1999]. Thus, the front-end may be written in any language supported by the CORBA Interface Definition Language compiler (for example, C++, Java, or Python). The controller forwards function calls from the front-end to the application servers. The application servers render image tiles, communicating with the database manager to read or update parts of the scene. The database manager interfaces with the file system. All processes running in the cluster communicate with each other via MPI [Gropp et al. 1999]. A server process does not have to run on a dedicated machine. In other words, there may be multiple server processes per physical machine.

To enable remote visualization, the application servers send pixels back to the front-end using sockets. In particular, the front-end can access the cluster remotely via internet. Each application server has a tile sender object that sends image tiles to the tile receiver object in the front-end. The user specifies whether or not the tiles should be compressed. The tile receiver object runs on a separate thread, and makes the tiles available to the main thread in no particular order. This scheme allows the main thread to draw the available tiles in parallel with the transmission of the other tiles.

To enable immersive visualization, the application servers may also send image tiles to the display processes that run on the machines that are connected to the projectors that make up the display wall. In this case the tile sender objects use MPI, as opposed to sockets, because the MPI implementation provided by the network vendor is highly optimized and typically gives us much lower latency and two to three times more bandwidth than sockets.

3 Built-in Optimizations

In this section we describe the optimization techniques that SPVN makes available to the application servers.

3.1 Spatialization

Spatialization is the process of building a hierarchical spatial data structure [Samet 1990] for a given dataset. Examples of such structures include octrees, k-d trees, BSP trees, trees of bounding boxes, and trees of bounding spheres. These structures have been used in many different contexts to speed up geometric algorithms. For example, if we wanted to find the closest object hit by a ray without a spatial hierarchy, we would have to test every object in the scene. If we have a spatial hierarchy, when a ray does not hit a certain node in the hierarchy, we know that the ray will not hit any of the descendants of that node either, so we can prune that entire subtree.

Spatial data structures can also be used for out-of-core rendering. The size of the structure of the hierarchy is typically tiny in comparison to the size of the contents of the nodes in the hierarchy. As long as we can hold in RAM the structure of the hierarchy and the contents of a single node, we can render an arbitrarily large dataset by swapping nodes in and out of memory.

SPVN provides a tool called *spvn_spatialize* that creates a spatial hierarchy for a given dataset. The tool uses out-of-core techniques to support arbitrarily large datasets (limited only by the size of secondary memory, not RAM). The tool recursively splits spatial nodes until all spatial leaves satisfy a given limit on the number of geometric primitives per leaf. For example, it is useful to limit the number of vertices of indexed primitive arrays to be less than 64K, so that indices can be stored in 16-bit integers. Currently SPVN supports hierarchies of boxes and k-d trees.

3.2 Simplification

Another optimization technique commonly used to deal with large datasets is simplification, which consists of precomputing and rendering approximate versions of the dataset. These approximations are known as levels of detail (LODs) [Luebke et al. 2002], and can be discrete, continuous, or view-dependent.

Approaches based on discrete (or static) levels of detail, precompute a sequence of several aproximations of the dataset, each at a different resolution. At runtime, one of the approximations is picked to be rendered based on some selection criterion such as distance to the viewer. Discrete levels of detail are the easiest to compute, and the fastest to render, but they may produce objectionable popping artifacts when switching from one level to another.

Approaches based on continuous (or progressive) levels of detail precompute a multi-resolution representation of the dataset so that transitions between approximations at runtime are smooth. Continuous levels of detail produce far fewer popping artifacts than discrete levels of detail, but they are harder to compute, and they impose a higher runtime overhead.

Approaches based on view-dependent levels of detail also precompute a multi-resolution representation of the dataset. In addition, these approaches allow a single object to have multiple levels of detail in the same frame. Parts of the object that are closer to the viewer are drawn in higher detail than parts farther from the viewer.

SPVN provides a tool called *spvn_simplify* that precomputes static levels of detail using a variant of vertex clustering. At runtime, SPVN selects the appropriate level of detail for a node based on either the distance between the node and the viewer or the number of pixels that the node's bounding box is visible from.

3.3 Visibility Culling

Visibility culling techniques avoid unnecessary rendering computation by not sending to the graphics pipeline those parts of the dataset that are not visible from the current viewpoint. We discuss three types of visibility culling techniques below: back-face culling, view-frustum culling, and occlusion culling.

Back-Face Culling

Back-face culling means not rendering geometry that faces away from the user, as determined by the current viewing direction and the normal of the face in question. Testing this condition is very easy in software and consists of a single dot product between the two vectors. If the geometry is found to be back-facing, it should be culled from the graphics pipeline; otherwise, it is front-facing and may be visible. Applications that use OpenGL can turn back-face culling on by simply enabling the GL_CULL_FACE flag.

View-Frustum Culling

View-frustum culling means not rendering geometry that is outside the viewer's field of vision. Testing whether individual faces are within the current field of view would be very time consuming for large models; therefore, hierarchical spatial data structures are often used to speed up view-frustum culling by testing large subsets of faces against the frustum in one simple test. If the bounding volume of the subset of faces is completely outside the frustum, all of the geometry must be outside as well (and can therefore be culled). Conversely, if it is entirely inside the frustum, none of the geometry can be culled. If the bounding volume partially overlaps the frustum, we recurse to the next level in the hierarchy and test again.

Occlusion Culling

The final culling technique, and the most difficult to implement efficiently, eliminates geometry from the graphics pipeline that is completely hidden from the viewer by other geometry, i.e., geometry that is completely occluded. Visibility algorithms may be classified according to several criteria [Cohen-Or et al. 2003]:

From-point vs. from-region: Some algorithms compute visibility from a single viewpoint, while others compute it from a (connected) region in space. From-region algorithms can amortize the cost of the visibility tests over a number of frames as the viewpoint will typically reside in a single region for a while.

Precomputed vs. online: Many visibility algorithms preprocess the model to provide greater interactivity during runtime, while others require no precomputation step and are said to work on the fly (or online). For example, from-region algorithms require a preprocessing step to divide the model in regions and compute region visibility. From-point algorithms, on the other hand, typically compute visibility at runtime.

Object space vs. image space: Algorithms that compute visibility using the 3D primitives are said to run in object space, while others that utilize the discrete rasterization fragments of the primitives are said to run in (2D) image space.

Conservative vs. approximate: Computing the exact set of visible geometry is very difficult and time consuming. Therefore, most algorithms overestimate the set of visible primitives but guarantee not to miss anything that may be visible. These algorithms are called conservative. In some cases, these algorithms may not be fast enough. Therefore, others have implemented approximate visibility algorithms, which are faster but do not guarantee to render all of the visible geometry.

SPVN today has two back-ends: one based on ray tracing and another based on OpenGL. Both back-ends use from-point, online, conservative visibility. The difference between them is that the ray tracing back-end operates in object space, while the OpenGL back-end operates in image space.

The OpenGL back-end takes advantage of the NVIDIA occlusion query extension, which allows us to send various objects to the graphics hardware at the same time, and get for each one of them the number of pixels that it is visible from.

3.4 Caching

A critical part of any system for large data visualization is the memory management subsystem. SPVN keeps in a cache in main memory the most recently used objects. This approach is particularly effective if the objects that are visible in any given frame fit together in the cache, and there is locality of reference, i.e., the changes in visibility from frame to frame are small.

SPVN implements caching by using the proxy pattern [Gamma et al. 1995], and swaps parts of the scene in and out memory automatically when necessary. The programmer does not have to make any explicit requests to the cache, and thus has the illusion that all the scene is in memory.

3.5 Multithreading

Caching alone is typically not enough to deliver smooth frame rates. Even small changes in the user's camera may correspond to big changes in visibility, which may lead to many cache misses, causing the application to stall because of bursts of disk or network activity.

To alleviate this problem, SPVN uses multithreading. For example, in the OpenGL back-end, the visibility culling thread pushes nodes onto a fetching queue. A set of fetching threads pop nodes from that queue, make sure they are in memory, and push them onto a ready queue. The rendering thread pops nodes from the ready queue in the order that they become available, and renders the ready nodes while the fetching threads validate the other nodes. Thus, rendering and fetching happen in parallel, which helps hide I/O latency.

3.6 Prefetching

A technique to reduce cache misses is speculative prefetching, which tries to bring into memory the objects that may become visible soon. What is considered soon may be difficult to define. We want to have the objects in which we are interested ready in memory when we need them, but we also want to avoid polluting the cache with too many objects that will end up not being used.

Traditional prefetching techniques [Funkhouser et al. 1992] use precomputed, from-region, conservative visibility algorithms. Such techniques partition the dataset into cells, and precompute the cells that can be visible from within each cell. This preprocessing step may be very time-consuming. At runtime, whenever a user enters a cell, all other cells potentially visible from that cell are prefetched. This strategy may pollute the cache because it is too conservative.

SPVN also employs prefetching, but using an online, from-point approximate visibility algorithm. Based on the recent history of camera motion, a separate thread guesses cameras for the next few frames, estimates the nodes that these cameras would see, and prefeches these nodes. Because the visibility algorithm is online, it requires much less preprocessing time. Because it is from-point, it tends to cause less cache pollution. And because prefetching is speculative in nature, an approximate algorithm is appropriate.

4 Support for Parallelization

Researches have long investigated the use of parallel machines for computer graphics. Parallelization strategies can be classified in three categories based on where in the rendering pipeline sorting for visible-surface determination takes place [Molnar et al. 1994]. Sorting may happen during geometry preprocessing, between geometry preprocessing and rasterization, or during rasterization. The three categories are sort-first, sort-middle, and sort-last:

Sort-first algorithms partition the 2D screen into tiles, and assign each tile to a rendering processor, which renders all of the objects that are visible in the tile. At every frame, the tiles in which each object falls are determined, and the objects are redistributed to the appropriate tiles. Since few objects tend to move between tiles from one frame to the next, sort-first algorithms typically benefit from frame-to-frame coherence. In addition, sort-first algorithms are not dependent on a particular type of rendering technique, because all of the necessary objects are available on each rendering processor. However, sort-first algorithms become inefficient if objects begin to cluster on a single rendering processor, leading to a rendering load imbalance. Furthermore, the potential for redundant work exists, because objects may overlap multiple tiles.

Sort-middle algorithms distribute screen-space primitives after the geometry preprocessing, but before the rasterization stage. A sort-middle algorithm assigns a subset of the scene's 3D objects to each geometry processor. In addition, the 2D screen is partitioned into tiles, and each tile is assigned to a rasterizer. The geometry processors perform all of the transformation and lighting calculations on their set of objects, and then send the transformed objects to the appropriate rasterizers. As with sort-first algorithms, sort-middle algorithms become inefficient if the objects are unevenly distributed across the screen. Furthermore, sort-middle algorithms require access to the rendering pipeline in between the geometry processing stage and the rasterization stage, as well as high bandwidth for the transfer of data between these stages.

Sort-last algorithms assign a subset of the dataset to a a rendering processor, which renders a full-resolution partial image for its subset. The resulting pixels (color, depth, and possibly alpha values) are sent to compositing processors, which create the final image. Because each object is rendered exactly once, sort-last algorithms scale well with dataset size. However, due to the large amount of pixel transfers, a high-bandwidth connection among compositors is critical. In addition, most compositing algorithms impose a strict ordering for compositing each of the partial images. This ordering may make it difficult or impossible to use certain rendering techniques such as anti-aliasing or rendering transparent objects.

SPVN currently supports the sort-first and sort-last approaches. The sort-first implementation employs dynamic load balancing techniques to avoid having application servers idle or overwhelmed. The application servers continuously feed the controller with performance statistics, and the controller recomputes the screen tiles whenever it is deems appropriate.

The sort-last implementation provides users with a preprocessing tool to partition large volumetric datasets into cubes or slabs (Figure 3). At runtime, each application process renders its partition, generating a partial image.

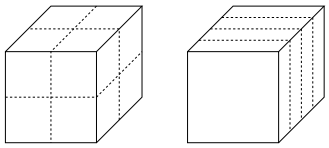


Figure 3: Volumes can be partitioned into cubes or slabs.

To generate the final image, the user has two options. One is depth-order compositing (Figure 4). In this case, all renderers send their partial images to the compositor, which composites corresponding pixels in back-to-front order. As the number of renderers (and therefore partial images) increases, the compositor may become a bottleneck. In addition, the renderers are idle during composition.

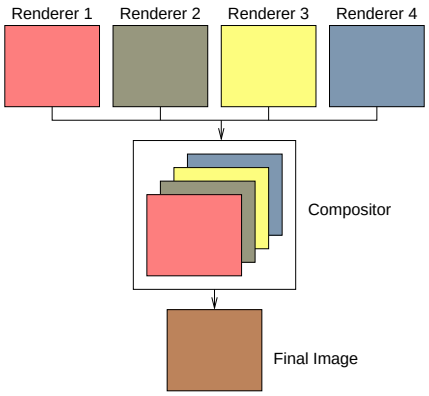


Figure 4: Depth-order compositing.

The other (and better) option is binary-swap compositing [Ma et al. 1994], which makes use of all the renderers during the composition phase (Figure 5). Each renderer generates its partial image and exchanges a subimage with another renderer. Each renderer then composites the subimages and sends a subimage of the result to another renderer for compositing. This process continues until each renderer has a subimage that is a composite of all the partial images.

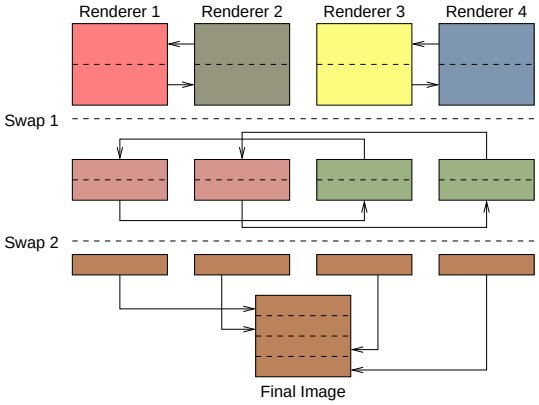


Figure 5: Binary-swap compositing.

Besides support for sort-first and sort-last rendering, SPVN provides a distributed shared memory mechanism that gives programmers the illusion that all the machines have all the data at all times. In reality, each machine only has the most recently used objects in its cache. Whenever an application process modifies an object, the database manager process (Figure 2) invalidates the corresponding cache entry in each of the other application processes. If an application process tries to use an object that is marked as invalid in the cache, SPVN will, without the process's knowledge, get that object from another application process that has a valid copy of the object, or from the file system.

Fetching an object happens transparently, which simplifies programming, but programmers should still be aware that the fetch time will vary depending on the location of the object. To minimize the effects of possible latency, application writers should use multithreaded object queues whenever possible.

5 Support for Multiple Rendering Libraries

Many popular visualization toolkits couple modeling with rendering, creating a dependency on a particular low-level rendering library (e.g., OpenGL, DirectX, or ray tracing). Unlike these toolkits, SPVN keeps modeling and rendering separate, allowing developers to use whatever low-level rendering library is more appropriate for their computing platform or better matches the desired level of interactivity or realism. This separation also allows users to have the same scene drawn by multiple rendering back-ends, which can even coexist at runtime.

To make that possible, the scene graph classes in SPVN do not know how to draw themselves. Instead, SPVN defines an abstract *Renderer* class with an interface for rendering. Different back-ends create concrete renderers by inheriting from the abstract class and providing definitions for the interface.

In addition, SPVN defines *Action* classes that encapsulate scene graph traversal. Concrete renderers define rendering actions that, when visiting different types of scene graph nodes, will call the appropriate low-level rendering functions.

Finally, SPVN defines the *Subject* and *Observer* classes. A subject allows observers to attach themselves to it. When the subject is modified or deleted, it notifies all of its observers. Scene graph nodes are derived from *Subject* and concrete renderers use observers to respond to changes in the scene. For example, the OpenGL back-end uses observers to manage the creation and destruction of display lists and textures.

6 Usability and Extensibility

When designing SPVN, one of our main goals was to make it easy to use and extend. To address usability, we designed SPVN around the familiar concept of a scene graph, which is used by virtually all popular visualization toolkits. In particular, SPVN provides classes for many typical scene graph nodes such as shape, material, light, and group. At the same time, SPVN provides clean interfaces to encapsulate concepts such as sort-first and sort-last rendering, which are less common and differentiate SPVN from other toolkits. We expect that programmers who are proficient in other visualization toolkits will face a gentle learning curve when switching to SPVN.

Another effort to improve usability was the use of well-established design patterns [Gamma et al. 1995] throughout SPVN. For example, the *Shared_ptr* class, which provides an elegant and robust implementation of reference counting, uses the proxy pattern. Of the classes mentioned in Section 5, *Renderer* is an example of the strategy pattern, *Subject* and *Observer* implement the observer pattern, and *Action* is an example of the visitor pattern. A set of stream classes use the decorator pattern. In our experience, using these patterns makes the code more understandable, and allows developers to communicate using a concise and accurate vocabulary.

To address extensibility, SPVN makes it easy to add a new rendering algorithm, a new type of shape, or support for a new file format. To change the rendering algorithm for an existing shape, the user would create a new rendering action, redefine the method corresponding to the shape, and register the new rendering action with SPVN. To create a new shape, the user would create a class directly or indirectly derived from the *Shape* class, create a rendering action that knows how to draw the new shape, and, as in the previous case, register the new rendering action. To add support for a new file format, the user would create a class derived from *Reader*, and create a plug-in library with a specific name (for some XYZ format, the name would be *libxyz_io.so*) that, when loaded, registers the new reader with SPVN.

7 Results

This section presents three sample applications that were written using SPVN. Figure 6 shows a screenshot of a sort-first viewer rendering the UNC powerplant. On a cluster of 9PCs, SPVN can render 20-megapixel images of this dataset at 60Hz.

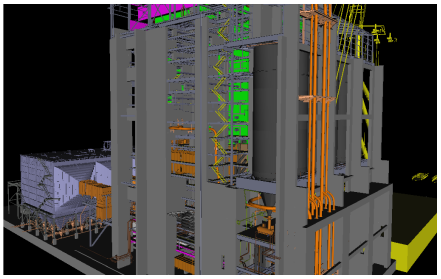


Figure 6: Powerplant dataset rendered by the sort-first viewer.

Figure 7 shows two screenshots of an application that can switch between the OpenGL and ray tracing back-ends at runtime. The ray tracing back-end is not optimized yet, but it shows the separation between modeling and rendering in SPVN.

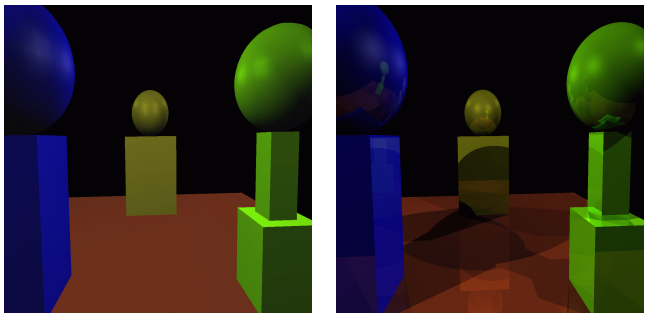


Figure 7: Users can switch from OpenGL to ray tracing at runtime.

Figure 8 shows the a screenshot of a sort-last viewer rendering the skull dataset. On a cluster of 8 PCs, SPVN can render 1-megapixel images of this dataset at 60Hz.

Figure 8: Skull dataset rendered by the sort-last viewer.

8 Conclusion

SPVN is a new framework for developing interactive visualization applications. What differentiates SPVN from other frameworks is the combination of all of the following features:

- Support for datasets larger than main memory
- Support for immersive and remote visualization
- Support for surface and volume datasets
- Ability to run on clusters of inexpensive PCs
- Built-in optimizations for interactive rendering
- Support for sort-first and sort-last parallel rendering
- Support for different low-level rendering libraries
- Ease of use and extension

There are many ways in which we plan to improve SPVN. First, we intend to support other types of spatial data structures and other simplification algorithms. Second, we plan to support lossy image compression with adjustable quality, similar to RVN [IBM 2007b]. Third, we want to extend and optimize the ray tracing back-end. Finally, we would like to port SPVN to other platforms including Windows and Blue Gene [IBM 2007a].

Acknowledgements

Thanks to the Boeing Company (especially David Kasik) for the 3D model of the 777 airplane. Thanks to the University of North Carolina at Chapel Hill for the powerplant model. Thanks to Matei Ciocarlie and Carlos Scheidegger for their contributions to SPVN.

References

BARTZ, D., STANEKER, D., STRAßER, W., CRIPE, B., GASKINS, T., ORTON, K., CARTER, M., JOHANNSEN, A., AND TROM, J. 2001. Jupiter: A toolkit for interactive large model visualization. In *Symposium on Parallel and Large Data Visualization and Graphics*, 129–134.

BURNS, D., AND OSFIELD, R., 1998. OpenSceneGraph. www.openscenegraph.org.

COHEN-OR, D., CHRYSANTHOU, Y., SILVA, C. T., AND DURAND, F. 2003. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9*, 3, 412–431.

CORRÊA, W. T. 2004. *New Techniques for Out-Of-Core Visualization of Large Datasets*. PhD thesis, Princeton University.

FUNKHOUSER, T. A., SÉQUIN, C. H., AND TELLER, S. J. 1992. Management of large amounts of data in interactive building walkthroughs. *1992 ACM Symposium on Interactive 3D Graphics*, 11–20.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GROPP, W., LUSK, E., AND SKJELLUM, A. 1999. *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd ed. The MIT Press.

HENNING, M., AND VINOSKI, S. 1999. *Advanced CORBA Programming with C++*. Addison-Wesley.

IBM, 2007. Blue Gene. www.research.ibm.com/bluegene.

IBM, 2007. Deep Computing Visualization. www.ibm.com/servers/deepcomputing/visualization.

LUEBKE, D., REDDY, M., COHEN, J. D., VARSHNEY, A., WATSON, B., AND HUEBNER, R. 2002. *Level of Detail for 3D Graphics*. Morgan Kaufmann.

MA, K.-L., PAINTER, J., HANSEN, C., AND KROGH, M. 1994. Parallel volume rendering using binary-swap compositing. In *IEEE Computer Graphics and Applications*, vol. 14, 59–68.

MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. 1994. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4, 23–32.

REINERS, D., VOß, G., BEHR, J., ROTH, M., AND ZIERINGER, A., 1999. The OpenSG scene graph system. www.opensg.org.

SAMET, H. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.

SCHROEDER, W., MARTIN, K., AND LORENSEN, B. 1996. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall.

SGI, 2003. OpenGL Performer. www.sgi.com/products/software/-performer.

TGS, 2003. Open Inventor. www.tgs.com.

7 Visibility-guided Rendering to Accelerate 3D Graphics Hardware Performance

Siggraph Course Notes:

Section 2: 11:20 – 12:15 Visibility-guided Rendering to Accelerate 3D Graphics Hardware Performance

Beat Bruderlin, Technical University of Ilmenau, Mathias Heyer, Sebastian Pfützner 3DInteractive GmbH

Abstract: Visibility-guided Rendering (VGR) is an output-sensitive approach for real-time rendering of very large 3d data sets, using a GPU. The goal of this section is to understand the foundations of VGR and its application to visualization of complex scenes, using hardware-accelerated OpenGL. The key issues covered in this section are hardware-based occlusion culling, memory management and out-of-core visibility determination, the data structures and algorithms as well as low-level operating system issues.

1 Overview: The Interviews3D Platform

In this section we present the algorithmic and technical background information on Interviews3D, a new Digital Mockup (DMU) system and library for large-scale CAD models, consisting of several hundred million or even over one billion polygons. In the aerospace, automotive and shipbuilding industry large construction models routinely have to be simulated and visually inspected. Similarly, in digital factory planning design reviews, walking through virtual buildings, clash detection and comparing versions of factories are carried out.

To harness the data explosion problem encountered in the industry a serious redesign of the systems software architecture became necessary. Foremost, an out-of-core scene representation with a two-level cache mechanism running on PCs has been developed, with tens or hundreds of gigabytes of disk space, that don't fit into 32-bit addressable memory space. Interviews3D is based on an output-sensitive approach, capable of rendering very large scenes for interactive VR-applications in real time on standard PCs and laptop. In addition

to scene navigation, object picking and manipulation, animation, as well as clash detection are supported by the system.

The following is a summary of main approaches and principles that have been employed by the Interviews3D platform to cope with massive data sets:

Output sensitivity: For output sensitive approaches, system performance is not so much governed by the amount of input data (the problem size), but on the size of the resulting data (output) of the problem's solution. The visibility-guided rendering approach described in the next section is such an approach, as it only needs to deal with the data visible from a specific point of view. This is in most situations (especially for large data sets) only a very small fraction of the total data.

Out-of-core: With out-of-core technology data do not need to reside in main memory but can be loaded from the hard disk on-the-fly. The application has full control over the memory management and caching of data. The out-of-core approach works hand-in-hand with the output-sensitive algorithms (in contrast to standard virtual memory mechanisms of the operating systems which are not controlled by the application).

Temporal coherence: The output generated by visibility-guided rendering usually does not radically change over time, as subsequent frames show mainly the same data. By exploiting temporal coherence, we can reduce the amount of work that needs to be done in each frame.

Selective updates: Rather than solving a problem from scratch every time the input data changes, incremental (selective) updates only need to deal with the actual changes, when they occur. Again, this is often only a fraction of the whole data. Spatial and temporal coherence of interaction can be exploited for updates.

Scalability: Modern platforms should be scalable for single processor PCs to multi processor computers, networked compute clusters, and even exploit multiple graphics cards in parallel. Future applications have to be not only out of-core but also distributed data all over the world-wide network.

2 Visibility-guided rendering

Visibility-guided rendering is an output-sensitive approach, which is motivated by the following observation: The average computer monitor is composed of about 1 million picture elements (pixels). A typical massive model contains 100 million or even over a billion polygons. This means, statistically, only one in one hundred or one in a thousand polygons cover at least a pixel of the screen. The other (over 99%) of data are invisible. Either these polygons are much smaller than a pixel and too small to contribute to the

polygons are much smaller than a pixel and too small to contribute to the image, or they are hidden by objects in front of them, or they are outside the view frustum.

Most current visualization tools rely on the graphics hardware to process all the polygons, convert them into pixels (rasterization), and only at the end determine their visibility on a per-pixel base. The notion of visibility-guided rendering implies determining the visibility of large parts of the model before sending them to the graphics card, or even before loading them into main memory from the disk. As a consequence, the graphics card has much less work to do and can focus on the essentials: Rendering objects in motion at a high frame rate. By using spatial data structures and queries we can quickly determine the potential visibility of large subsets of all polygons in the database, even before they are loaded into core memory. Those objects in front (close to the camera) have the highest priority. A visibility test can already be performed for objects not yet rendered in the current frame. A second-level caching mechanism loads the visible parts onto the graphics card (V-RAM). For this task we use a spatial index data structure in combination with well-known culling techniques [AM00, BWPP04, HeBr04, HeBr04b, Sta03]. View frustum culling determines large portions of polygons that are outside of the view frustum, and therefore with certainty cannot be seen by the camera. Detail culling can be used as a simple means for culling objects that are invisibly small

A hardware extension (ARB_occlusion_query, supported by OpenGL 1.5 [nVi03]) determines visibility of objects without having to draw them into the frame buffer. This test is used to determine groups of polygons that are guaranteed to be hidden. This process simultaneously uses the graphics card to determine the visibility of convex hulls of groups of objects which themselves are still stored in the database: The CPU controls the overall caching process, and loads objects into main memory or into the video RAM of the graphics card. Using efficient spatial algorithms and data structures visibility-guided rendering significantly reduces the load on the graphics card.

In comparison: In the conventional way of rendering every polygon is converted into pixels of the screen and finally the visibility is determined on a per-pixel basis. Therefore the frame rate tends to decrease approximately linearly with the complexity of the scene (defined by the number of polygons. See fig. 1). This does not yet take into consideration additional slowdown when data no longer fits in the graphics memory and needs to be transferred in each frame. With visibility-guided rendering, on the other hand, performance (similar to ray tracing [WDS01]) only decreases approximately logarithmically with the number of polygons. Therefore, large models with over one billion triangles can still be rendered in real time (20 fps), whereas classical OpenGL rendering would be limited to a few million polygons (see figure 2). In practice, visibility-guided rendering results in an almost 1000-fold increase of data size which can still be handled interactively.

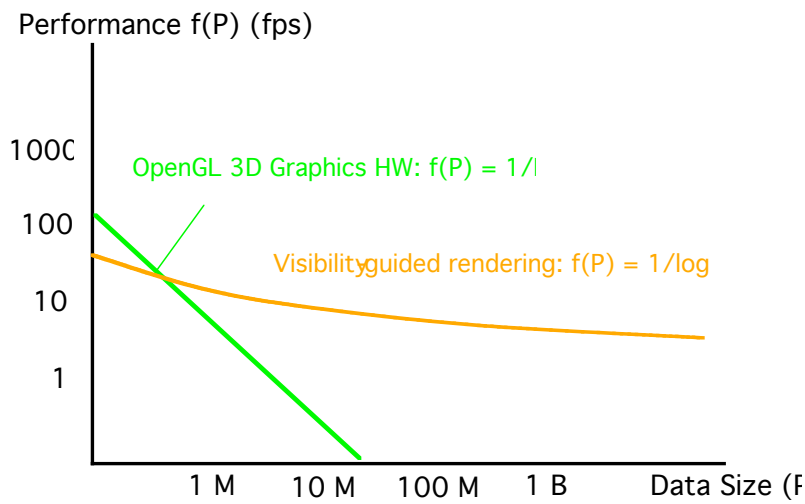


Figure 1: Performance comparison of different rendering approaches: a) Straight-forward scene graph rasterization with OpenGL, b) Visibility-guided rendering.

2.1 Representing Objects and Spatial Hierarchies

One of the main differences of visibility-guided rendering, in contrast to classical scene graph-based rendering is that the rendering algorithm is not guided by the logical structure of the scene graph, but rather rendered in an independent ordering scheme that is optimal for render performance. In Interviews3D the object structure is superimposed upon the internal rendering structure to maintain the render performance for large data sets, as well as support the complete functionality that we expect from standard scene graph-based tools.

Our approach applies an *Axis Aligned Bounding Box Tree* (AABB-Tree, see fig. 2), which, like a regular kd-Tree for each node stores an additional splitting plane (axis identification plus coordinate). The bounding boxes are used directly as test geometry for the occlusion query. In addition to geometry an object structure and material data is stored with the geometry. The splitting information is used to traverse the tree in front-to-back order, as much as possible. If a bounding box is hidden by geometry in front of it, then all polygons inside the box (and all bounding boxes inside the test box) are hidden as well, and don't need to be rendered or tested.

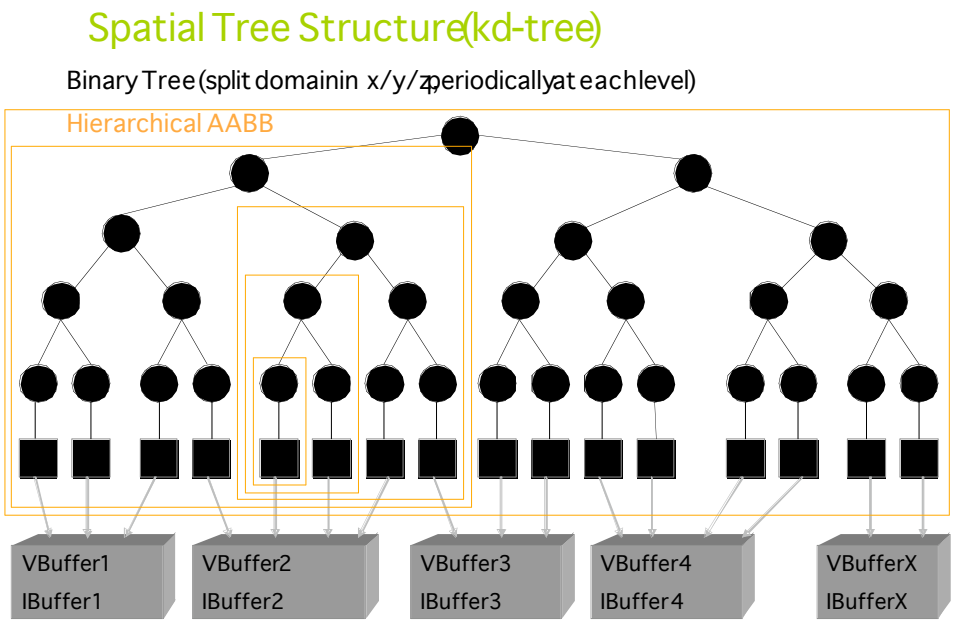


Figure 2: The spatial data representation of large scenes as a kd-tree with bounding boxes and vertex buffer objects.

To avoid micromanagement of polygons during the tree construction, polygons are neither split nor referenced twice. We therefore accept that bounding boxes of child nodes may slightly overlap. The tree is generated from the top down. The heuristics tries to keep the side lengths of bounding boxes as equal as possible to balance the culling effect for all view angles. The subdivision process ends, if the number of polygons in a node falls below a threshold that depends on the hardware, the size of the scene, and the shading function applied, and it directly influences the tree size, the batch size [Spi03], the number of occlusion queries per frame, and the number of polygons rendered per frame.

2.2 Queueing of Occlusion Queries

Ideally the occlusion queries are carried out on a front-to-back, top-down order (see fig. 3, top left).This way we detect early-on whether large parts of the scene are hidden by previously rendered objects in front of them. However, the results of an occlusion query are passed to the CPU with some delay. Synchronizing occlusion tests with rendering would paralyze the graphics hardware and cause a so-called pipeline stall. The GPU is very effi-

cient, only when it can render a lot of geometry in a without interruption of the pipeline. Therefore, our occlusion culling approach runs occlusion queries in parallel to the tree traversal and rendering process. ARB_occlusion_query allows for asynchronous requests of the results. We devise a (first-in-first-out) query queue with a length that is dynamically adapted to the load of the graphics card. Ideally the result of the oldest query becomes available exactly when requested by the render command. The program can subsequently start an additional query. We therefore adapt the queue to the actual latency time at run time.

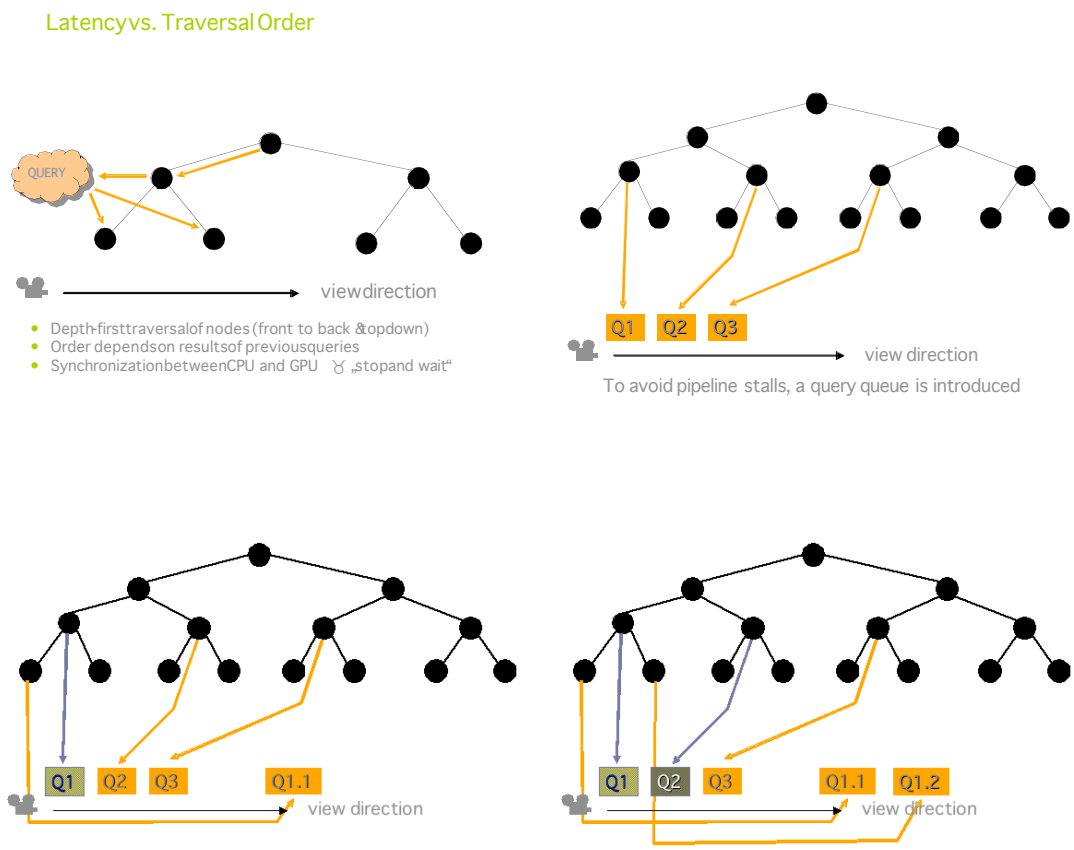


Figure 3: Scheduling occlusion queries. A priority queue is employed to avoid lag times during rendering. The order of visibility checks is no longer depth-first, front-to-back but a mixture of breadth-first and front-to-back

This departure form a strict depth-first traversal means that we no longer guarantee a perfect occlusion result. Parts of the scene will be rendered before the occlusion by objects in front of them can be tested. This deficiency is offset by the fact that without pipeline stalls

graphics cards can render much most efficiently when the queue length is only slightly larger than necessary to avoid stalls.

2.3 Frame-to-Frame Coherence

Since the z-buffer of the graphics hardware represents the occluder geometry, it is necessary to fill the z-buffer before the first occlusion query of potentially occluded objects for frame i "reasonably." Otherwise the first occlusion tests would run empty. We found that the visibility of nodes does not change drastically from frame to frame. Therefore it is more efficient to first render all the nodes that were visible in the last frame i-1. The visibility test of previously visible nodes does not need to be repeated for each frame to achieve optimal performance (see figure 4). After a change in camera view angle, at each frame some previously visible nodes may become hidden and will become marked as "invisible" eventually. This way the number of occlusion queries and the computational overhead for them is kept constant between frames. Either a node is marked as "visible", "invisible" or "untested." This attribute is propagated bottom up from the leaves within the kd-tree. This way unnecessary occlusion queries can be avoided as complete tree branches don't have to be tested for visibility, which yields drastic performance improvement.

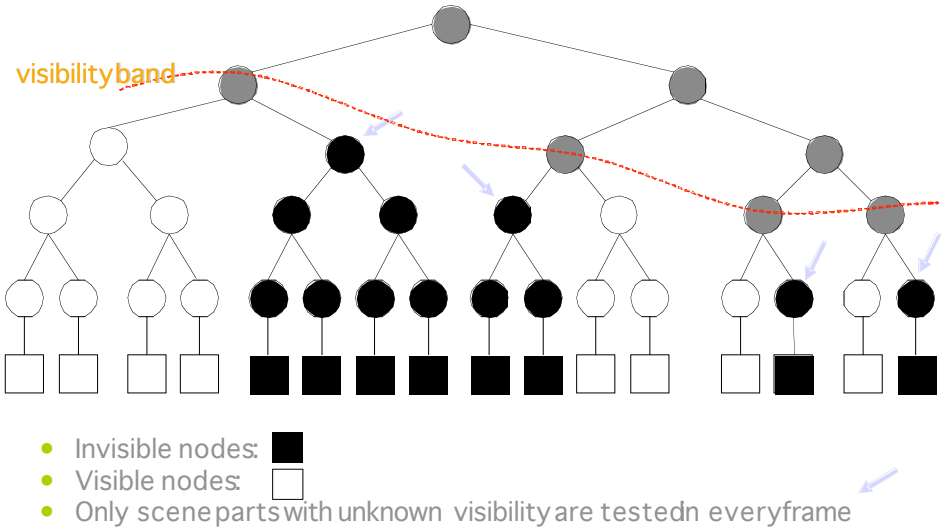


Figure 4: Exploiting temporal coherence- Making the undecided nodes for testing in the next frame. The leaves marked black were invisible in frame i-1. They need to be tested in every successive frame. The white nodes were visible in the previous frame. They will be tested for visibility only every n-th frame. The gray nodes have no information on previous visibility. The illustration shows the bottom-up propagation of leaf status.

Occlusion queries are necessary in each frame only for the black nodes, as their status might have changed to "visible" since the last frame. For these nodes, the tree is recursively traversed front-to-back and top-down using the view frustum and occlusion tests. When a black node is met, an occlusion test is immediately carried out, and recursion is stopped. For white nodes the recursion is also aborted, as these nodes have already been rendered. This very consistent initialization of the query queue generates a large pre-run and thus avoids pipeline stalls.

2.4 Simple LOD Using Point Rendering

To attribute for the cases where many, very small polygons are visible but each covers less than a pixel of the screen space, we apply a very simple LOD approach, based on point rendering. During preprocessing, we also test the size of the largest triangles in each leaf node. Before rendering a visible leaf node we estimate the potentially projected area (in number of pixels on the screen) of the largest triangles of a leaf node. If this projected area is less than one pixel, we resort to point rendering for this node. This guarantees a visually correct image. If for distant leaf nodes the projected area of the triangles becomes even smaller, we can thin out the point cloud, by only rendering a fraction of the triangles, proportionally to the screen area covered. This simple thinning is not completely accurate and may show aliasing effects, particularly if the triangles of the node have vastly different reflection properties and color. The advantage is that it works fine for most continuously shaded areas with almost no additional preprocessing effort, and it dramatically increases the performance of large models with extremely fine resolution. Generally speaking, point rendering fits seamlessly into the occlusion rendering pipeline, and we can take advantage of a vast amount previous results in this area (see e.g. [RuLe00]).

2.5 Out-of-core Data Handling

For interactively handling objects of a very large scene, as well as for real-time rendering a spatial database is generated. Since the 3D scenes can be much larger than the available memory of the computer, the database is also generated by an out-of-core preprocessing. More on pre-processing can be found further below.

During navigation, the render application loads the parts in a prioritized order onto the graphics card.

A 2-level cache mechanism with two asynchronous threads is used to transfer data from the hard disk to main memory and from there to the video ram (VRAM) of the graphics card (see fig. 5).This mechanism is tightly integrated into the visibility guided rendering approach, described above. A pre-fetching mechanism tries to load the date most likely visible in the next frame.

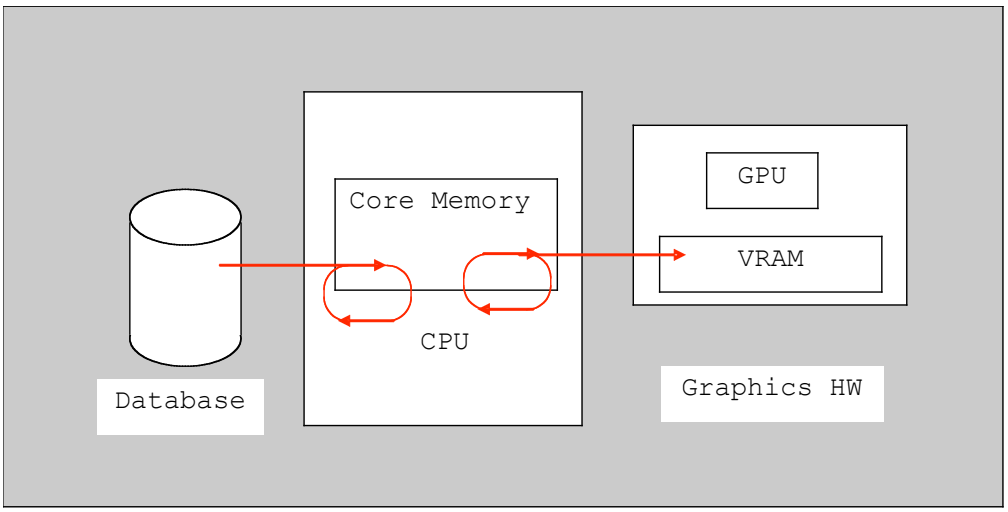


Figure 5: Caching geometry in core and on the graphics card (VRAM)

Fast spatial queries can be enabled by memory-coherent data organization. The latency during data access on disk can be reduced by pre-fetching. Latencies during rendering of frames are not acceptable. Therefore asynchronous loading of geometry is an important requirement. This results in approximate (preview quality) rendering during load times. The goal is the fastest-possible image development during loading of geometry from the database by means of adequate loading, replacing and prefetching strategies. The leaves marked "visible" (see section on temporal coherence) have the highest priority. Next come the hidden parts of the scene that are within the extended frustum. A low priority is attributed to

scene parts that are far outside of the view frustum, followed by leaf nodes below the so called LOD band, which have the lowest priority. – The LOD band is, similar to the visibility band (see above), a list of sub-tree nodes that contains details that are smaller on the screen an adjustable threshold (e.g. 1 pixel). This way we can prioritize parts of the scene that have the biggest visual impact and load small details later. Whenever there is time, additional scene parts are loaded to preempt future camera motion.

In each frame, there is potentially a different distribution of priorities. However, due to the assumed temporal coherence from frame to frame, only a few changes need to be applied simultaneously.

The prefetching-and-replacement strategy tries to keep as many leaf nodes as possible in main memory, starting with the leaves with the highest priority. Leaves with lower priority that have stayed in memory for the longest time will be removed from the systems memory first, such that only the data that is important for the actual view is kept in memory. Leaf nodes marked as "visible" will never be removed from memory, to avoid flickering.

Typically for 32-bit PC configurations the main memory has 0.5 – 2 Gigabytes, at the most. Large models are several tens of gigabytes large, and therefore do not fit into the main memory of average PCs. The Video RAM is even much smaller. Typically, 3D graphics cards have 64 MB – 256 MB of VRAM. Also the visualization application may coexist with other programs or modules of the application systems which also may use large parts of available memory. Therefore the actually usable physical memory of this render system can be configured to only use only 300 MB of main memory and down to 60 MB VRAM (this his way older graphics cards can be supported).

2.6 Supporting Interference Detection for Large Scenes

Clash detection is a very important tool for early detection of interferences between building components within a 3-dimensional digital mock up, e.g. before the construction of the physical artifact is begun. This way, enormous cost savings at the building site can be realized through digital factory planning. The task is generally carried out by comparing groups of objects with specific functionality (called works or layers) with each other. E.g. one may compare structural data (steel beams) against water pipes, or air ducts against electric wires, etc. On the other hand, it is often considered unnecessary to compare objects within a group (e.g. steel beams against each other) because they normally already have been tested for consistency beforehand, since they come from the same supplier. An anew testing would generate unnecessary false collisions which turn out to be permissible in the end (e.g. where steel beams are connected to each other).

The clash detection takes advantage of the out-of core data representation to determine potentially colliding parts, already in the database, before loading objects into main memory. The caching mechanism described above is employed for this task as well. Only if a preliminary test determines potential clashes, the corresponding part of the model is loaded into main memory and an exact test is carried out on a polygon-by-polygon base.

The clashes are returned as pairs of objects, sorted by severity (usually determined by the approximate penetration depth.) This way, the most severe clashes can be handled and eliminated with high priority, before secondary tests are performed.

The interferences detected by the clash module can be visualized interactively one after another, as it is shown in figure 6, below. A wireframe sphere indicates the location of the clash graphically. Automatic camera control leads the user to the next clash in the list.

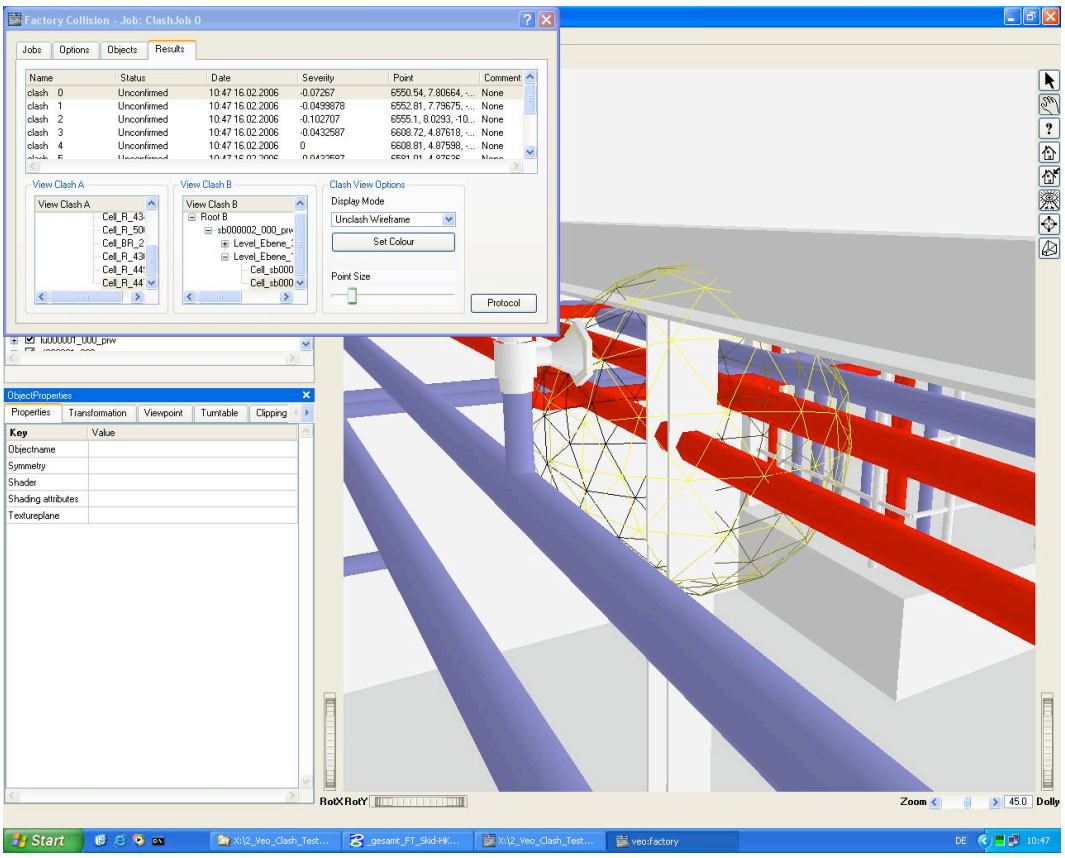


Figure 6: Screen shot presenting the results of clash detection to the interactive user. A list of pairs of colliding objects can be sorted by severity and then visually inspected one-by-one.

2.7 Preprocessing: Building the scene database for real time data handling

The task of pre-processing is to hierarchically and spatially subdivide the scene into sets of roughly 1000 to 8000 triangles, which are stored as the leaves of a kd-tree. This data structure enables the efficient data access for VGR frustum culling, detail culling and occlusion culling as described at the beginning of this paper (see fig. 2).

For VGR pre-processing mostly consists of a simple spatial sorting. No complex data reduction schemes or progressive mesh generation is necessary during this step.

The top-down generation of a kd-tree for very large scenes requires an out-of-core strategy. Parts of the geometry data potentially are used several times during the tree generation in different places of the algorithm. Therefore, in principle, data needs to be loaded multiple times. This causes problems with swapping data in and out of systems memory and fragmentation, which are addressed in the next sections. Properly addressing or avoiding these problems leads to a faster pre-processing. Theoretically the pre-processing time grows asymptotically with time $n \cdot \log n$.

In practice, the pre-processing time is comparable to data conversion and copying time of other systems. With the new “selective update” functionality (described below), the database can be updated interactively.

2.8 Selective updates: Incremental building of the model database

In real world applications especially for the very largest models the data does not change radically on a regular basis. Rather the changes are often gradual. Interviews3D in its latest release no longer needs to rebuild the database for every change, but allows for in-place changes. To support selective updates the spatial data structure (k-d tree) needs to be locally restructured by expanding some leaf nodes or merging two neighbors when their total sum of data is smaller than the size of a node. The performance analysis below shows that selective updates can be carried out at a fraction of the cost of a complete rebuild.

3 Operating System Related Issues

In this chapter we discuss some of the low-level operating system issues that need to be considered to make performance optimal for large data sets. In the following we discuss disk and systems memory management, video memory management, swapping, and fragmentation.

3.1 Swapping

To handle large data sets also during pre-processing an out-of-core approach had to be taken. This means that data is only read into the system memory temporarily. Two swapping mechanisms have been tested and compared for pre-processing (memory mapped files & explicit swapping of geometry chunks). For 64 bit architectures, theoretically one could have relied entirely on the operating system to handle the swapping. However this was considered as too inefficient)

Instead of memory mapped files, we found an explicit swapping is more useful since it reflects the hierarchical subdivision of data, and also controls read and write actions.

- For every node of the tree that is being generated (i.e. in every recursion step) the complete subtree data needs to be read, handled and written back to disk (which amounts to the complete data set once per level of recursion)
- If the necessary data is smaller than the available systems memory, the complete leaf data can be handled in core.
- Otherwise, the subdivision has to be carried out in stages, where only a subset what fits into memory can be handled simultaneously.

Advantage over the memory-mapped file approach:

- Subdivision of geometry data into chunks that can be handled completely in core.
- Acceleration of pre-processing by up to a factor of 10.
- Swapping of data can be done highly efficiently due to linear access of hard drive (average transfer rates of 50MB/s (vs. vs. only 1 – 5 MB/s with memory mapped files and linear view on data)
- Enables asynchronous writing of data during generation. This further accelerates the process.

3.2 Fragmentation

Fragmentation of memory may lead to denial of requests for free memory of certain size despite the availability of enough free memory, because the gaps are not consecutive. We distinguish between disk fragmentation, virtual memory fragmentation and cache memory fragmentation.

In our implementation all geometry data of individual objects and fragments generated during subdivision are managed by several caches of fixed size in system memory. This

way geometry data no longer fragments the memory (only the management overhead data itself does) Geometry data may fragment the caches themselves, though. However, this can be mitigated by explicit cache de-fragmentation.

On 64-bit architectures the virtual memory is so large, that fragmentation does not matter, and caching can be omitted. Geometry data is not compacted on the hard disk. Therefore fragmentation does not occur, as long as no data is deleted.

The top-down generation of multi-dimensional hierarchies (kd-trees) promotes coherent data organization on the hard drive (in contrast to rendering, where data is needed in more or less random order) Separate storage of geometry data left and right of the separating plane prevents disk fragmentation.

Data are read, handled and deleted in the order in which they were also written. This leads to large gaps that can be utilized well for the final files. As soon as the data of a node completely fits into memory, no more swapping operations are necessary. Therefore no gaps in the swap space smaller than the size of the available system memory can be generated. By preventing fragmentation the available memory can be used more efficiently. This enlarges the useful memory and accelerates the preprocessing.

3.3 System Memory Management

During preprocessing geometry is subdivided into chunks of approximately equal size. For rendering, a chunk is either completely loaded into system memory or not at all (atomicity). Since chunks all have about equal size, there will be hardly any fragmentation, and no special system memory caching structure was employed for rendering.

3.4 VRAM Management

In addition to systems memory, also the graphics memory (VRAM) needs to be managed. VRAM is usually much smaller than main memory. Therefore more often geometry needs to be replaced, which could lead to more fragmentation.

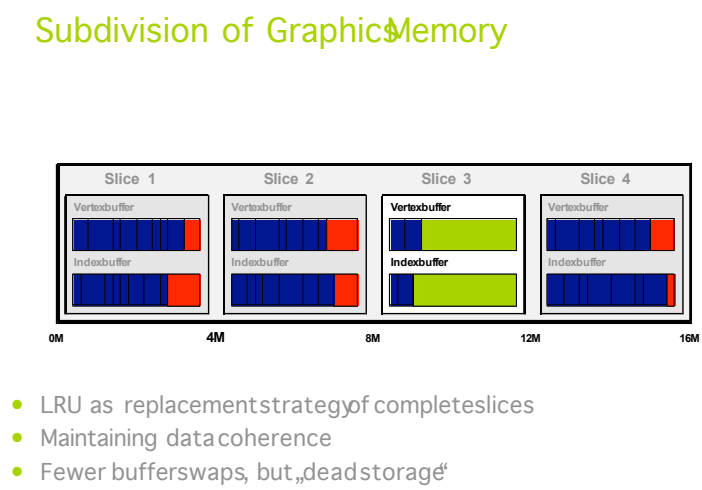


Figure 7: Illustrating memory management of VRAM

Management of data in VRAM is done with OpenGL Vertex- and Index buffers. If a leaf node has its own buffer, it needs to be swapped after each frame, which is an expensive operation. (approximately 1000 – 5000 times per frame).

Subdivision of VRAM in a few large slices is theoretically better, however it causes an internal fragmentation by continuous replacement. Fragmentation is avoided by always filling a slice from front to back, in render priority (only visible nodes are in VRAM).

If there are no more free slices, a decision is made using the LRU (least recently used) strategy, as to which slice can be completely emptied and refilled. Several nodes are replaced at once in VRAM, and new nodes are entered again in the order in which they are rendered. This maintains the coherence of data and finally, this accelerates rendering. However, data being loaded from main memory into VRAM causes a slow down during the first frames after loading, but is hardly noticeable later on.

4 Performance Analysis of Interviews3D

To evaluate the performance of the visibility-guided rendering approach and output sensitive clash detection, test data sets are required which present a challenge even to modern

graphics cards and CPUs. The freely available “powerplant” model consists of 12.748.510 triangles. In test case we multiplied this model up to 100-fold with individual transformations. The complete scene contains approximately 1.3 billion polygons, which occupy roughly 35 GB of memory on the hard disk (see fig. 8). A real world airplane model, the Boeing 777 contains 350 million polygons and over 700.000 objects. Navigation through these extremely large models can still be carried out in real time (over 20 frames per second - see fig. 9).

The performance measurements have been taken on a PC with an AMD Athlon X2 3800+ (DualCore 2Ghz) processor 2GB RAM, with a high-end consumer graphics card: Geforce 7900 GTX (512 Mb VRAM), a 7200 rpm hard drive with 8 Mb Cache. The geometry cache was set to 1024 MB.

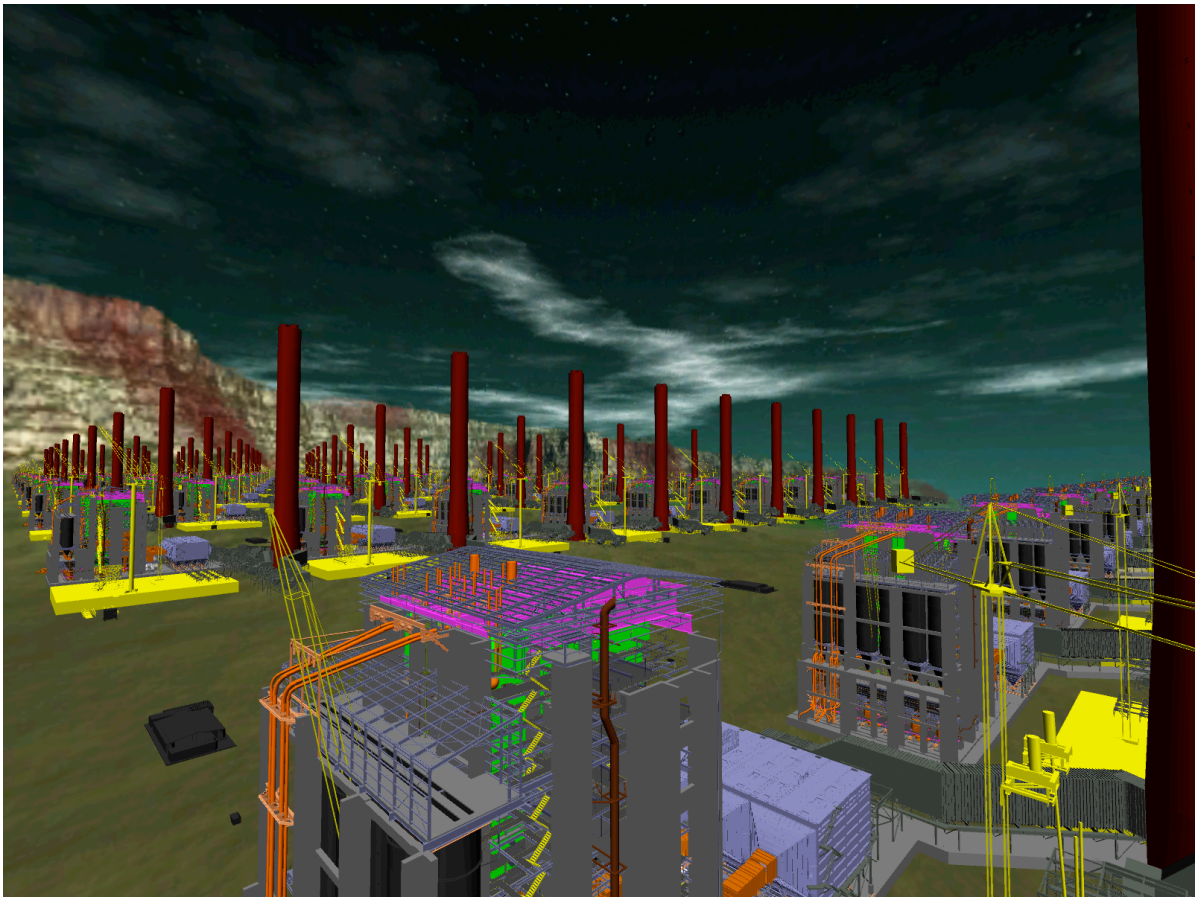
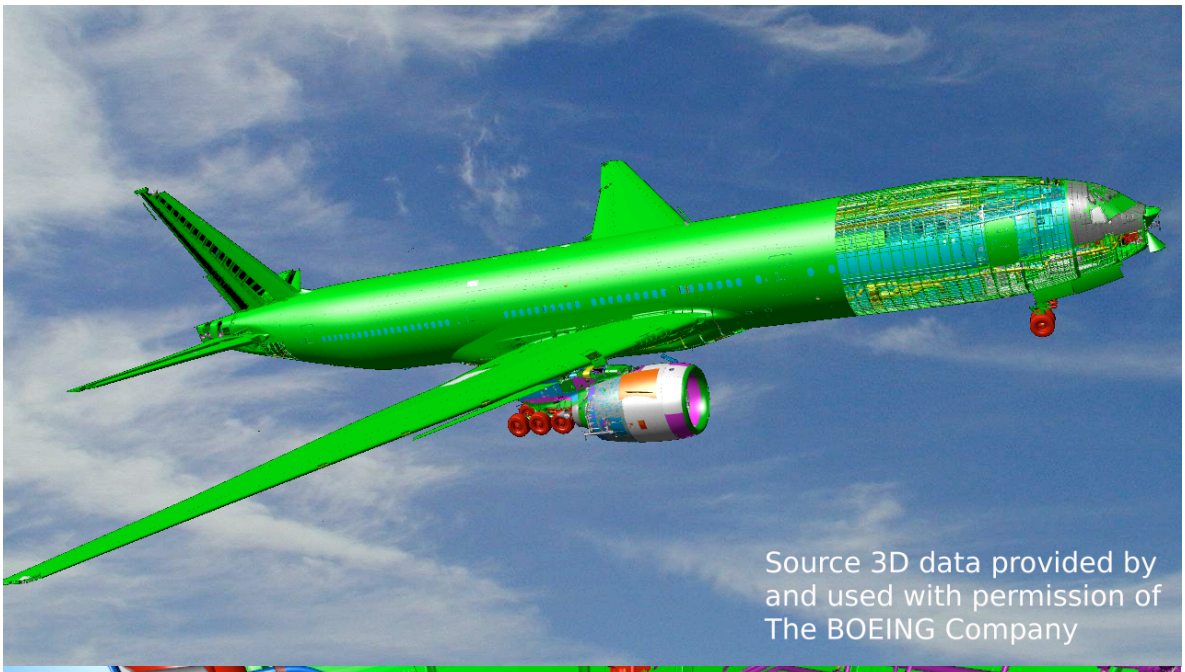
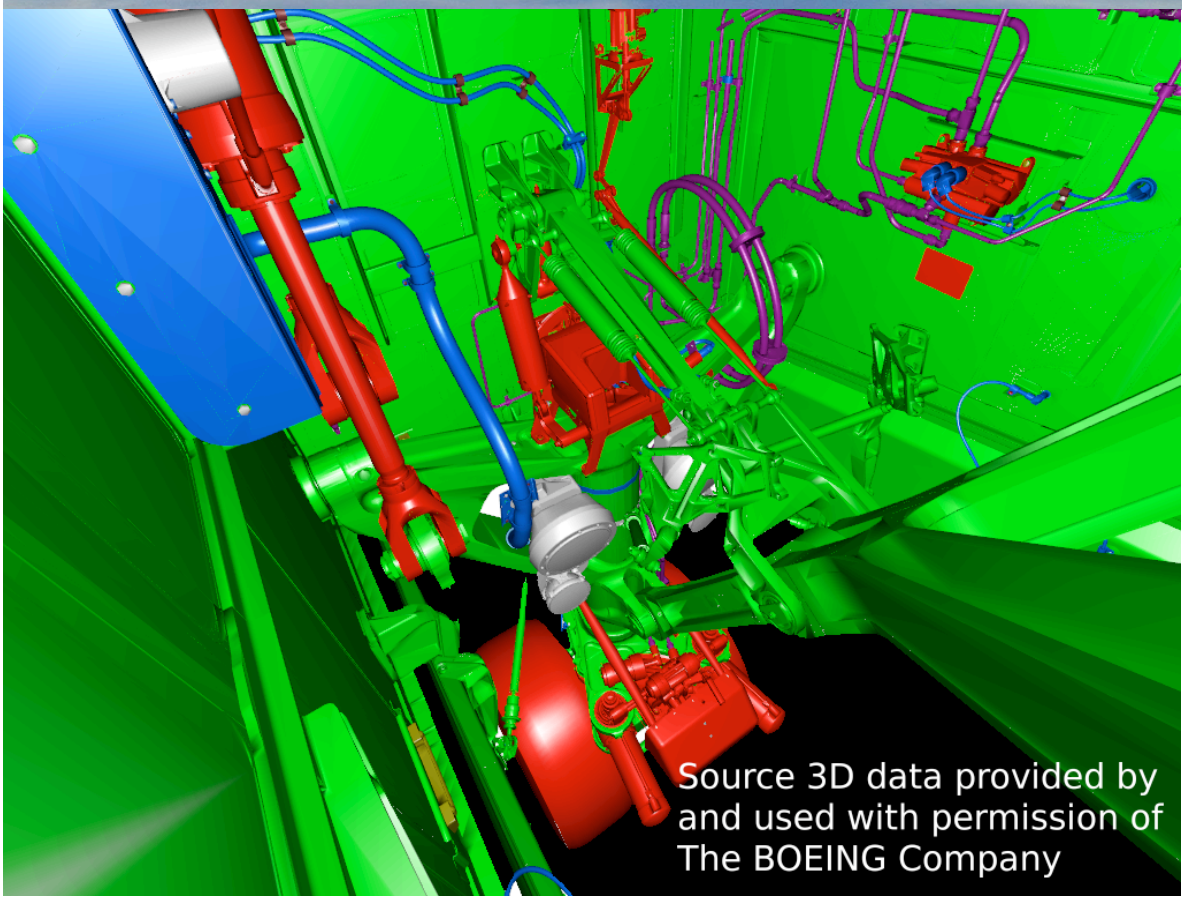


Figure 8: Real-time rendering of an extremely large scene, consisting of approximately 1.300.000.000 polygons.


Source 3D data provided by
and used with permission of
The BOEING Company


Source 3D data provided by
and used with permission of
The BOEING Company

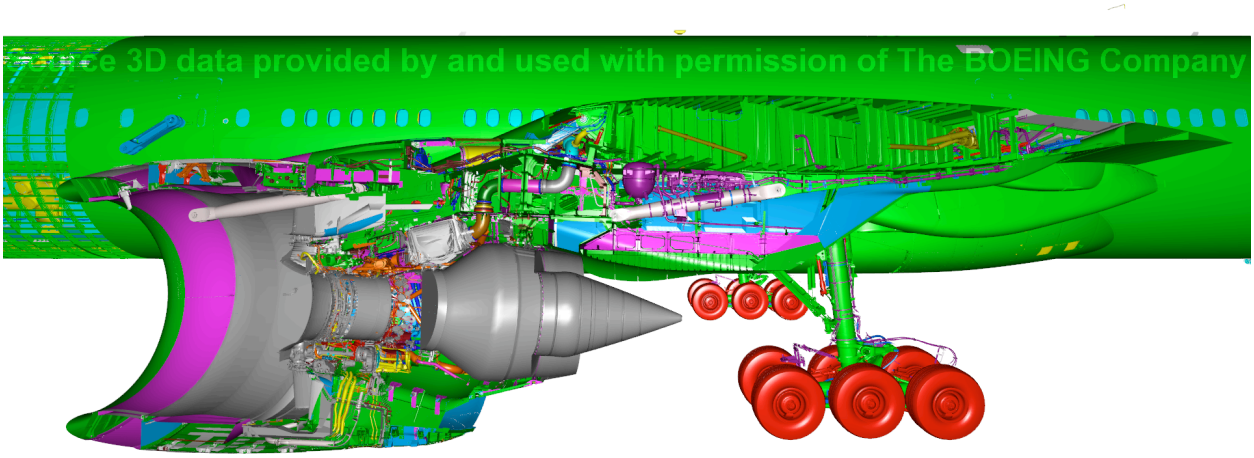
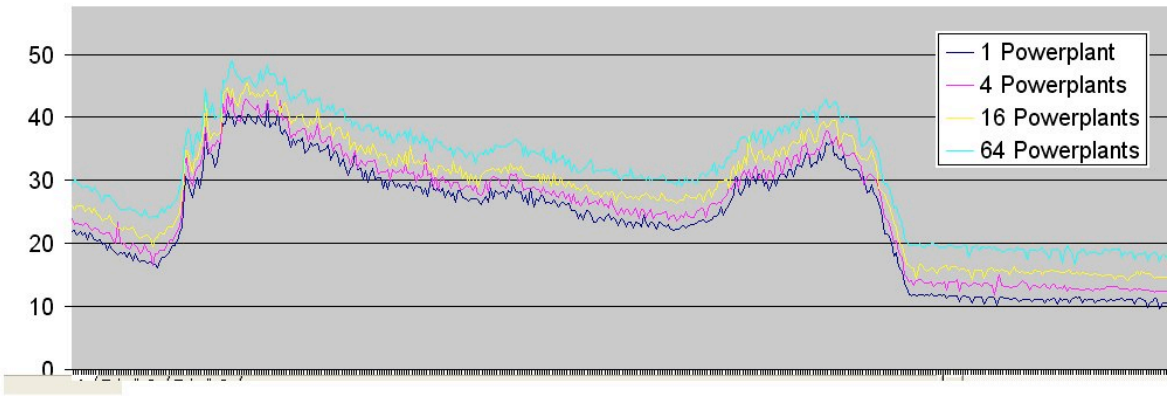


Figure 9: Screen shots from a camera path through the Boeing 777 model. A detailed model consisting of 350.000.000 polygons rendered at 20 fps and higher.

4.1 Render times as a function of data volume



This diagram shows the render times (ms) a camera path and measured for 13 million polygons (1 powerplant) to 800 million polygons (64 powerplants)

The frame rate varies only slightly with the data size (15% slower for a 64 times larger model with 800 million triangles or even for 1.3 billion triangles) and depends more on the view angle and how much is actually visible from the camera position (output sensitivity!!). The frame rate basically never falls below 20 fps.

4.2 Render times as a function of screen size

Output sensitivity is also reflected by the fact that slightly more polygons are rendered for a large screen since more details become visible (the data size does not vary). The performance is the average of the camera path used in 4.1

Screen Resolution	800x600	1024x768	1152x864	1280x1024
Nr. of Pixels	480,000	786,432	995,328	1,310,720
Avg. render time per frame (ms)	28.51	35.36	42.25	42.60
Avg. frames per second.	35.07	28.27	23.66	23.47

4.3 Complete build of large models as function of data size

The time for creating a complete spatial database of spatially separated objects grows roughly linearly with the data size. (from 20 seconds for 13 million triangles to over 1 hr

for 1.3 billion polygons.) There is a penalty of about a factor of two for models over one Gigabyte that don't fit into core memory.

model	#objects	#tris	db size (mb)	total time (s)
2 Powerplants	2,488	25,497,020	643	39
4 Powerplants	4,976	50,993,996	1,219	139
8 Powerplants	9,952	101,987,708	2,437	281
16 Powerplants	19,904	203,974,120	4,884	589
32 Powerplants	39,808	407,943,384	9,718	1,194
64 Powerplants	79,616	815,874,048	19,354	2,639
100 Powerplants	12,440	1,274,851,000	30,240	4,127

4.4 Incremental build of the Boeing 777 Model

In the following test the Boeing 777 model with 350 million polygons (756.000 objects) is built in four steps and in one piece. One can observe that each step takes a bit longer than the previous one, because the new data needs to be inserted within previously stored objects. Four incremental steps take about 10% longer than building the same data set at once.

model	#objects	#tris	db size (mb)	time (s)
Boeing 777 part 1/4	180,508	85,280,861	2,734	356
Boeing 777 part 2/4	319,192	172,841,175	5,612	472
Boeing 777 part 3/4	544,754	260,654,753	8,745	601
Boeing 777 part 4/4	756,263	348,431,158	11,980	732
Boeing 777 in 4 parts	756,263	348,431,158	11,980	2,161
Boeing 777 complete	756,263	348,431,158	10,400	1,920

Incremental build vs. complete build

4.5 Selective delete of partial models

The deletion of partial models is the counterpart to incremental build. Here most of the time is used up by the hard drive. The CPU hardly contributes to the timing.

model	total time (s)	cpu time (s)
delete 1 Powerplant out of 16	2	1
delete 2 Powerplants out of 16	3	1
delete 4 Powerplants out of 16	5	1
delete 8 Powerplants out of 16	6	1
delete 6.25% of a Boeing 777	170	26
delete 12.5% of a Boeing 777	188	31

delete 25% of a Boeing 777	270	42
delete 50% of a Boeing 777	448	64

4.6 Selective updates of parts in a model

Finally, selective update operations replace parts within a model by new versions. Here selective update is combined by deletion and insertion. In-place updates (without intermediate deletion) can be significantly faster since the same sections of the database often can be reused, without restructuring the spatial index.

Task	Time used
Replace 2,5% of the Boeing 777	2.4 minutes
Delete 2.5% of the Boeing 777	65.7 seconds
Insert 2.5% of the Boeing 777	80 seconds

Selective updates of the Boeing 777 model: Approximately 8.75 million out of 350 million polygons can be replaced in 2.4 minutes

5 Conclusion

Interviews3D is a DMU platform that already today is capable of handling in real time over 100 to 1000 times larger data volumes than before on standard PC hardware. By exploiting several new approaches, such as incremental (selective) update and output-sensitive algorithms and data structures, as well as out-of core handling of data, important functions such as real time rendering, picking and manipulating objects, but also clash detection or ray tracing can be done for almost unlimited data volumes. The compute times grow only weakly with data size, mostly logarithmically. Therefore, the systems architecture can be regarded as future-proof even for doubly exponential growth of data volume over time. The platform can take advantage of multi processing GPUs and CPUs. Currently we are limited only by disk space of a computer or network server. In the future we will also support data network.

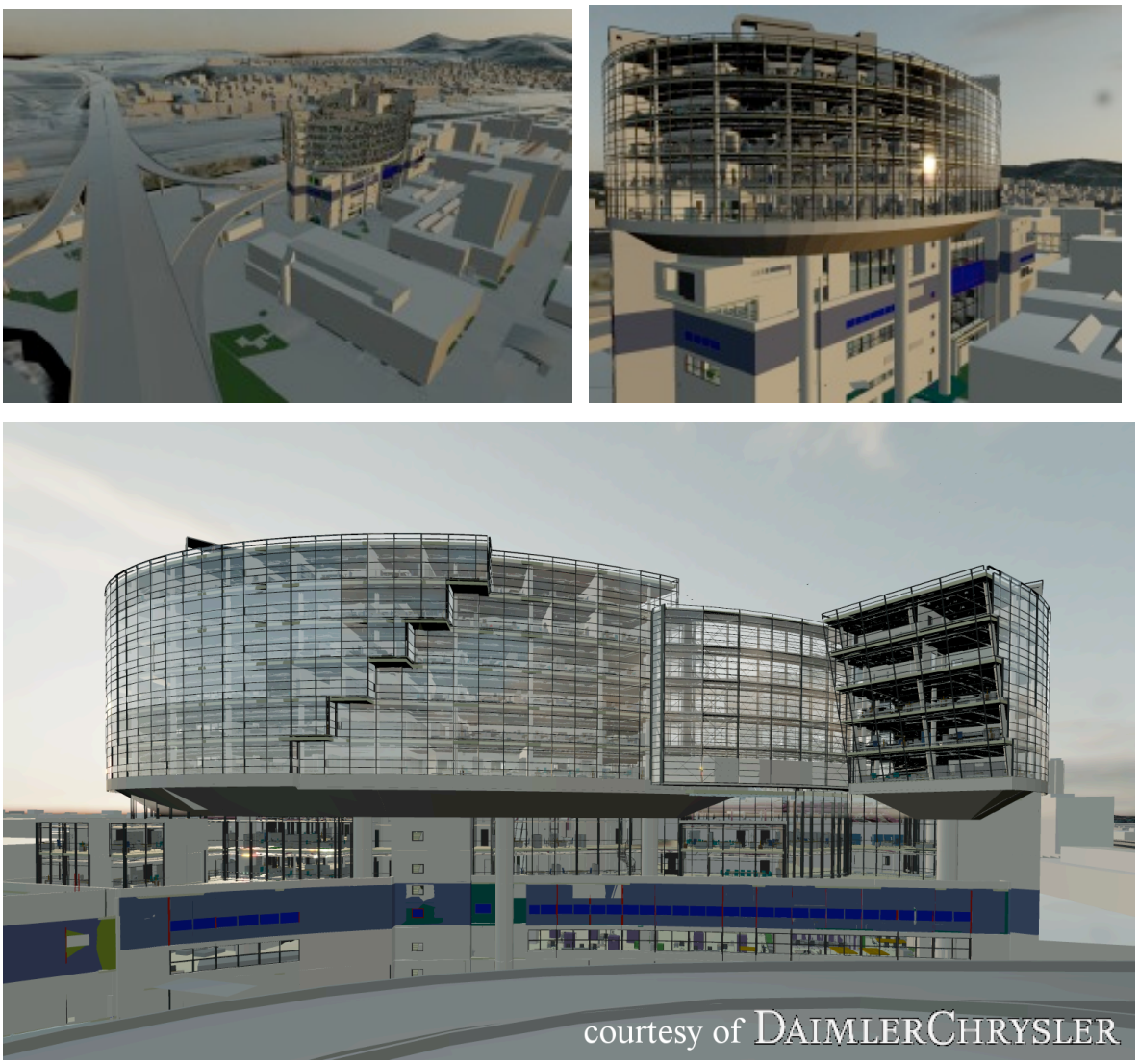


Figure 10: A digital Model of DaimlerChrysler's newly built Van Technology Center seen in its environment in Stuttgart-Untertürkheim. Courtesy of Daimler-Chrysler Digital Factory Planning.

Figure 11: A model of a VW golf GTI using programmable hardware shaders for realistic material effects.

6 References

[AM00] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. Journal of graphics tools,, pages 9–22, 2000.

[BWPP04] J Bittner, M. Wimmer, H. Piringer, and W. Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. In proceedings of Eurographics Conference, 2004.

[GGMR04] J Gausemeier, M. Grafe, C. Matysczok, R. Radowki. PC-Cluster für die Visualisierung von hoch-polygonalen 3D-Modellen, in Proceeding 3. Paderborner Workshop, Paderborn Augmented & Virtual Reality in der Produktentstehung, 17. ,18. Juni 2004. Hanser Verlag 2004.

[HeBr04] M. Heyer, B. Brüderlin, Hardwareunterstütztes Occlusion Culling zur Visualisierung sehr großer Modelle, in proceeding 3. Paderborner Workshop, Paderborn Augmented & Virtual Reality in der Produktentstehung, 17. ,18. Juni 2004. Hanser Verlag 2004.

[HeBr04b] M. Heyer, B. Brüderlin, Visibility-guided Rendering for Visualizing Very Large Virtual Reality Scenes, in proceeding 1st GI-Workshop on VR/AR TU-Chemnitz, September 2004.

[nVi03] nVidia. Using Vertex Buffer Objects. WWW, 2003. http://developer.nvidia.com/object/using VBOs.html.

[Sta03] D. Staneker. An Occlusion Culling Toolkit for OpenSG PLUS. Technical report, WSI/GRIS University of Tübingen, Germany, 2003.

[WDS01] I. Wald., A. Dietrich, and P. Slusallek. An interactive out-of-core rendering framework for visualizing massively complex models. In Proc. Eurographics Symposium on Rendering, 2004.

[WFPHS01] M. Wand, M. Fischer, I Peter, F.M. auf der Heide and W. Strasser. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In SIGGRAPH 2001 Proceedings, 2001

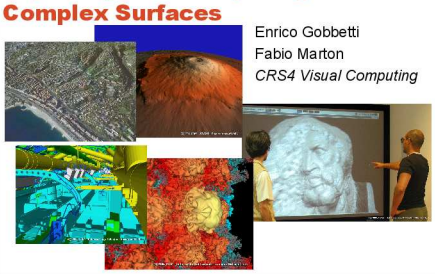
[Spi03] John Spitzer. OpenGL Performance Tuning. Game Developers Conference, 2003.

[RuLe00] Szymon Rusinkiewicz, Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes, SIGGRAPH 2000

8 GPU-friendly Accelerated Mesh-based and Mesh-less Techniques for the Output-sensitive Rendering of Huge Complex 3D Models

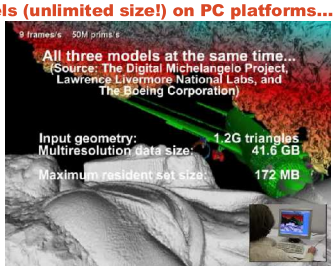
Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces

Enrico Gobbetti
Fabio Marton
CRS4 Visual Computing



Goal and Motivation

Accurate interactive inspection of very large models (unlimited size!) on PC platforms...

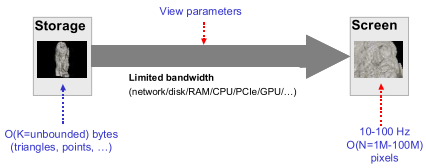


Xeon 2.4GHz / 1GB RAM / 70GB SCSI 320 Disk / NVIDIA 6800GTS

Size matters! Or does it? (1/10)

A real-time data filtering problem!

- Models of unbounded complexity on limited computers
  - We assume **less data on screen (N) than in model (K →∞)**
  - Need for **output-sensitive** techniques (O(N), not O(K))



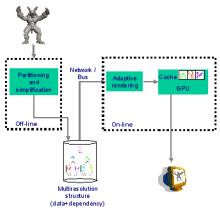
Ouput sensitive technique

- All techniques <u>must</u> combine
  - A multiresolution and spatial subdivision structure
    - Visual/geometric approximations
    - Spatial indexing
  - A view-dependent renderer
    - LOD culling
    - Visibility culling
  - An efficient memory management subsystem

Our contributions

GPU-friendly output-sensitive techniques

- Underlying ideas
  - **Chunk-based multiresolution structures**
    - Combine space partitioning + level of detail
    - Same structure used for visibility and detail culling
  - **Seamless combination of chunks**
    - Dependencies ensure consistency at the level of chunks
  - **Complex rendering primitives**
    - GPU programming features
    - Curvilinear patches, view-dependent voxels, ...
  - **Chunk-based external memory management**
    - Compression/decompression, block transfers, caching



Our contributions

GPU-friendly output-sensitive techniques

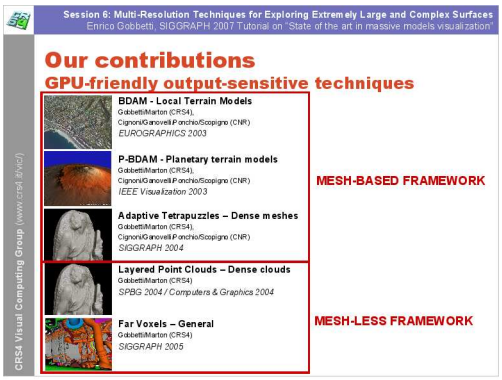
BDAM - Local Terrain Models
Gobbetti/Marton (CRS4),
Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
EUROGRAPHICS 2003

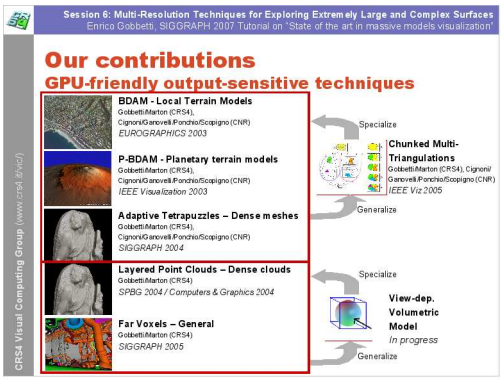
P-BDAM - Planetary terrain models
Gobbetti/Marton (CRS4),
Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
IEEE Visualization 2003

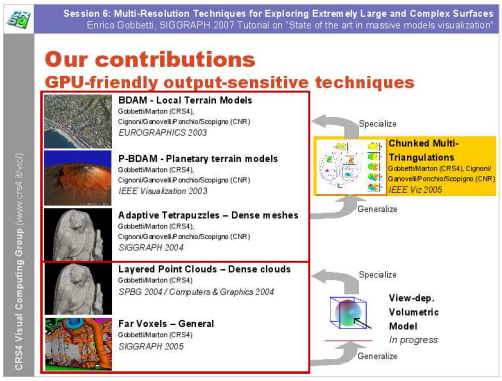
Adaptive Tetrapuzzles – Dense meshes
Gobbetti/Marton (CRS4),
Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
SIGGRAPH 2004

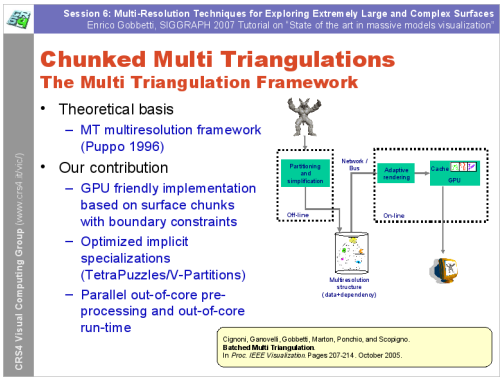
Layered Point Clouds – Dense clouds
Gobbetti/Marton (CRS4)
SPBG 2004 / Computers & Graphics 2004

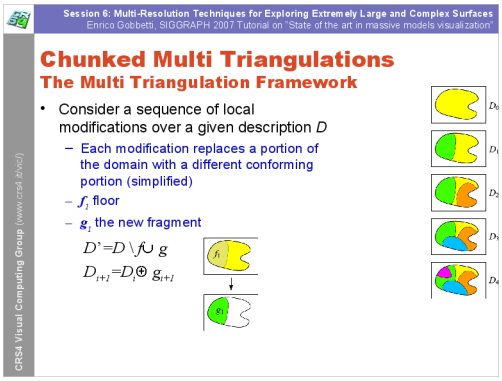
Far Voxels – General
Gobbetti/Marton (CRS4)
SIGGRAPH 2005

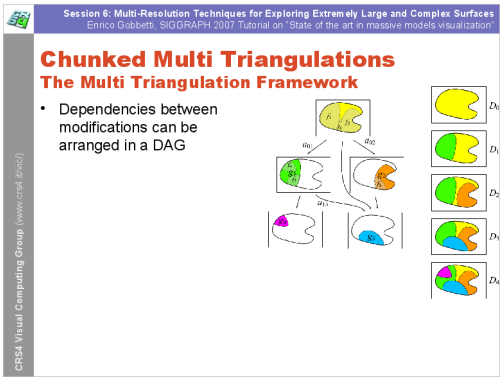

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Our contributions
GPU-friendly output-sensitive techniques
BDAM - Local Terrain Models
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
EUROGRAPHICS 2003
P-BDAM - Planetary terrain models
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
IEEE Visualization 2003
Adaptive Tetrapuzzles – Dense meshes
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
SIGGRAPH 2004
Layered Point Clouds – Dense clouds
Gobbetti/Marton (CRS4)
SPBG 2004 / Computers & Graphics 2004
Far Voxels – General
Gobbetti/Marton (CRS4)
SIGGRAPH 2005
MESH-BASED FRAMEWORK
MESH-LESS FRAMEWORK


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Our contributions
GPU-friendly output-sensitive techniques
BDAM - Local Terrain Models
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
EUROGRAPHICS 2003
P-BDAM - Planetary terrain models
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
IEEE Visualization 2003
Adaptive Tetrapuzzles – Dense meshes
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
SIGGRAPH 2004
Layered Point Clouds – Dense clouds
Gobbetti/Marton (CRS4)
SPBG 2004 / Computers & Graphics 2004
Far Voxels – General
Gobbetti/Marton (CRS4)
SIGGRAPH 2005
Specialize
Chunked Multi-Triangulations
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
IEEE Viz 2005
Generalize
Specialize
View-dep. Volumetric Model
In progress
Generalize


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Our contributions
GPU-friendly output-sensitive techniques
BDAM - Local Terrain Models
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
EUROGRAPHICS 2003
P-BDAM - Planetary terrain models
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
IEEE Visualization 2003
Adaptive Tetrapuzzles – Dense meshes
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
SIGGRAPH 2004
Layered Point Clouds – Dense clouds
Gobbetti/Marton (CRS4)
SPBG 2004 / Computers & Graphics 2004
Far Voxels – General
Gobbetti/Marton (CRS4)
SIGGRAPH 2005
Specialize
Chunked Multi-Triangulations
Gobbetti/Marton (CRS4), Cignoni/Ganovelli/Ponchio/Scopigno (CNR)
IEEE Viz 2005
Generalize
Specialize
View-dep. Volumetric Model
In progress
Generalize

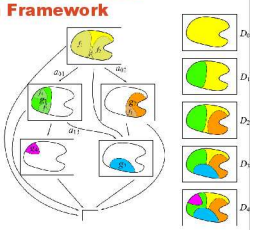

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Chunked Multi Triangulations
The Multi Triangulation Framework
Theoretical basis
MT multiresolution framework (Puppo 1996)
Our contribution
GPU friendly implementation based on surface chunks with boundary constraints
Optimized implicit specializations (TetraPuzzles/V-Partitions)
Parallel out-of-core pre-processing and out-of-core run-time
Partitioning and simplification
Network / Bus
Adaptive rendering
Cache
GPU
Off-line
On-line
Multiresolution structure (data+dependency)
Cignoni, Ganovelli, Gobbetti, Marton, Ponchio, and Scopigno.
Batched Multi Triangulation.
In Proc. IEEE Visualization. Pages 207-214. October 2005.


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Chunked Multi Triangulations
The Multi Triangulation Framework
Consider a sequence of local modifications over a given description D
Each modification replaces a portion of the domain with a different conforming portion (simplified)
f_i floor
g_i the new fragment
$D' = D \setminus f \cup g$
$D_{i+1} = D_i \oplus g_{i+1}$


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
CRS4 Visual Computing Group (www.crs4.it/vic/)
Chunked Multi Triangulations
The Multi Triangulation Framework
Dependencies between modifications can be arranged in a DAG

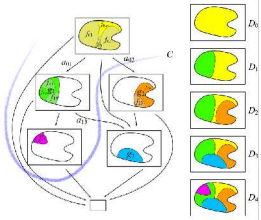
Chunked Multi Triangulations
The Multi Triangulation Framework

- Dependencies between modifications can be arranged in a DAG
  - Adding a sink to the DAG we can associate each fragment to an arc leaving a node



Chunked Multi Triangulations
MT Cuts

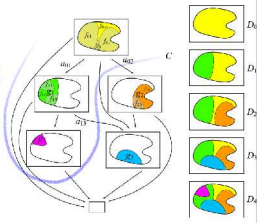
- A cut of the DAG defines a new representation
  - Just paste all the fragments above the cut



$D^* = D_0 \oplus g_1 \oplus g_4$

Chunked Multi Triangulations
MT Cuts

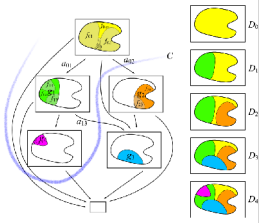
- A cut of the DAG defines a new representation
  - Collect all the fragment floors of cut arcs and you get a new conforming mesh



$D^* = D_0 \oplus g_1 \oplus g_4 = f_{0\infty} \cup f_{02} \cup f_{03} \cup f_{13} \cup f_{1\infty} \cup f_{4\infty}$

Chunked Multi Triangulations
GPU Friendly MT

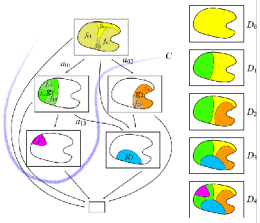
- Chunked MT assume fragments are triangle patches with proper boundary constraints
  - DAG << original mesh (patches composed by thousands of tri)
  - Structure memory + traversal overhead amortized over thousands of triangles
  - Per-patch optimizations



$D^* = D_0 \oplus g_1 \oplus g_4 = f_{0\infty} \cup f_{02} \cup f_{03} \cup f_{13} \cup f_{1\infty} \cup f_{4\infty}$

Chunked Multi Triangulations
GPU Friendly MT

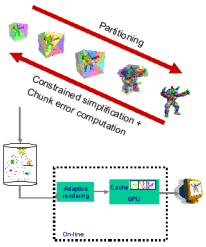
- Chunked MT assume regions provide good hierarchical space-partitioning
  - Compact
    - Close-to-spherical
  - Used for computing fast projected error upper bounds
  - Used for visibility queries



$D^* = D_0 \oplus g_1 \oplus g_4 = f_{0\infty} \cup f_{02} \cup f_{03} \cup f_{13} \cup f_{1\infty} \cup f_{4\infty}$

Chunked Multi Triangulations
GPU Friendly MT

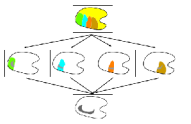
- Construction
  - Start with hires triangle soup
  - Partition model using a **hierarchical space partitioning** scheme
  - Construct non-leaf cells by bottom-up **recombination and simplification** of lower level cells
  - Assign **model space errors** to cells
- Rendering
  - Refine conformal hierarchy, render selected precomputed cells
  - Project errors to screen
  - Dual queue

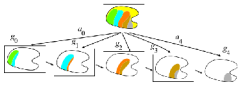


CRS4 Visual Computing Group (www.crs4.it/vc/)

Chunked Multi Triangulations
DAG problems

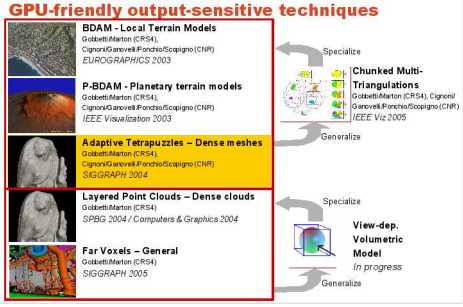
- Not all MTs are good MTs!
  - The **topology of dependencies** may lower the adaptivity of the multiresolution structure
    - Cascading dependencies are BAD!!!
  - The **geometry of DAG regions** may cause problems in view-dependent rendering
    - Compact (close-to-spherical) regions for good constant error bounds
    - Long+thin regions are BAD!
- Proposed solutions:
  - SIGGRAPH 2004: Efficient constrained technique (TetraPuzzles)
  - IEEE Viz 2005: General construction technique (V-Partition)





CRS4 Visual Computing Group (www.crs4.it/vc/)

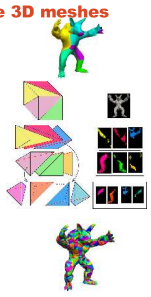
Our contributions
GPU-friendly output-sensitive techniques



CRS4 Visual Computing Group (www.crs4.it/vc/)

Adaptive TetraPuzzles
Multiresolution Model for Dense 3D meshes

- **Adaptive TetraPuzzles**: High performance visualization of dense 3D meshes
  - Two-level multiresolution model based on volumetric decomposition
  - Implicit MT based on tetrahedra hierarchy



Cignoni, Ganovelli, Gobbetti, Marton, Ponchio, and Scopigno.
Adaptive TetraPuzzles - Efficient Out-of-core Construction and Visualization of Gigantic Polygonal Models.
ACM Transactions on Graphics, 23(3), August 2004
(Proc. SIGGRAPH 2004).

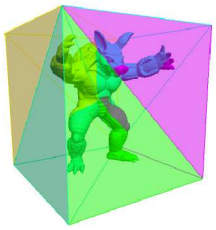
CRS4 Visual Computing Group (www.crs4.it/vc/)

Adaptive TetraPuzzles
Overview

- **Construction**
  - Start with hires triangle soup

CRS4 Visual Computing Group (www.crs4.it/vc/)

Adaptive TetraPuzzles
Overview

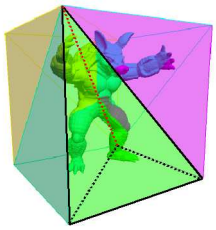


Target = *k* triangles/chunk

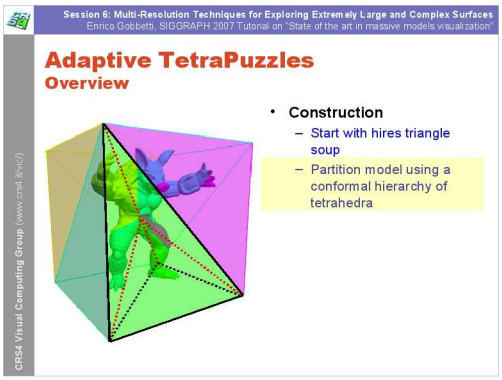
- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra

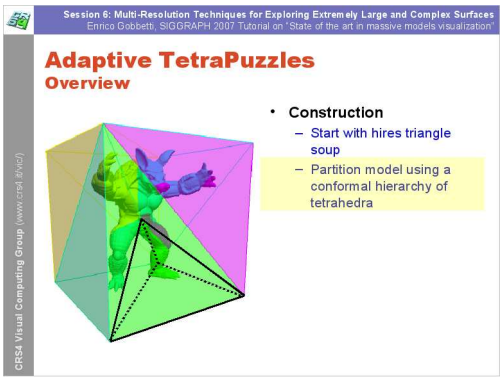
CRS4 Visual Computing Group (www.crs4.it/vc/)

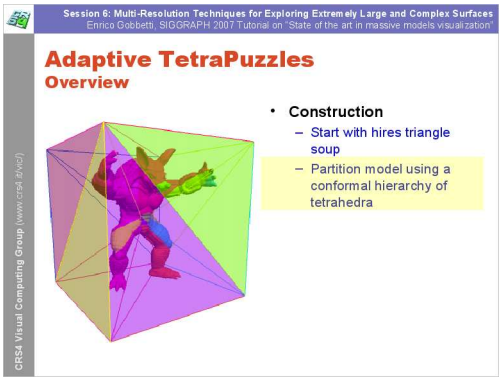
Adaptive TetraPuzzles
Overview

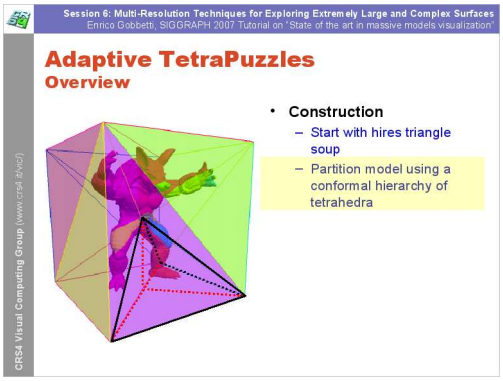


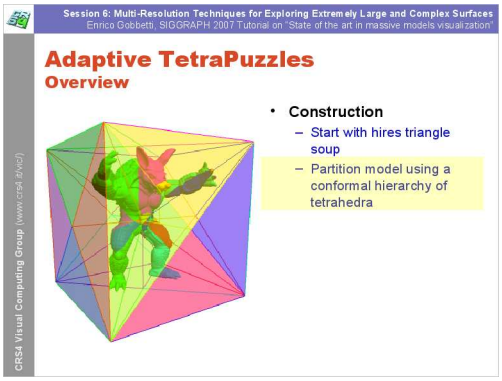
- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra

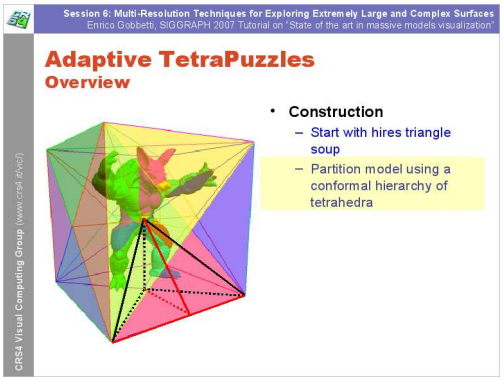

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
• Construction
– Start with hires triangle soup
– Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vic/)

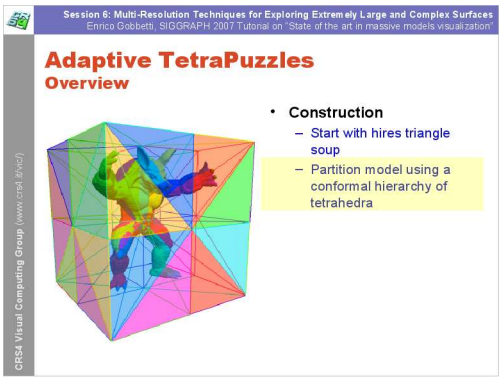

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
• Construction
– Start with hires triangle soup
– Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vic/)

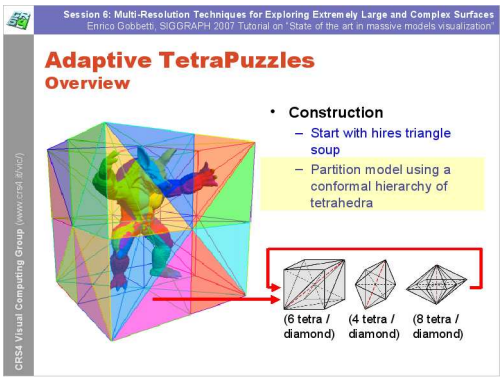

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
• Construction
– Start with hires triangle soup
– Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vic/)

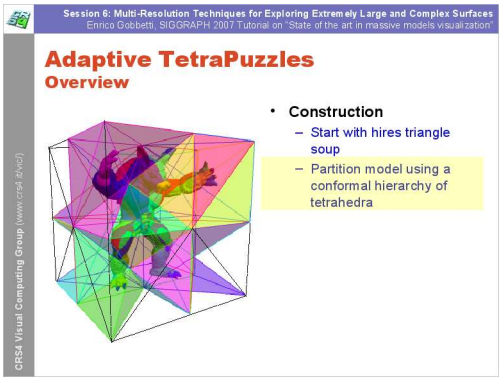

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
• Construction
– Start with hires triangle soup
– Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vic/)

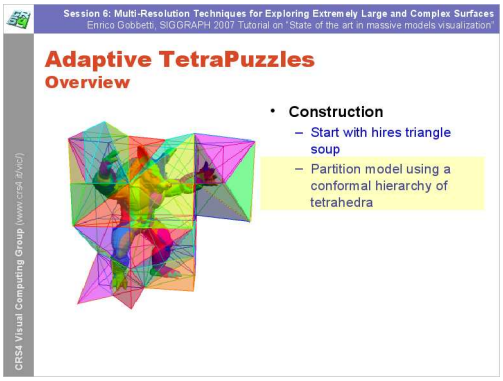

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
• Construction
– Start with hires triangle soup
– Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vic/)

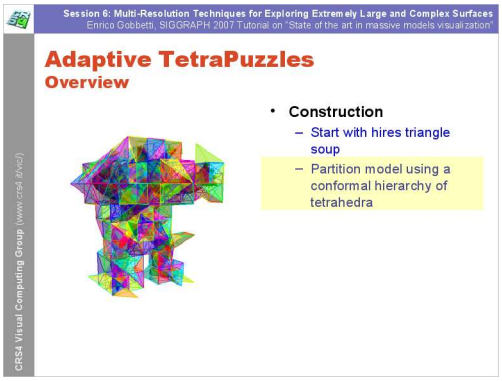

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
• Construction
– Start with hires triangle soup
– Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vic/)

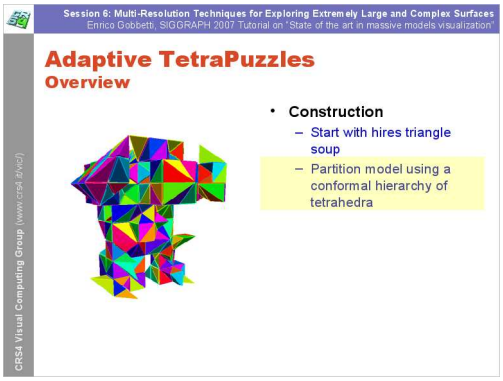

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
Construction
Start with hires triangle soup
Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vc/)


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
Construction
Start with hires triangle soup
Partition model using a conformal hierarchy of tetrahedra
(6 tetra / diamond)
(4 tetra / diamond)
(8 tetra / diamond)
CRS4 Visual Computing Group (www.crs4.it/vc/)


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
Construction
Start with hires triangle soup
Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vc/)


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
Construction
Start with hires triangle soup
Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vc/)


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
Construction
Start with hires triangle soup
Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vc/)


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"
Adaptive TetraPuzzles
Overview
Construction
Start with hires triangle soup
Partition model using a conformal hierarchy of tetrahedra
CRS4 Visual Computing Group (www.crs4.it/vc/)

Adaptive TetraPuzzles
Overview

k triangles/chunk

- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Adaptive TetraPuzzles
Overview

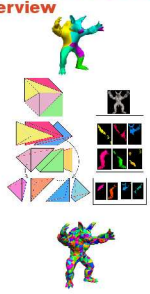
k triangles/chunk

- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Adaptive TetraPuzzles
Overview

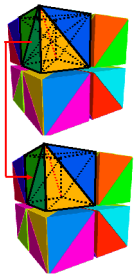


- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Adaptive TetraPuzzles
Overview

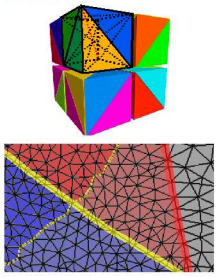


- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Adaptive TetraPuzzles
Overview



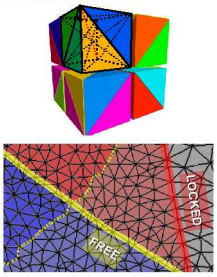
- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells

Diamond external boundary
Diamond internal boundary
Child tetrahedra boundary

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Adaptive TetraPuzzles
Overview

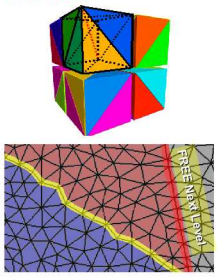


- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells

Diamond external boundary
Diamond internal boundary
Child tetrahedra boundary

CRS4 Visual Computing Group (www.crs4.it/vic/)

Adaptive TetraPuzzles
Overview



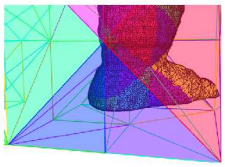
- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells

Diamond external boundary
Diamond internal boundary

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Adaptive TetraPuzzles
Overview

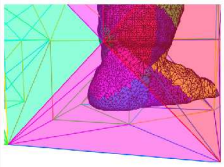


- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Adaptive TetraPuzzles
Overview



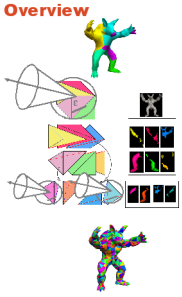
- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells

NO CRACKS / NO GLOBALLY LOCKED BOUNDARY!

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Adaptive TetraPuzzles
Overview

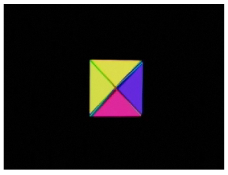


- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells
- **Rendering**
  - Refine conformal hierarchy, render selected precomputed cells

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Adaptive TetraPuzzles
Overview



View dependent mesh refinement

- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells
- **Rendering**
  - Refine conformal hierarchy, render selected precomputed cells

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Adaptive TetraPuzzles
Overview

Independent diamond processing

For each mesh chunk:
Simplify + stripify +
compress + eval bounds/error

Out-of-core + parallel

- **Construction**
  - Start with hires triangle soup
  - Partition model using a conformal hierarchy of tetrahedra
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells

Out-of-core cull+refine traversal / GPU cached optimized meshes

- **Rendering**
  - Refine conformal hierarchy, render selected precomputed cells

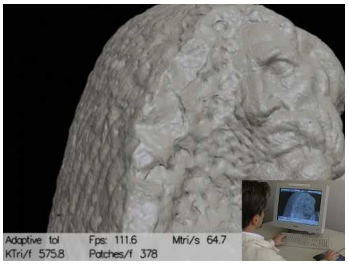
CRS4 Visual Computing Group (www.crs4.it/vic/)

CRS4 Visual Computing Group (www.crs4.it/vic/)

Adaptive TetraPuzzles
Results

Michelangelo's St. Matthew
Source: Digital Michelangelo Project
Data: 374M triangles

Intel Xeon 2.4GHz 1GB
GeForce FX 5800U AGP8X



Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

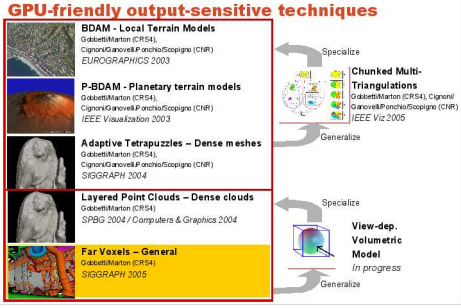
Adaptive TetraPuzzles
Conclusions

- Yet another multiresolution algorithm for rendering large static meshes
  - First GPU bound method for very large meshes
  - State of the art performance
    - GPU bound
    - >4Mtri/frame at >30 fps on modern GPUs
  - Tuned for large dense models with "well behaved" surface
  - Special case of general MT framework

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Our contributions
GPU-friendly output-sensitive techniques

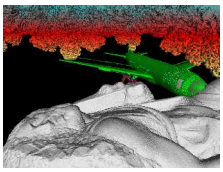


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Our contributions
Far Voxels – General 3D models

- Far Voxels: High performance visualization of arbitrary 3D models
  - Mixed model
  - Seamless integration of occlusion culling with out-of-core data management and multiresolution rendering



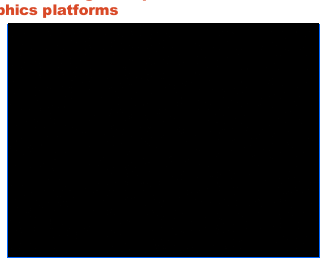
Gobbetti and Marton.
Far Voxels – A multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms.
ACM Transactions on Graphics, 23(4), August 2005
(Proc. SIGGRAPH 2005).

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Real-time inspection of huge complex models on commodity graphics platforms

- Huge
  - $O(10^9)$ triangles/bytes
- Complex
  - Heterogeneous materials
  - High topological genus
  - Highly variable depth complexity
  - Fine geometric details
  - "Bad" tessellations



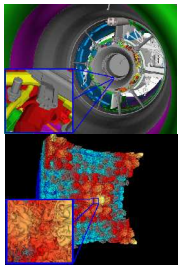
Xeon 2.4GHz / 1GB RAM / 70GB SCSI 320 Disk
NVIDIA 6800GT AGP 8X

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Handling Huge Complex 3D models

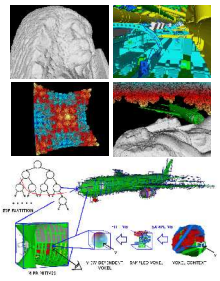
- Classic multiresolution models
  - Error measured on boundary surfaces
  - LOD construction based on local surface coarsening/simplification operations
  - Visibility culling decoupled from multiresolution
- Hard to apply to models with high detail <u>and</u> complex topology <u>and</u> high depth complexity!



CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Handling Huge Complex 3D models

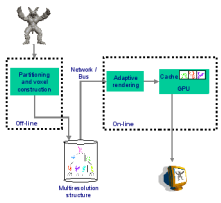
- General purpose technique that targets many model kinds
- Underlying ideas
  - Multi-scale modeling of appearance rather than geometry
  - Volume-based rather than surface-based
  - Tight integration of visibility and LOD construction
  - GPU accelerated (programmabilty + batching)



CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Overview

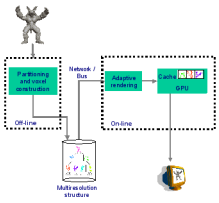
- Basic building block
  - Far voxel primitive
- Construction
  - BSP of the input model
  - Multiresolution structure
  - Far voxel
- Rendering
  - Selective refinement
  - Occlusion culling
  - Far voxel rendering
- Results
  - Preprocessing
  - Rendering



CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Overview

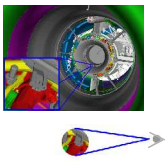
- Basic building block
  - Far voxel primitive
- Construction
  - BSP of the input model
  - Multiresolution structure
  - Far voxel
- Rendering
  - Selective refinement
  - Occlusion culling
  - Far voxel rendering
- Results
  - Preprocessing
  - Rendering



CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
The Far Voxel Concept

- Assumption: opaque surfaces, non participating medium
- Goal is to represent the appearance of complex far geometry
  - Near geometry can be represented at full resolution

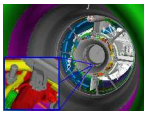


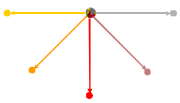
CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
The Far Voxel Concept

- Assumption: opaque surfaces, non participating medium
- Goal is to represent the appearance of complex far geometry
  - Near geometry can be represented at full resolution
- Idea is to discretize a model into many small volumes located in the neighborood of surfaces
  - Approximates how a small subvolume of the model reflects the incoming light

=> View-dependent voxel

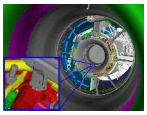




CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
The Far Voxel Concept

- A far voxel returns color attenuation given
  - View direction
  - Light direction



$$Shader_i(v,l) = BRDF_i(v,l)(n(v).l)_+$$

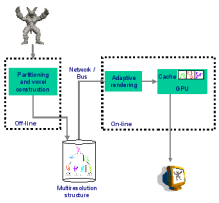
Shader = f (view direction, light direction)

- Rendered using a customized vertex shader executed on the GPU

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Overview

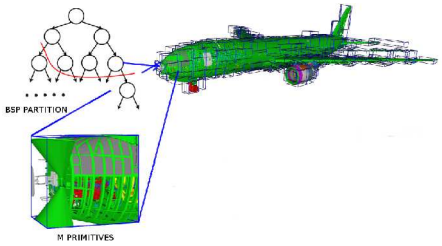
- Basic building block
  - Far voxel primitive
- Construction
  - BSP of the input model
  - Multiresolution structure
  - Far voxel
- Rendering
  - Selective refinement
  - Occlusion culling
  - Far voxel rendering
- Results
  - Preprocessing
  - Rendering



Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

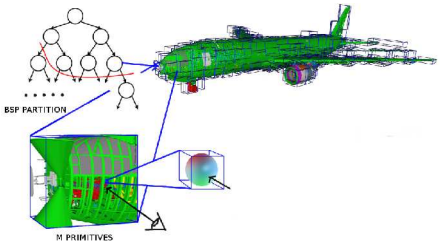
Far Voxels
Construction overview



Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

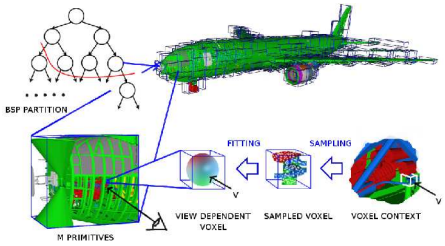
Far Voxels
Construction overview



Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Construction overview

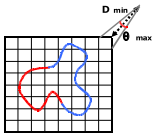


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Construction overview: Inner nodes

- Sample a model subvolume to build a grid of far voxels
- Voxels are far
  - Project to worst case θ_{max}
  - Viewed not closer than d_{min}



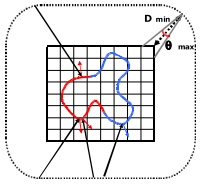
Section of the 3D grid of far voxels

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Construction overview: Inner nodes

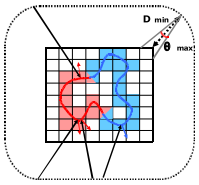
- Sample a model subvolume to build a grid of far voxels
- Voxels are far
  - Project to worst case θ_{max}
  - Viewed not closer than d_{min}
- Raycasting samples original model and identifies visible voxels



Section of the 3D grid of far voxels

Far Voxels
Construction overview: Inner nodes

- Sample a model subvolume to build a grid of far voxels
- Voxels are far
  - Project to worst case θ_{max}
  - Viewed not closer than d_{min}
- Raycasting samples original model and identifies visible voxels

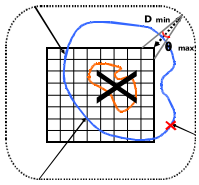


Section of the 3D grid of far voxels

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Far Voxels
Construction overview: Object Space Occlusion

- Environment occlusion
- Cull interior part of grid of far voxels



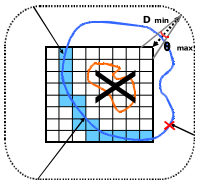
Section of the 3D grid of far voxels

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Far Voxels
Construction overview: Object Space Occlusion

- Environment occlusion
- Cull interior part of grid of far voxels



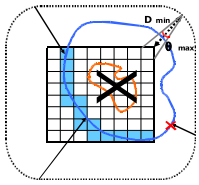
Section of the 3D grid of far voxels

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Far Voxels
Construction overview: Object Space Occlusion

- Environment occlusion
- Cull interior part of grid of far voxels
- Culls 40% of the high depth complexity Boeing 777 model,
  - worst case θ_{max} = 0.5 deg (~10 pixel tolerance for 1024x1024 viewport using 50deg FOV)
- Minimize artifacts due to leaking of occluded parts of different colors



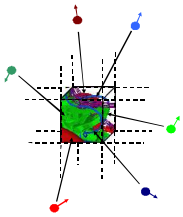
Section of the 3D grid of far voxels

CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Far Voxels
Construction overview: Far Voxel

- Consider voxel subvolume
- Samples gathered from unoccluded directions
  - Sample:
    - (BRDF, **n**) = f(view direction)

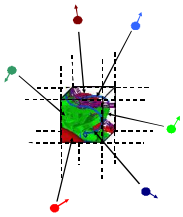


CRS4 Visual Computing Group (www.crs4.it/vic/)

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

Far Voxels
Construction overview: Far Voxel

- Consider voxel subvolume
- Samples gathered from unoccluded directions
  - Sample:
    - (BRDF, **n**) = f(view direction)
- Compress shading information by fitting samples to a compact analytical representation

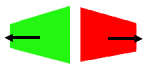


CRS4 Visual Computing Group (www.crs4.it/vic/)

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Construction overview: Far Voxel Shaders

- Phenomenological shader types
  - Flat shader
    - Normal
    - Front and back material
  - Smooth shader
    - 6 normals
    - 6 materials
    - associated with ±x, ±y, ±z



Flat shader: front and back material

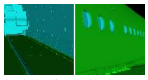
Smooth shader: stored info view dep. representation

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Construction overview: Far Voxel Shaders

- Phenomenological shader types
  - Flat shader
    - Normal
    - Front and back material
  - Smooth shader
    - 6 normals
    - 6 materials
    - associated with ±x, ±y, ±z



Flat shader: front and back material

Smooth shader: complex geometry

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Construction overview: Far Voxel Shaders

- Build all the K different far voxels representations
  - K = flat, smooth..
  - Principal component analysis
- Evaluate each representation error
  - Compare real values (samples) with the voxel approximations from the sample direction

Flat proxy: 2 components

Smooth proxy: 6 components

Others...

$$Err_{(k)} = \Sigma_i \Sigma_j \left(BRDF_i^{(sampled)}(\mathbf{v}_i, \mathbf{l}_j) \max(\mathbf{n}_i \cdot \mathbf{l}_j, 0) - Shader^{(k)}(\mathbf{v}_i, \mathbf{l}_j) \right)^2$$

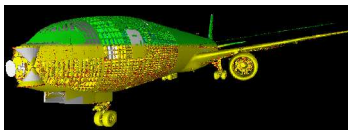
- Choose approximation with lowest error

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxel Distribution
on a perspective view of the Boeing 777

- Flat shaders
- Smooth shaders (complex local geometry)
- Triangles

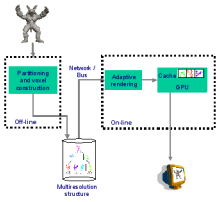


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Overview

- Basic building block
  - Far voxel primitive
- Construction
  - BSP of the input model
  - Multiresolution structure
  - Far voxel
- Rendering
  - Selective refinement
  - Occlusion culling
  - Far voxel rendering
- Results
  - Preprocessing
  - Rendering

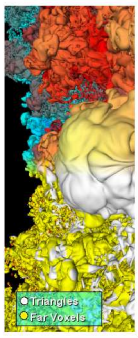


Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Rendering

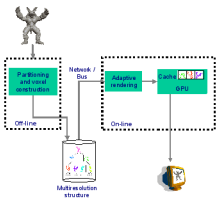
- Hierarchical traversal with coherent culling
  - Stop when out-of view, occluded (GPU feedback), or accurate enough
- Leaf node: Triangle rendering
  - Draw the precomputed triangle strip
- Inner node: Voxel rendering
  - For each far voxel type
    - Enable its shader
    - Draw all its view dependent primitives using glDrawArrays
  - Splat voxels as antialiased point primitives
  - Limits
    - Does not consider primitive opacity
    - Rendering quality similar to one-pass point splat methods (no sorting/blending)



CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Overview

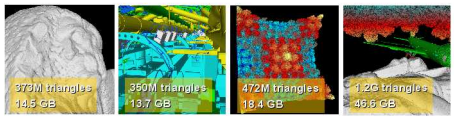
- Basic building block
  - Far voxel primitive
- Construction
  - BSP of the input model
  - Multiresolution structure
  - Far voxel
- Rendering
  - Selective refinement
  - Occlusion culling
  - Far voxel rendering
- Results
  - Preprocessing
  - Rendering



CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Results

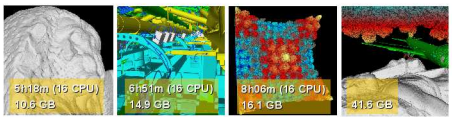
- Tested on extremely complex heterogeneous surface models
  - St.Matthew, Boeing 777, Richtmyer Meshkov isosurf., all at once
- Tested in a number of situations
  - Single processor / cluster construction
  - Workstation viewing, large scale display



CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Results

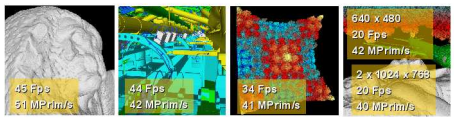
- 1-16 Athlon 2200+ CPU, 3 x 70GB ATA 133 Disk (IDE+NFS)
- 1-20K triangles/sec
  - Scales well, limited by slow disk I/O for large meshes
  - Slow!! (but similar to recent adaptive tessellation methods)
- Avg. triangles per leaf 5K
- Avg. voxels per inner node 2.5K



CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Results

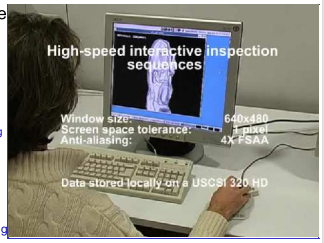
- Xeon 2.4GHz, 70GB SCSI 320 Disk, GeForce FX6800GT AGP 8x
- Window size: from video resolution to stereo projector display
  - St.Matthew, Boeing, Isosurface: 640 x 480
  - All at once: 640 x 480 and Stereo 2 x 1024 x 768
- Pixel tolerance: [Target 1 | Actual ~0.9 | Max ~10]
- Resident set size limited to ~200 MB



CRS4 Visual Computing Group (www.crs4.it/vic/)

Far Voxels
Conclusions

- General purpose technique that targets many model kinds
  - Seamless integration of
    - multiresolution
    - occlusion culling
    - out-of-core data management
  - High performance
  - Scalability
- Main limitations
  - Slow preprocessing
  - Non-photorealistic rendering quality



Intel Xeon 2.4GHz 1GB, GeForce 6800GT AGP8X

CRS4 Visual Computing Group (www.crs4.it/vic/)

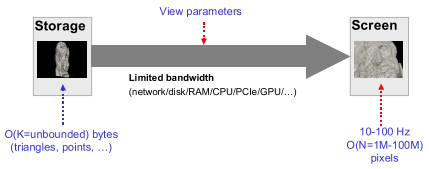
Time for a conclusion, right?

CRS4 Visual Computing Group (www.crs4.it/vic/)

Size matters! Or does it?

A real-time data filtering problem!

- Models of unbounded complexity on limited computers
  - We assume **less data on screen (N) than in model (K →∞)**
  - Need for **output-sensitive** techniques (O(N), not O(K))



Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Application domains / data sources

Local Terrain Models
2.5D – Flat – Dense regular sampling

Planetary terrain models
2.5D – Spherical – Dense regular sampling

Laser scanned models
3D – Moderately simple topology – low depth complexity - dense

CAD models
3D – complex topology – high depth complexity – structured - 'ugly' mesh

Natural objects / Simulation results
3D – complex topology + high depth complexity + unstructured/high frequency details

- Many important application domains
- Models exceed
  - O(10^8-10^9) samples
  - O(10^9) bytes
- Varying
  - Dimensionality
  - Topology
  - Sampling distribution

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Application domains / data sources

Local Terrain Models
2.5D – Flat – Dense regular sampling

Planetary terrain models
2.5D – Spherical – Dense regular sampling

Laser scanned models
3D – Moderately simple topology – low depth complexity - dense

CAD models
3D – complex topology – high depth complexity – structured - 'ugly' mesh

Natural objects / Simulation results
3D – complex topology + high depth complexity + unstructured/high frequency details

- "Well behaved" surfaces
- Multiresolution dominates visibility
- Good results with surface based methods based on sequences of local modifications
- GPU-MT / TetraPuzzles / … already fast/good enough

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

Application domains / data sources

Local Terrain Models
2.5D – Flat – Dense regular sampling

Planetary terrain models
2.5D – Spherical – Dense regular sampling

Laser scanned models
3D – Moderately simple topology – low depth complexity - dense

CAD models
3D – complex topology – high depth complexity – structured - 'ugly' mesh

Natural objects / Simulation results
3D – complex topology + high depth complexity + unstructured/high frequency details

- Highly complex surfaces
- Visibility needs to be tightly combined with LODs
- Need to go to volumetric models
- Far Voxel is a state-of-the-art solution
- Still not the final word…

Session 6: Multi-Resolution Techniques for Exploring Extremely Large and Complex Surfaces
Enrico Gobbetti, SIGGRAPH 2007 Tutorial on "State of the art in massive models visualization"

CRS4 Visual Computing Group (www.crs4.it/vic/)

So many things, so little time…

- More info:
  http://www.crs4.it/vic/
  http://vcg.isti.cnr.it/
- Q&A: Your turn…

Bibliography

Fabio Bettio, Enrico Gobbetti, Fabio Marton, and Giovanni Pintore. Multiresolution Visualization of Massive Models on a Large Spatial 3D Display. In Proc. Eurographics Symposium on Parallel Graphics and Visualization, May 2007. Eurographics Association. Aire-la-Ville, Switzerland.

Enrico Gobbetti, Fabio Marton, Paolo Cignoni, Marco Di Benedetto, and Fabio Ganovelli. C-BDAM - Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering. Computer Graphics Forum, 25(3), September 2006. Proc. Eurographics 2006.

Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Batched Multi Triangulation. In Proceedings IEEE Visualization. Pages 207-214. IEEE Computer Society Press, October 2005.

Enrico Gobbetti and Fabio Marton. Far Voxels - A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms. ACM Transactions on Graphics, 24(3): 878-885, August 2005. Proc. SIGGRAPH 2005.

Enrico Gobbetti and Fabio Marton. Layered Point Clouds - a Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-Sampled Models. Computers & Graphics, 28(6): 815-826, December 2004.

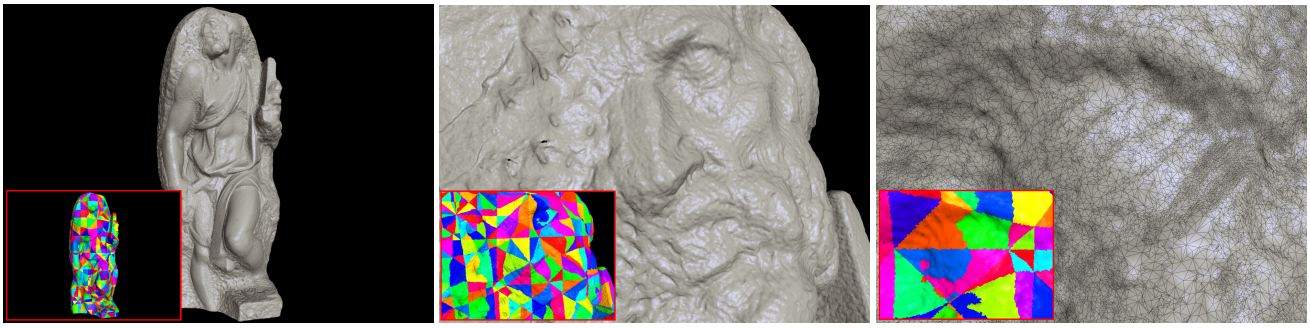
Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Adaptive TetraPuzzles - Efficient Out-of-core Construction and Visualization of Gigantic Polygonal Models. ACM Transactions on Graphics, 23(3): 796-803, August 2004. Proc. SIGGRAPH 2004.

Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM). In Proceedings IEEE Visualization. Pages 147-155. IEEE Computer Society Press, October 2003.

Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization. Computer Graphics Forum, 22(3): 505-514, September 2003. Proc. Eurographics 2003.

Adaptive TetraPuzzles: Efficient Out-of-Core Construction and Visualization of Gigantic Multiresolution Polygonal Models

Paolo Cignoni ISTI - CNR *
Fabio Ganovelli ISTI - CNR
Enrico Gobbetti CRS4 †
Fabio Marton CRS4
Federico Ponchio ISTI - CNR
Roberto Scopigno ISTI - CNR



(a) 1473 patches (2179K triangles) (b) 1221 patches (1855K triangles) (c) 625 patches (991K triangles)

Figure 1: **View-dependent rendering of the St. Matthew dataset.** The full resolution model contains 373 million triangles and is inspected at over 40 fps on a commodity PC platform. The main images present the mesh rendered with Gouraud shading using 4x Gaussian Multisampling on a 1280x1024 window, while the small inset figures depict the adaptive mesh structure with a different color for each patch. The rightmost image also shows the adaptive triangulation.

Abstract

We describe an efficient technique for out-of-core construction and accurate view-dependent visualization of very large surface models. The method uses a regular conformal hierarchy of tetrahedra to spatially partition the model. Each tetrahedral cell contains a precomputed simplified version of the original model, represented using cache coherent indexed strips for fast rendering. The representation is constructed during a fine-to-coarse simplification of the surface contained in diamonds (sets of tetrahedral cells sharing their longest edge). The construction preprocess operates out-of-core and parallelizes nicely. Appropriate boundary constraints are introduced in the simplification to ensure that all conforming selective subdivisions of the tetrahedron hierarchy lead to correctly matching surface patches. For each frame at runtime, the hierarchy is traversed coarse-to-fine to select diamonds of the appropriate resolution given the view parameters. The resulting system can interatively render high quality views of out-of-core models of hundreds of millions of triangles at over 40Hz (or 70M triangles/s) on current commodity graphics platforms.

CR Categories: I.3.3 [Computer Graphics]: Picture and Image Generation—; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—.

Keywords: Out-Of-Core Algorithms, Level of Detail

*ISTI-CNR, Via Moruzzi 1, 56124 Pisa Italy www:http://vcg.isti.cnr.it/ e-mail:first.last@isti.cnr.it
†CRS4, POLARIS Edificio 1, 09010 Pula, Italy www:http://www.crs4.it/ e-mail:first.last@crs4.it

1 Introduction

The need for interactively inspecting very large surface meshes, consisting of hundreds of millions of polygons, arises naturally in many application domains, including 3D scanning, geometric modeling, and numerical simulation. However, despite the rapid improvement in hardware performance, these meshes largely overload the performance and memory capacity of state-of-the-art graphics and computational platforms. A wide variety of simplification methods and dynamic multiresolution models have been proposed to face the problem, but, unfortunately, none of them is able to perform both scalable simplification and interactive view-dependent visualization of very large meshes without imposing a lossy decimation of the original dataset [Lindstrom 2003]. This is mainly because current methods, heavily CPU bound, are unable to generate model updates at full GPU speed and to efficiently communicate them to the graphics hardware through preferential data paths. This CPU/GPU gap is doomed to widen, since CPU processing power grows at a much slower rate than that of the GPU.

The original contribution of this paper is a solution for interactive and accurate visualization of very large surface models on consumer graphics platforms. The underlying idea of the proposed method is to depart from current point- or triangle-based multiresolution models and adopt a patch-based data structure, from which view-dependent conforming mesh representations can be efficiently extracted by combining precomputed patches. Since each patch is itself a mesh composed of a few thousand triangles, the multiresolution extraction cost is amortized over many graphics primitives, and CPU/GPU communication can be optimized to fully exploit the complex memory hierarchy of modern graphics platforms.

The method uses a conformal hierarchy of tetrahedra generated by recursive longest edge bisection to spatially partition the model. Each tetrahedral cell contains a precomputed simplified version of the original model. The representation is constructed off-line during a fine-to-coarse parallel out-of-core simplification of the surface contained in diamonds (sets of tetrahedral cells sharing their longest edge). Appropriate boundary constraints are introduced in the simplification process to ensure that all conforming selective subdivisions of the tetrahedron hierarchy lead to correctly matching surface patches. At run-time, selective refinement queries based on projected error estimation are performed on the external memory

tetrahedron hierarchy to rapidly produce view-dependent continuous mesh representations by combining precomputed patches. The resulting technique, dubbed *Adaptive TetraPuzzles (ATP)* since it heavily builds on the composition properties of conformal hierarchies of tetrahedra, has the following properties: it is fully adaptive and is able to retain all the original topological and geometrical detail even for massive datasets; it is not limited to meshes of a particular topological genus or with a particular subdivision connectivity and preserves geometric continuity of variable resolution representations at no run-time cost; it is strongly GPU bound and over one order of magnitude faster than existing adaptive tessellation solutions on current PC platforms, since its patch based structure successfully exploits on-board caching, cache coherent stripification, compressed out of core representation and speculative prefetching for efficient rendering on commodity graphics platforms with limited main memory; high quality simplified representations can be constructed with a distributed out of core simplification algorithm.

As highlighted in the short overview of related work (sec. 2), while certain other algorithms share some of these properties, they typically do not meet the capability of our method in all of the areas. The details of the proposed data structure are presented in section 3, while section 4 describes algorithms for view-dependent refinement and rendering, and section 5 introduces an efficient distributed out-of-core technique for constructing a multiresolution model using a generic high quality simplification algorithm. The efficiency of the approach has been successfully evaluated with a number of large models, including a massive 373 million polygon model of Michelangelo's St. Matthew (section 6).

2 Related Work

Rapidly rendering adaptive representations of large models is a very active research area. In the following, we will discuss the approaches that are most closely related with our work. Readers may refer to recent surveys (e.g., [Chiang et al. 2003]) for further details.

Out-of-core mesh simplification. Various techniques have been presented to face the problem of huge mesh simplification. With the exception of memoryless clustering approaches [Lindstrom 2000; Lindstrom 2003] and stream-based methods [Wu and Kobbelt 2003; Isenburg et al. 2003], most of these techniques, such as Hoppe's hierarchical method for digital terrain management [1998] and the octree based structure OEMM [Cignoni et al. 2003a], are based on some kind of mesh partitioning and subsequent independent simplification. Hoppe hierarchically divides the mesh in blocks, simplifies each block while freezing borders and then traverses the block hierarchy bottom-up by merging sibling cells and again simplifying. In this approach some of the borders remain unchanged until the very last simplification step. OEMM avoids this kind of problem, but it does not build a multiresolution structure. On the other hand, BDAM [Cignoni et al. 2003c] allows both the independent processing of small sub-portions of the whole mesh and the construction of a multiresolution structure, but is limited to height fields. Our work generalizes this approach to arbitrary surfaces, and parallelizes the simplification process in order to efficiently build a multiresolution structure for very large meshes.

View-dependent triangulations. The vast majority of view-dependent simplification methods for general meshes are based on constructing a graph of possible refinement/coarsening operations at the vertex or triangle level. Early methods used edge collapse [Xia and Varshney 1996; Hoppe 1997] or vertex clustering [El-Sana and Varshney 1999; Luebke and Erikson 1997] as primitive operations, and assumed in-core hierarchy construction, limiting their applicability. Few techniques have been presented for both construction and rendering from external memory. Hoppe's hierarchical terrain management method [1998] was an early example, later extended to arbitrary meshes [Prince 2000]. More recently, El-Sana and Chiang [2000] proposed a technique for segmenting the surface and ordering edge collapses to handle block boundaries without explicitly imposed constraints. Both methods have only been tested on models of a few million polygons, and their scalability is unclear. Lindstrom [2003] recently proposed a scheme for out-of-core construction and visualization of multiresolution surfaces based on vertex clustering on a rectilinear octree. While it significantly improves over earlier approaches, it is still unable to retain the fidelity of the original mesh and is heavily CPU bound at rendering time, with a peak performance of 2M triangles/s and asynchronous hierarchy updates at no more than 1 frame/s.

Efficient host-to-graphics communication. All adaptive mesh generation techniques spend a great deal of rendering time to compute the view-dependent triangulation. For this reason, many authors have proposed techniques to alleviate popping effects due to small triangle counts [Cohen-Or and Levanoni 1996; Hoppe 1998] or to amortize construction costs over multiple frames [Duchaineau et al. 1997; Hoppe 1997; Lindstrom 2003]. Our technique reduces instead the per-triangle workload by composing at run-time pre-assembled optimized surface patches. The idea of grouping together sets of triangles in order to alleviate the CPU/GPU bottleneck was presented also in the RUSTIC [Pomeranz 2000], CABTT [Levenberg 2002], and BDAM [Cignoni et al. 2003c] data structures for terrains, and HLOD [Erikson et al. 2001] for general environments. RUSTIC and CABTT are extensions of the ROAM algorithm in which subtrees of the ROAM bintree are cached and reused during rendering. BDAM constructs instead a forest of hierarchies of right triangles, in which each node is a general triangulation of a small surface region. These methods produce adaptive conforming surfaces but are hard to generalize to surfaces with arbitrary topology. HLOD improves instead the classic LOD scene graph by providing multiple precomputed levels of details not only for each model but also for entire subtrees. While related to our method, HLOD focuses on the run-time handling of a large number of small unconnected objects, typical of large CAD assemblies.

Point rendering approaches. An alternative to mesh refinement is to use multi-resolution hierarchies of point primitives to render highly complex scenes in output-sensitive time [Rusinkiewicz and Levoy 2000]. Since current rendering hardware is optimized for triangle rendering, high quality filtered point splatting requires considerable effort, and high visual quality has often to be sacrificed for rendering speed. The latest approaches try to solve this problem by exploiting the programmability features of modern GPUs, leading to impressive peak performances that range from 50M points/second for low quality rendering with unfiltered splats [Dachsbacher et al. 2003] to 10M points/second for high-quality filtering [Botsch and Kobbelt 2003] for in-core models of a few million polygons. By using cache coherent triangle strip primitives, we are able to exceed such rates even for out of core models that are two orders of magnitude larger.

Hierarchies of tetrahedra. In the scientific visualization and finite element literature, much research has been devoted to nested tetrahedral meshes generated by recursive subdivision (see [Cignoni et al. 2003b] for a recent survey). Our data structure is constructed from a recursive partitioning of the input dataset guided by the same regular tetrahedron bisection rule that is often used for modeling and viewing regular 3D grids. We are interested, however, in the space partitioning induced by the multi-level tetrahedralization, rather than in extracting values at mesh vertices as in numerical methods [Maubach 1995] or scientific visualization [Gregorski et al. 2002].

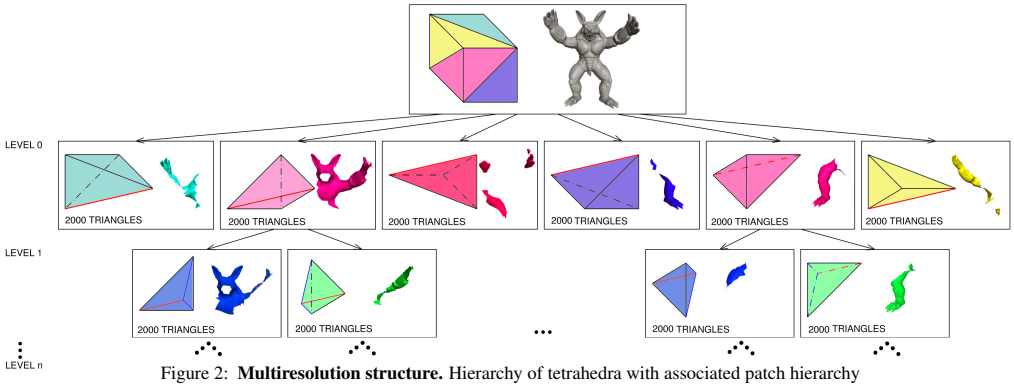
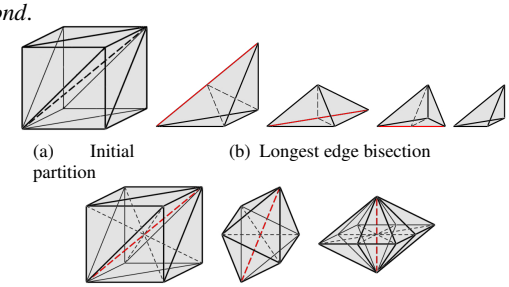


Figure 2: **Multiresolution structure.** Hierarchy of tetrahedra with associated patch hierarchy

3 Multiresolution model

A multiresolution surface model supporting view-dependent rendering must encode the steps performed by a mesh refinement or coarsening process in a compact data structure from which a virtually continuous set of variable-resolution meshes can be efficiently extracted. Our approach is based on the idea of moving the grain of the multiresolution surface model up from points or triangles to small contiguous portions of mesh. The benefits of this approach are that the workload required for a unit refinement/coarsening step is amortized on a large number of triangle primitives, and that the small patches can be optimized off-line for best performance.

To avoid making assumptions on a particular topological genus or subdivision connectivity of the input mesh, we exploit the partitioning induced by a recursive volumetric subdivision of the mesh bounding volume in a *hierarchy of tetrahedra* (see figure 3). The partitioning consists of a binary forest of tetrahedra, whose roots correspond to six tetrahedra around a major box diagonal and whose other nodes are generated by tetrahedron bisection. This operation consists in replacing a tetrahedron σ with the two tetrahedra obtained by splitting σ at the midpoint of its longest edge by the plane passing through such point and the opposite edge in σ. To guarantee that a conforming tetrahedral mesh is always generated after a bisection, all the tetrahedra sharing their longest edge with σ are split at the same time. Such a cluster of tetrahedra is called *diamond*.

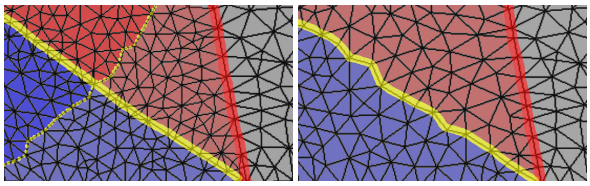


(a) Initial partition (b) Longest edge bisection

(c) Diamond types

Figure 3: **Hierarchy of tetrahedra for space partitioning.** The longest edge is highlighted in red, while next level edges are light-dashed.

The hierarchy of tetrahedra structure has the important property that, by selectively refining or coarsening it on a diamond by diamond basis, it is possible to extract conforming variable resolution volumetric mesh representations.We exploit this property to construct a level-of-detail structure for the surface of the input model. The basic idea (see figure 2) is to generate from the tetrahedral structure a hierarchy of surface representations. We first partition the input model triangles among the leaf tetrahedra. We then recursively associate to each non-leaf tetrahedron a simplification, up to a given triangle count, of the portion of the mesh contained in its two children, along with all the information required for evaluating view dependent errors.



(a) level i (b) level $i-1$

Figure 4: **Generating conforming triangulations.** The four patches at the left of figure 4(a) are part of the same diamond, and are simplified into the two patches at the left of figure 4(b) when coarsening the mesh. The generation of a conforming triangulation is ensured by locking the vertices shared with the neighboring diamond (highlighted in red), and by consistently simplifying the vertices shared by different patches in the diamond (highlighted in yellow).

The above *diamond-by-diamond* property is sufficient for guaranteeing conforming tetrahedral meshes, but, when switching from tetrahedra to the small patches associated to them, the correct connectivity along borders of patches at different simplification levels must be guaranteed by imposing appropriate constraints during simplification. This is efficiently done by carrying out bottom-up construction on a diamond by diamond basis. When building a simplified representation for all tetrahedra in a given diamond, the borders of a patch contained in a tetrahedron are of three possible kinds: (a) *diamond-internal borders*, i.e., borders connecting it with other triangles contained in tetrahedra of the same diamond; (b) *diamond-external borders*, i.e., borders connecting it with triangles contained in tetrahedra of other diamonds; (c) *original borders*, i.e., borders of the original mesh. Since all tetrahedra in a diamond are, by definition, always split/merged at the same time, handling borders of kind (a) just requires that the mesh contained in a diamond is simplified as a single unit, while borders of kind (b) need to be kept fixed to ensure connectivity with all possible neighbors, and borders of type (c) do not need any special handling. The main idea behind these rules is to associate a merge operation of the internal edges of a diamond with the overall simplification of the patch strictly inside the diamond. Thus, when we coarsen our tetrahedral mesh by merging the split-edges inside a diamond, we can safely substitute the patches inside the involved tetrahedra with the ones of the merged diamond: by construction they share the same border and therefore correctly match with the rest of the surface. In this way, we ensure that each mesh composed by a collection of small patches arranged as a correct hierarchy of tetrahedra generates a globally correct surface triangulation (see figure 4). It is worth mentioning that, unlike other hierarchical simplification approaches [Hoppe 1998; Prince 2000], these constraints have little effect on overall simplification quality, since constrained vertices alternate from diamond-internal to diamond-external throughout the hierarchy, and are locked only when in diamond-external state. Moreover, the fact that each diamond is simplified independently can be exploited, see section 5, to design a parallel out-of-core high quality simplification algorithm.

4 View-dependent rendering

The adaptive rendering algorithm is based on a top-down refinement of the tetrahedra hierarchy, designed to fully exploit the rendering capabilities of modern graphics accelerators through batched primitive rendering. Its main components can be considered a generalization of the BDAM approach [Cignoni et al. 2003c] to arbitrary surfaces.

Data organization. Since the algorithm is designed to work in a standalone PC architecture (as opposed to a distributed, network-based solution), we assume that all data is stored locally on a secondary storage unit visible to the rendering engine. Our approach, similarly to recent large scale terrain visualization approaches [Lindstrom and Pascucci 2002; Cignoni et al. 2003c], is based on optimizing the data layout to improve memory coherency and on accessing external memory geometry data through system memory mapping functions, demanding to the operating system the task of loading, when needed, the requested data. To maximize memory locality, we have thus chosen to represent our nested subdivision as a forest of binary trees, and to extract conforming meshes without requiring neighbor finding we employ a saturation technique [Ohlberger and Rumpf 1998]. Therefore, each tree is stored as a memory mapped linear array, and each of its nodes, corresponding to a particular tetrahedron, contains just the following information: a reference to the associated patch data (vertex attributes and connectivity) in a patch repository; the tight bounding sphere and bounding cone of normals for the patch; the saturated model space error and bounding sphere of the neighborhood; the index of child nodes in the linear arrays, which correspond to the two tetrahedra generated by bisection. To minimize the number of page faults, data storage order reflects traversal order. All data in the tree and in the corresponding patch repository is therefore sorted by level, then by geometric proximity, by ordering the nodes in a given level by increasing Morton code [Samet 1990] of their center point. The external size of geometric representation is also reduced by storing each patch in compressed form (see section 5).

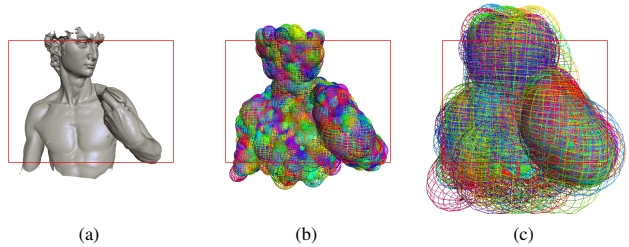


Figure 5: **Refinement and culling.** The model is refined to a screen space error tolerance of one pixel and is culled against the red rectangle. The spheres used for culling are in sub-figure 5(b), while the saturated spheres used for refinement are in figure 5(c)

Refinement algorithm. With this structure, variable resolution rendering is implemented by simple stateless top-down traversals of the binary trees, that combine view-frustum, backface, and contribution culling (see figure 5). As we recurse the hierarchy, we test if the current node is invisible or fully backfacing by checking the tight bounding sphere and cone of normals of the associated patch against the current view volume. If so, we simply stop, culling away the entire branch of the tree. If the node is potentially visible, we test whether its patch is an accurate enough representation by measuring its saturated screen space error. If so, we can render the associated patch, otherwise we continue the recursive refinement with the node's children.

Saturated screen space error is the quantity that guides refinement. We obtain a consistent upper bound by measuring the apparent size of a sphere with diameter equal to the saturated object space errors and centered at the saturated bounding sphere point closest to the viewpoint. The refinement condition, once the point closest to the viewpoint is found, requires only one multiplication to check if the ratio of error sphere radius to distance is larger than the screen space threshold. The extraction of a consistent mesh is ensured by constructing the saturated object space errors/spheres so that they are equal for all the tetrahedra in a diamond and monotonically decreasing when descending the hierarchy. Since the projection to screen space is also monotonic, the diamond composition property ensures that we always extract a conforming volumetric mesh, and our patch construction rules ensure that we generate a globally correct surface triangulation.

Host to graphics processing unit communication. To take advantage of spatial and temporal coherency, it is worth spending time to build an optimal rendering representation for each patch, that can be efficiently reused on subsequent frames, instead of using direct rendering each time. This is the only way to harness the power of current graphics architectures, that heavily rely on extensive on board data caching. We have thus combined our refinement method with a memory manager based on a simple LRU strategy, that explicitly manages graphics board memory, using OpenGL's *Vertex Buffer Objects* extension. Each time we need to render a patch, we reuse the cached version if present, otherwise we render it and cache its representation in place of the oldest one. The transformation from compressed external memory representation to an efficient graphics format happens only at cache faults. The primitive geometric element is a patch composed of multiple triangles, that is heavily optimized during pre-processing using cache-coherent tri-stripping. Since we use an indexed representation, the post-texture-and-lighting cache of current GPUs is fully exploited.

Speculative prefetching. Since the disk is by far the slowest hardware component, we can further hide data access latency by prefetching from compressed external memory the geometry that will soon be accessed. The prefetching routine, that may be executed in parallel to the rendering thread or sequentially as an idle task, executes the same refinement algorithm as the adaptive rendering code, taking as input the predicted camera position instead of the current one. When the refinement terminates, instead of rendering the patches, it simply checks whether the required graphics objects are in pages already in core. If not, it advises the operating system kernel that the pages containing their representation will likely be accessed in the near future and that it would be advantageous to asynchronously read them ahead. This technique is easy to implement on Linux with the `mincore/madvise` system calls. Furthermore, the main rendering needs not to be aware of the prefetching component, and we exploit the extensive performance optimizations of the operating system's virtual memory manager, such as reordering of requests to reduce seek access time and overlapping of computation and disk access.

5 Construction

The off-line component of our method constructs a multiresolution structure starting from a high resolution mesh. As input, we assume that the mesh is represented as a *triangle soup*, i.e., a flat list of triangles with direct vertex information, and that a list of boundary vertices is also available. This representation is commonly employed for out-of-core methods, and can be derived in a I/O efficient way from the more common indexed mesh representation using standard external memory graph techniques [Chiang et al. 1995].

5.1 Mesh Partitioning

The first phase – mesh partitioning – generates a binary forest of tetrahedra, whose roots partition the mesh bounding box and whose leaves contain less than a predefined number of mesh triangles.

The forest is built in a top-down fashion, through recursive insertion of mesh triangles by starting from an initial subdivision of the mesh bounding box into six tetrahedra around a major box diagonal. When a new triangle is inserted, we locate the leaf that contains its center point and, if the number of triangles already contained in it does not exceed the maximum, we simply insert the new one into the associated triangle bucket. Otherwise, we refine the hierarchy by tetrahedron bisection and recursively continue the insertion procedure. Each time a tetrahedron is split, all the triangles in the associated bucket are reassigned to its children by a recursive application of the insertion procedure. The end result is a tetrahedron graph, that describes the subdivision structure using a DAG of diamonds [Pascucci 2002], and a set of triangle buckets associated with leaf tetrahedra that cover the mesh. The graph, rather small since each node typically contains a few thousand triangles, is for efficiency reasons maintained in main memory, while triangle buckets are stored in secondary memory.

This simple partitioning scheme, that clusters triangles solely based on the location of their center point, does not adapt to surface features, and, as for all spatial clustering methods, could lead to patches with exceedingly complex boundaries. Even though we have not found this to be a problem in practice, we plan to improve partitioning in a second pass, that adaptively reassigns triangles among leaf tetrahedra after the first partitioning. This kind of approach demonstrated its efficiency in recent out-of-core simplification methods (e.g., [Shaffer and Garland 2001]).

5.2 Level-of-detail hierarchy construction

The second and final phase – simplification – completes the volumetric structure with a hierarchy of surface representations by recursively associating to each non-leaf tetrahedron a fixed triangle count simplification of the portion of the mesh contained in its two children, along with all the information required for evaluating view dependent errors. This is efficiently done by carrying out bottom-up construction on a diamond by diamond basis. For leaf diamonds, we retain all the fidelity of the original mesh and directly produce an optimized representation for each tetrahedron from the stored triangle buckets. For non-leaf diamonds, we retrieve from the repository the triangles associated to the children of all involved tetrahedra, merge them in a single mesh, that is then simplified so that each involved tetrahedron contains less than a predefined number of triangles. The repository is then updated by erasing the buckets associated to child tetrahedra before saving parent ones.

Diamond simplification. As explained in section 3, to ensure that the patches contained in the diamond's tetrahedra match with all the possible neighbors at different simplification levels, we lock all the vertices on the diamond external boundary. These are easily identified, without maintaining special connectivity information, as the end points of all edges shared by only one triangle of the diamond patch, which were not part of the original model boundary. To avoid introducing new boundary points and simplify bookkeeping, we also constrain original boundary vertices to be removed only by a collapse with another original boundary vertex. Note that any simplification technique can be adopted as long as it allows the choice of the number of triangles for different regions of the output mesh and the specification of vertex constraints. This means for example that, if needed by a particular application, it is possible to exactly preserve the original mesh topology and therefore preserve any existing parametrization. On the other hand, it is also possible to decide to simplify more aggressively by closing holes or clustering vertices. In our case, we have implemented a variation of Hoppe's method for simplifying meshes with appearance attributes [Hoppe 1999], that combines a quadric error metric with regularization penalties for improving sampling regularity and mesh quality in regions of null quadric error. Moreover, to ensure that each tetrahedron in the diamond is simplified to a predefined number of triangles, the simplifier maintains a separate collapse queue for each tetrahedron. We put a edge collapse in the queue of each of the tetrahedrons where it removes a triangle. During iterative simplification, collapses are taken in order of increasing cost from the queues of the tetrahdera that still need to be simplified, and all queues are maintained up-to-date after each simplification step.

Errors and bounds. After simplification, model space errors, bounding spheres and normal cones have to be computed and saturated for each tetrahedron to complete the structure. For errors, we have currently taken the common approach of deriving the model space error ε directly from the quadric metric ε_q (see, e.g., [Lindstrom 2003]). We employ the simple formula $\varepsilon = s\sqrt[3]{\varepsilon_q}$, where s is an empirical scale factor for converting to world units (see, e.g., [Lindstrom 2003]). The scale factor is determined prior to rendering time by finding the smallest value of s leading to no image difference in a fixed number of random views, when setting the screen space tolerance below 1 pixel. Bounding spheres and normal cones, are, instead, computed and saturated using optimal methods from computational geometry for finding the minimum enclosing ball of points (for leafs) and minimum enclosing ball of balls (for all other nodes) [Fischer and Gärtner 2003].

Conversion into final format. Each patch is stored on disk in a compressed representation from which a version optimized for efficient rendering can be rapidly extracted. On current graphics architectures, the most efficient mesh representation is the *cache coherent indexed stripification*. In this representation, vertex attributes are stored in vertex arrays, and triangulation topology is specified with a generalized triangle strip ordered such that vertex cache miss rate is minimized. We greedily compute this ordering by first decomposing the mesh into strips of length approximately equal to vertex cache size, and, then, starting with the strip with the maximum number of border vertices, incrementally adding the other strips, always choosing the one with the minimum cache miss to strip length ratio. For disk storage, topology is compressed using a mesh encoding scheme preserving stripification [Isenburg 2001], while vertex attributes are optionally quantized and entropy encoded. For this paper, that emphasizes the ability of our method to preserve all the original details, we chose a 3x24 bits/position and 32 bit/normal quantization, which correspond to no loss.

Network parallel construction. The hierarchy construction phase dominates, by far, the overall processing cost. The whole process is however inherently parallel, because the grain of the individual diamond processing tasks is very fine and synchronization is required only at the completion of each level. In a network parallel implementation, the coordinator process traverses all the diamonds bottom up and by geometric proximity, distributes the diamond processing job to a number of workers, which execute it and send the result back to the coordinator for updating the repository and generating output file. Load balancing is ensured if the coordinator initially seeds each worker with a single diamond and subsequently always sends a new job request to the work that sent back a result. The generation of the output file in the correct order can be ensured with a small buffer for handling out-of-sync output requests. In this solution, the main memory required for each worker is that required for processing a single diamond, while the coordinator simply needs to reserve enough memory for holding out-of-sync requests.

Figure 6: **Test models.** The main images show the models as presented to the user during interactive inspection session, while the inset images illustrate the mesh structure. Left: Bonsai isosurface (6.4M triangles); Middle: David 2mm (8.3M triangles) and 1mm (56M triangles); Right: St. Matthew 0.25mm (373M triangles).

6 Results

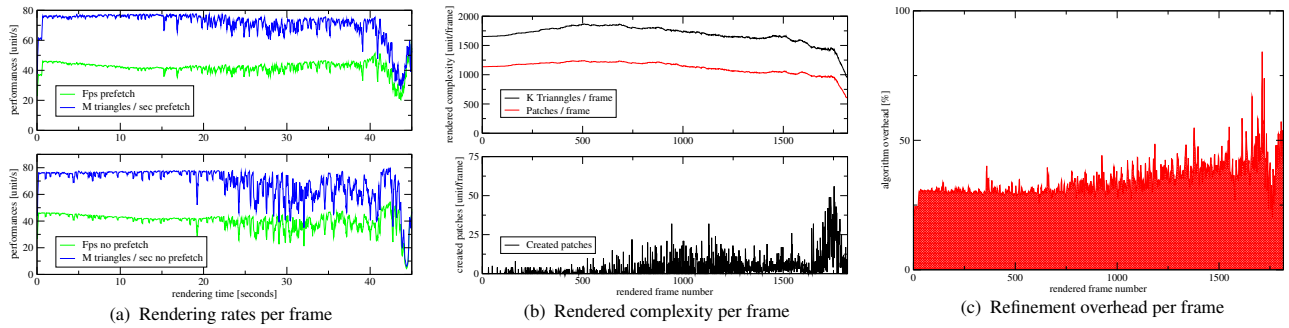
An experimental software library and a rendering application supporting the technique have been implemented on Linux using C++ with OpenGL and the MPICH MPI implementation. We have extensively tested our system with a number of large surface models. The quantitative and qualitative results discussed here are restricted to the freely available models of figure 6.

Model	Triangles	CPU	Partition	Bottom-up construction			Total time	Disk usage (MB)		
				I/O	Process	Total		in	tmp	out
Bonsai	6,317,116	1+1	45	42	1,654	1,696	1,741	289	808	76
		1+4	40	63	435	498	538			
		1+8	41	44	239	283	324			
		1+14	42	63	114	177	219			
David 2mm	8,277,479	1+1	59	93	3,583	3,676	3,735	379	1,059	158
		1+4	60	94	836	930	990			
		1+8	62	144	366	510	572			
		1+14	62	148	216	364	426			
David 1mm	56,230,343	1+1	477	2,335	21,687	24,022	24,499	2,574	6,850	967
		1+4	501	2,299	3,981	6,280	6,781			
		1+8	583	2,086	1,758	3,844	4,427			
		1+14	477	2,238	879	3,117	3,594			
St. Matthew	372,767,445	1+1	3,234	15,722	73,299	89,021	92,255	17,063	41,289	5,887
		1+4	3,400	16,639	19,917	36,556	39,956			
		1+8	3,201	15,876	11,020	26,896	30,097			
		1+14	3,215	16,863	7,712	24,575	27,790			

Table 1: **Numerical results for out-of-core construction.** Tests performed on a network of PCs. All times are in seconds.

Preprocessing. Table 1 lists numerical results for our out-of-core preprocessing method for a number of runs on all the test datasets. The tests were executed on a moderately loaded network of PCs running Linux 2.4. Each PC has two CPU Athlon 2200+ CPUs, 1GB DDR memory, a 70GB ATA 133 hard disk, and a Ethernet 100 Mb/s network connection. We constructed all multiresolution structures with a prescribed maximum leaf size of 4000 triangles/tetrahedron for the partitioning phase and an average non-leaf size of 2000 triangles/tetrahedron for the bottom-up construction phase. To test parallel performance, all tests were repeated with 1, 4, 8, and 14 workers. Overall processing times range from about 3K-4K triangles/s for 1 CPU to 15K-30K triangles/s for 14 CPU. As a point of reference, current state-of-the-art high quality out-of-core simplification methods achieve simplification rates up to 70K triangles/s [Chiang et al. 2003]. Our speed is currently slower. This is mainly because we generate a full multiresolution structure, as opposed to a single small model, and simplification is only a single step of diamond processing, that also includes cache-coherent stripification, mesh compression, and optimal bound computation, each costing as much as simplification. As the number of CPUs increases, construction time, initially dominated by diamond processing, starts to be dominated by raw I/O. The almost linear reduction in processing time shows the efficiency of the distributed approach. The large I/O overhead is due to the repeated access to the temporary triangle repository during bottom-up construction, stored on a slow IDE disk. Similarly to competing methods [Lindstrom 2003], temporary storage size is a constant multiple (roughly a factor of 3) of input size, since we need to store at most the partitioning of the input in the leaves of the tetrahedra hierarchy. Our current version uses a pessimistic fixed-size bucket per leaf, and stores triangles in raw uncompressed form. We anticipate that reducing the size of the repository with compressed dynamically sized buckets would radically decrease I/O cost. Not included in the table is the maximum resident memory usage of the method, which is low and constant for workers (26 MB each) and variable for the master process, due to the in-core graph layout data structure and to disk buffer caches used by the operating system (for a maximum of 280 MB on all runs).

Adaptive rendering. We evaluated the rendering performance of the technique on a number of inspection sequences on all test datasets, using a Linux PC with a Intel Xeon 2.4 GHz, 2GB RAM, a Seagate ST373453LW 70 GB ULTRA SCSI 320 hard drives, AGP 8x and NVIDIA GeForce FX 5800 Ultra graphics. The quantitative results presented here in details were collected during a 45 seconds inspection of the largest model (the 373M triangles reconstruction of Michelangelo's St. Matthew). The session, performed using a window size of 800x600 pixels, hardware full scene antialiasing (4x Gaussian Multisampling), and a screen tolerance of ± 2 pixels (i.e., a refinement epsilon of 4), was designed to be representative of typical mesh inspection tasks and to heavily stress the system, and includes rotations and rapid changes from overall views to extreme close-ups. To further emphasize the quality of the view-dependent simplification, we used glossy material properties and a single off-center positional light placed slightly above and to the right of the camera. The qualitative performance of our adaptive renderer is also illustrated in an accompanying video, that shows live recordings of the analyzed flythrough sequence, and of similar sequences with the other datasets. In all cases, during live sessions there were practically no visible artifacts due to adaptive rendering, since the error measure, even though empirically derived, is very conservative. To fully test our out-of-core data management components, the benchmarks were started with all data off core and disk buffers flushed. During the entire walkthrough, the resident set size of the application is maintained at roughly 144 MB, i.e. less than 3% of out-of-core data size, demostrating the effectiveness of out-of-core data management. Figure 7(a) illustrates the rendering performance of the application, both with and without speculative prefetching. The speculative prefetching version executed an additional refinement step per frame to prefetch pages that are likely to be accessed in the near future, using a linear prediction of camera position with a half a second look-ahead. The prefetching version is smoother, due to the success of prefetching in hiding the latency due to page faults. The only noticeable jitters with the prefetching version (visible in the graph near the end of the sequence) correspond to rapid accelerations of the path while zooming. In the prefetch-



(a) Rendering rates per frame (b) Rendered complexity per frame (c) Refinement overhead per frame

Figure 7: **Rendering Performance Evaluation.** To fully test our out-of-core data management components, benchmarks were started with all data off core and disk buffers flushed.

ing version, we were able to sustain an average rendering rate of over 70 millions of triangles per second, with peaks exceeding 78 millions. By comparison, on similar machines, Lindstrom's [2003] multiresolution vertex clustering method's peak performance was measured at roughly 3 millions of triangles per second, with a representation update latency of up to 1s, even though the model was radically downsampled during preprocessing to only 3M polygons. The increased performance of our approach is due to the larger granularity of the structure, that amortizes structure traversal costs over many graphics primitives, reduces AGP data transfers through on-board memory management and fully exploits the post-transform-and-lighting cache with optimized indexed triangle strips. This favorably compares also with the latest point rendering approaches, that render in-core models of a few million polygons at 10M-50M points/second depending on filtering quality [Dachsbacher et al. 2003; Botsch and Kobbelt 2003]. The overhead of the prefetching and rendering code, measured by repeating the test without executing OpenGL calls, is only about 30% of total frame time (Fig. 7(c)), and is mostly due to external data access (mainly I/O wait and patch decompression). This demonstrates that we are GPU bound even when handling extremely large out-of-core data sets. Rendered scene granularity is illustrated in figure 7(b): even though the peak complexity of the rendered scenes exceeds 1.8M triangles per frame, the number of rendered graphics primitives per frame remains relatively small, never exceeding 1240 patches per frame, which are maintained on graphics memory. Since we are able to render such complex scenes at high frame rates, it is possible to use very small pixel thresholds, virtually eliminating popping artifacts, without the need to resort to costly geomorphing features. Figure 8 is an example of the extremely detailed representation that can be inspected in real-time using our technique.

7 Conclusions

We have presented an efficient technique for end-to-end out-of-core construction and view-dependent visualization of very large surface models on commodity graphics platforms. The proposed solution consists in an innovative combination of volumetric subdivision, mesh compression and simplification, out-of-core data management, and batched rendering techniques, resulting in an unprecedented spatiotemporal quality in the interactive rendering of massive models.

Besides improving the proof-of-concept implementation, we plan to extend the presented approach in a number of ways. In particular, we plan to explore more aggressive and lossy compression techniques and to investigate error metrics taking into account the effect of simplification on the shading of the surface. We are also currently incorporating occlusion culling techniques, useful for datasets with a high depth complexity, and we plan to introduce more sophisticated shading/shadowing techniques. Given the

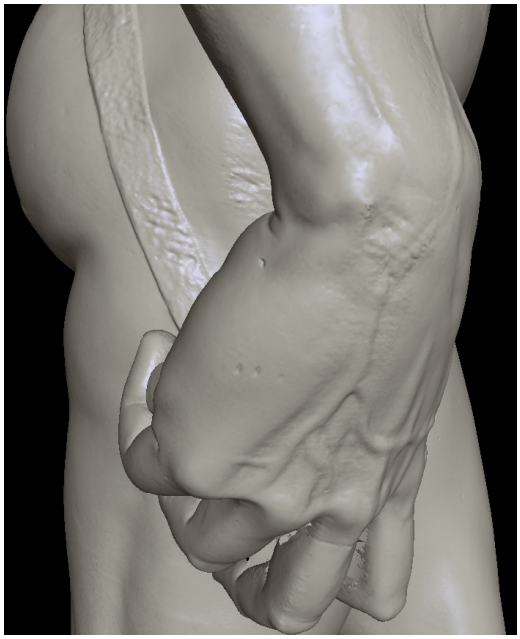


Figure 8: **David 1mm hand close-up.** Model rendered at ± 1 pixel screen tolerance with 841 patches and 1172K triangles at 50 fps on a 1280x1024 window with 4x Gaussian Multisampling, one positional light and glossy material. Note the very fine geometric and illumination details.

performance of the method, we are confident that multi-pass or vertex/fragment program techniques will become readily applicable to gigantic datasets.

The proposed approach, as well as recently introduced techniques such as BDAM [Cignoni et al. 2003c], can be seen as a step in a larger effort of adapting the classic general multiresolution models to modern GPUs, by moving the granularity of the atomic operations of multiresolution models from triangles/points to small surface portions. In our opinion, this is the most promising way to harness the power of current graphics architectures, that heavily rely on extensive on board caching of indexed primitives. Moreover, our results demonstrate that this approach blends also well with other important large model optimizations, such as compression and prefetching techniques. We envision an entire breed of multiresolution mesh rendering algorithms adapted to this kind of mesh representation, and we are currently defining a general framework for experimenting with them.

Acknowledgments. This research is partially supported by the V-PLANET project (EU RTD contract IST-2000-28095). We are grateful to the anonymous reviewers for their thorough work and to the Stanford Graphics Group, the Digital Michelangelo project, and the University of Stuttgart for making benchmark datasets available.

References

BOTSCH, M., AND KOBBELT, L. 2003. High-quality point-based rendering on modern GPUs. In *Proc. Pacific Graphics*, 335–343.

CHIANG, Y.-J., GOODRICH, M. T., GROVE, E. F., TAMASSIA, R., VENGROFF, D. E., AND VITTER, J. S. 1995. External-memory graph algorithms. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 139–149.

CHIANG, Y.-J., EL-SANA, J., LINDSTROM, P., PAJAROLA, R., AND SILVA, C. T. 2003. Out-of-core algorithms for scientific visualization and computer graphics. IEEE Visualization 2003, Tutorial 4 Course Notes.

CIGNONI, P., MONTANI, C., ROCCHINI, C., AND SCOPIGNO, R. 2003. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics 9*, 525–337.

CIGNONI, P., DE FLORIANI, L., PASCUCCI, V., ROSSIGNAC, J., AND SILVA, C. T. 2003. Multiresolution modeling, visualization, and compression of volumetric data. IEEE Visualization 2003, Tutorial 3 Course Notes.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2003. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3, 505–514.

COHEN-OR, D., AND LEVANONI, Y. 1996. Temporal continuity of levels of detail in delaunay triangulated terrain. In *IEEE Visualization '96*, IEEE.

DACHSBACHER, C., VOGELSGANG, C., AND STAMMINGER, M. 2003. Sequential point trees. In *Proc. SIGGRAPH*, 657–662.

DUCHAINEAU, M., WOLINSKY, M., SIGETI, D., MILLER, M., ALDRICH, C., AND MINEEV-WEINSTEIN, M. 1997. ROAMing terrain: Real-time optimally adapting meshes. In *Proceedings IEEE Visualization '97*, IEEE, 81–88.

EL-SANA, J., AND CHIANG, Y.-J. 2000. External memory view-dependent simplification. *Computer Graphics Forum 19*, 3 (Aug.), 139–150.

EL-SANA, J., AND VARSHNEY, A. 1999. Generalized view-dependent simplification. *Computer Graphics Forum 18*, 3, 83–94.

ERIKSON, C., MANOCHA, D., AND BAXTER, W. 2001. HLODs for faster display of large static and dynamic environments. In *Proc. ACM Symposium on Interactive 3D Graphics*, 111–120.

FISCHER, K., AND GÄRTNER, B. 2003. The smallest enclosing ball of balls: combinatorial structure and algorithms. In *Proceedings of the nineteenth conference on Computational geometry*, ACM Press, 292–301.

GREGORSKI, B., DUCHAINEAU, M., LINDSTROM, P., PASCUCCI, V., AND JOY, K. I. 2002. Interactive view-dependent rendering of large IsoSurfaces. In *Proc. IEEE Visualization*, 475–484.

HOPPE, H. 1997. View-dependent refinement of progressive meshes. In *SIGGRAPH 97 Conference Proceedings*, Addison Wesley, T. Whitted, Ed., Annual Conference Series, ACM SIGGRAPH, 189–198. ISBN 0-89791-896-7.

HOPPE, H. 1998. Smooth view-dependent level-of-detail control and its aplications to terrain rendering. In *IEEE Visualization '98 Conf.*, 35–42.

HOPPE, H. 1999. New quadric metric for simplifying meshes with appearance attributes. In *Proceedings of the 10th Annual IEEE Conference on Visualization (VIS-99)*, ACM Press, New York, pages 59–66.

ISENBURG, M., LINDSTROM, P., GUMHOLD, S., AND J.SNOEYINK. 2003. Large mesh simplification using processing sequences. In *Proc. IEEE Visualization*.

ISENBURG, M. 2001. Triangle strip compression. *Computer Graphics Forum 20*, 2, 91–101.

LEVENBERG, J. 2002. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings IEEE Visualization '02*, IEEE, 259–266.

LINDSTROM, P., AND PASCUCCI, V. 2002. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transaction on Visualization and Computer Graphics 8*, 3, 239–254.

LINDSTROM, P. 2000. Out-of-core simplification of large polygonal models. In *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH 2000), ACM Press*, Addison Wesley, 259–262.

LINDSTROM, P. 2003. Out-of-core construction and visualization of multiresolution surfaces. In *ACM 2003 Symposium on Interactive 3D Graphics*, 93–102,239.

LUEBKE, D., AND ERIKSON, C. 1997. View-dependent simplification of arbitrary polygonal environments. In *ACM Computer Graphics Proc., Annual Conference Series, (SIGGRAPH 97)*, 199–208.

MAUBACH, J. 1995. Local bisection refinement for n-simplicial grids generated by bisection. *SIAM Journal of Scientific Computing 16*, 210–227.

OHLBERGER, M., AND RUMPF, M. 1998. Adaptive projection operators in multiresolution scientific visualization. *IEEE Transactions on Visualization and Computer Graphics 4*, 4, 344–364.

PASCUCCI, V. 2002. Slow growing subdivision (SGS) in any dimension: Towards removing the curse of dimensionality. *Computer Graphics Forum 21*, 3, 451–460.

POMERANZ, A. A. 2000. *ROAM Using Surface Triangle Clusters (RUSTiC)*. Master's thesis, University of California at Davis.

PRINCE, C. 2000. *Progressive Meshes for Large Models of Arbitrary Topology*. Master's thesis, Department of Computer Science and Engineering, University of Washington, Seattle.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. QSplat: A multiresolution point rendering system for large meshes. In *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH 00)*, ACM Press, 343–352.

SAMET, H. 1990. *Applications of Spatial Data Structures*. Addison Wesley, Reading, MA.

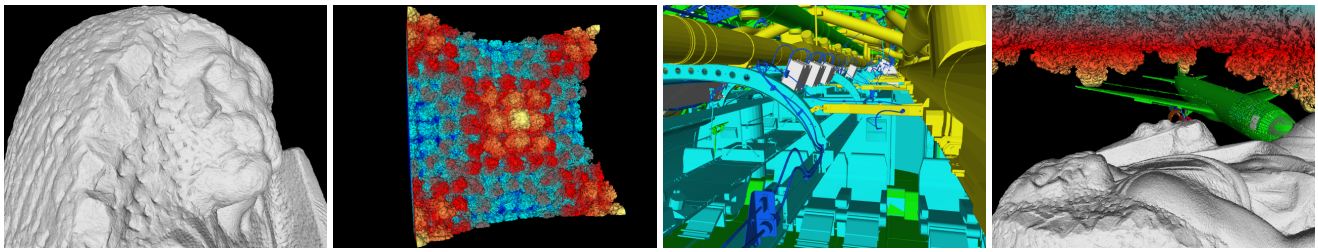
SHAFFER, E., AND GARLAND, M. 2001. Efficient adaptive simplification of massive meshes. In *Proc. IEEE Visualization 2001*, IEEE Press, 127–134.

WU, J., AND KOBBELT, L. 2003. A stream algorithm for the decimation of massive meshes. In *Proc. Graphics Interface*, 185–192.

XIA, J., AND VARSHNEY, A. 1996. Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization '96 Proc.*, R. Yagel and G. Nielson, Eds., 327–334.

Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms

Enrico Gobbetti and Fabio Marton
CRS4 – Visual Computing Group *



(a) St. Matthew (372M triangles approximated with 1.5M voxels and 345K triangles)

(b) Richtmyer-Meshkov isosurface (472M triangles approximated with 4.2M voxels and 137K triangles)

(c) Boeing 777 (350M triangles approximated with 1M voxels and 3.4M triangles)

(d) All three datasets at once (1.2G triangles approximated with 4.0M voxels and 595K triangles)

Figure 1: **View-dependent rendering of extremely complex models.** These multi-gigabyte datasets cover a wide range of model classes. We can render them interactively at 1pixel tolerance on a Xeon 2.4 GHz PC with 1GB RAM and a NVIDIA GeForce 6800GT AGP8X graphics board.

Abstract

We present an efficient approach for end-to-end out-of-core construction and interactive inspection of very large arbitrary surface models. The method tightly integrates visibility culling and out-of-core data management with a level-of-detail framework. At preprocessing time, we generate a coarse volume hierarchy by binary space partitioning the input triangle soup. Leaf nodes partition the original data into chunks of a fixed maximum number of triangles, while inner nodes are discretized into a fixed number of cubical voxels. Each voxel contains a compact direction dependent approximation of the appearance of the associated volumetric subpart of the model when viewed from a distance. The approximation is constructed by a visibility aware algorithm that fits parametric shaders to samples obtained by casting rays against the full resolution dataset. At rendering time, the volumetric structure, maintained off-core, is refined and rendered in front-to-back order, exploiting vertex programs for GPU evaluation of view-dependent voxel representations, hardware occlusion queries for culling occluded subtrees, and asynchronous I/O for detecting and avoiding data access latencies. Since the granularity of the multiresolution structure is coarse, data management, traversal and occlusion culling cost is amortized over many graphics primitives. The efficiency and generality of the approach is demonstrated with the interactive rendering of extremely complex heterogeneous surface models on current commodity graphics platforms.

CR Categories: I.3.3 [Computer Graphics]: Picture and Image Generation—; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—.

Keywords: Out-Of-Core Algorithms, Level of Detail

*CRS4 Visual Computing Group, POLARIS Edificio 1, 09010 Pula, Italy www: `http://www.crs4.it/vic/` e-mail: `{gobbetti|marton}@crs4.it`

1 Introduction

Many important application domains, including 3D scanning, computer aided design, and numerical simulation, require the interactive inspection of huge geometric models. Despite the rapid improvement in hardware performance, rendering today's multi-gigabyte datasets at interactive rates largely overloads the performance and memory capacity of state-of-the-art hardware platforms. To overcome this limitation, researchers have proposed a wide variety of output-sensitive rendering algorithms, i.e., rendering techniques whose runtime and memory footprint is proportional to the number of image pixels, not to the total model complexity. Very few techniques exist, however, that tightly integrate visibility culling and out-of-core rendering with level-of-detail management, limiting the applicability of the different approaches to only specific classes of objects. The lack of one of these techniques, or their independent application within a rendering engine, poses important problems when dealing with datasets that combine complicated geometry and appearance with a large depth complexity. Consider, for instance, the most advanced methods for rendering large scale models with fine geometric details [Yoon et al. 2004; Guthe et al. 2004; Cignoni et al. 2004], which are based on multiresolution point- or vertex- hierarchies constructed off-line through a geometric simplification process. Typically, these methods, that repeatedly merge nearby surface points or mesh vertices based on error minimization considerations, perform best for highly tessellated surfaces that are otherwise relatively smooth and topologically simple, since it becomes difficult, in other cases, to derive good "average" merged properties. Moreover, and most importantly, the off-line simplification process that generates the multiresolution hierarchy from which view-dependent levels of detail are extracted at rendering time is essentially unaware of visibility. When approximating very complex models, however, resolving the ordering and mutual occlusion of even very close-by surfaces, potentially with different shading properties, is of primary importance (see figure 2).

Main contributions. We present a new efficient approach for end-to-end out-of-core construction and view-dependent rendering of very large arbitrary surface models on commodity graphics platforms. The method tightly integrates visibility culling and out-of-core data management with level-of-detail construction and rendering. The underlying idea is to depart from current point- or triangle-based multiresolution surface models and adopt a volumetric approach based on more complex rendering primitives. At preprocessing time, we generate a coarse volume hierarchy by a binary

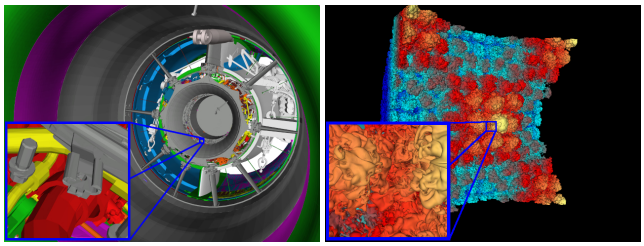


Figure 2: **Boeing 777 engine details (left) and isosurface details (right).** These kinds of object, composed of many loosely connected interweaving detailed parts of complex topological structure, are very hard to simplify effectively using off-line geometric simplification methods that do not take into account visibility.

space partitioning process of the input dataset. Leaf nodes simply split the original data into chunks of a fixed maximum number of triangles, while inner nodes are discretized into a fixed number of cubical voxels. Each voxel contains a compact direction dependent approximation of the appearance of the associated volumetric subpart of the model when viewed from a distance. The approximation is constructed by a visibility aware algorithm that fits parametric shaders to samples obtained by casting rays against the full resolution dataset. At rendering time, the volumetric structure, maintained off-core, is refined and rendered in front-to-back order, exploiting vertex programs for GPU evaluation of view-dependent voxel representations, hardware occlusion queries for culling occluded subtrees, and asynchronous I/O for avoiding out-of-core data access latencies. Since the granularity of the multiresolution structure is coarse, data management, traversal and visibility culling costs are amortized over many graphics primitives, and disk/CPU/GPU communication can be optimized to fully exploit the complex memory hierarchy of modern graphics PCs.

Advantages. The resulting technique, dubbed *Far Voxels*, has the following properties: it is applicable to a wide range of model classes, that include very detailed colored objects composed of many loosely connected interweaving detailed parts of complex topological structure; it is fully adaptive and is able to retain all the original topological and geometrical detail even for massive datasets; it is strongly GPU bound, since its coarse grained structure successfully exploits the batched rendering model of current commodity graphics platforms; multiresolution representations can be constructed in parallel with a out of core algorithm.

Limitations. As for all current large scale model rendering approaches, our method has also limitations: it has been designed for static models, editing is not supported; the sampling and fitting process is general and supports many model kinds, but at the expense of high preprocessing costs or aliasing problems if not enough rays are shot; the particular rendering primitives employed in this paper are phenomenological and appropriate mostly for diffuse materials – generalizing them is an important avenue of future work; the splatting method used for rendering does not correctly handle transparency; our current renderer implementation strives to maintain interactivity rather than ensuring strict guarantees on image quality; we have currently not implemented out-of-core compression and speculative prefetching – these are orthogonal to our method, but important to reduce perceived refinement latency.

Despite these limitations, the current method and prototype system is of immediate practical use and provides unprecedented performance in rendering very large complex models. As highlighted in section 2, while certain other methods share some of *Far Voxel*'s advantages, they typically do not meet its capability in all of the areas. A general overview of our approach is presented in section 3, while section 4 introduces a general out-of-core technique for constructing the multiresolution model, and section 5 describes view-dependent refinement and rendering algorithms. The efficiency and generality of the approach has been successfully evaluated with extremely complex CAD, isosurface, and scanned models (section 6).

2 Related Work

Rapidly rendering adaptive representations of large models is a very active research area. In the following, we will discuss the approaches that are most closely related with our work. Readers may refer to recent surveys (e.g., [Chiang et al. 2003; Cohen-Or et al. 2003]) for further details.

Out-of-core view-dependent simplification. The vast majority of view-dependent simplification methods for general meshes are based on constructing a graph of possible refinement/coarsening operations at the point, vertex, or triangle level. Up until recently, most techniques required an incore preprocessing step, even though rendering was performed out-of-core. QSplat [Rusinkiewicz and Levoy 2000], based on a out-of-core hierarchy of bounding spheres traversed at run-time to generate point splats, was the first fully out-of-core point-based method, while Lindstrom's [2003] scheme based on vertex clustering on a rectilinear octree was the first fully out-of-core mesh-based technique. As for all classic adaptive rendering techniques, these methods spends a great deal of rendering time to compute the view-dependent representation and do not scale well to gigantic meshes. A number of authors have thus proposed various ways to push the rendering performance limits in particular situations. The randomized z-buffer [Wand et al. 2001] uses a hierarchical traversal of a structure where the leaf nodes contain arrays of random point samples. They focus on scenes with many instances of simple objects. Stamminger and Drettakis [2001] dynamically adjusts the point sampling rate for rendering complex procedural geometry at high frame rates, but they require a parameterization of the model, while we focus on arbitrarily complex environments. A number of authors have recently proposed techniques reducing the per-primitive workload by using coarse multiresolution structures that compose at run-time pre-assembled optimized primitive groups [Erikson et al. 2001; Levenberg 2002; Cignoni et al. 2003; Yoon et al. 2004; Guthe et al. 2004; Cignoni et al. 2004; Gobbetti and Marton 2004]. We also follow this approach. All the mentioned methods employ, however, error metrics defined on the boundary surface of the objects, and are thus optimized for small numbers of high-complexity, densely-tessellated objects, rather than for many objects forming a topologically rich assembly. A number of authors have proposed topology reducing techniques using intermediate volumetric representations (e.g, [Andujar et al. 2002]). These methods, however, are view independent, and have been applied mostly to off-line mesh simplification.

Visibility culling. Visibility culling approaches are broadly classified into from-point and from-region visibility algorithms [Cohen-Or et al. 2003]. From-region algorithms compute a potentially visible set (PVS) for cells of a fixed subdivision of the scene and are applied offline in a preprocessing phase. From-point algorithms are applied online for each particular viewpoint. For general environments, accurate PVSs are hard to compute [Bittner et al. 2004]. Image-based occlusion representations are thus widely used, and the most recent algorithms exploit graphics hardware to perform online visibility culling [Bittner et al. 2004; Yoon et al. 2004; Zhang et al. 1997; Klosowski and Silva 2001]. Our method combines an off-line phase, integrated with level-of-detail generation, that removes voxels practically always occluded, with an on-line phase that exploits hardware occlusion queries to perform coarse hierarchical visibility culling. Spatiotemporal coherence is exploited as in [Bittner et al. 2004] to optimize the scheduling of queries. Few approaches exist that integrate LODs with occlusion culling both in the construction and rendering phases. Notable exceptions are hardly visible sets [Andujar et al. 2000], and visibility guided simplification [Zhang and Turk 2002], which, however, are non-conservative techniques that favor model simplification in areas that are likely to be occluded.

Hybrid rendering approaches. Many hybrid algorithms have been proposed that combine multiple techniques to render massive models. A representative example is the MMR/Gigawalk system [Aliaga et al. 1999; Govindaraju et al. 2003], which combines static LODs, HLODs [Erikson et al. 2001] and image based impostors with occlusion culling and out-of-core computation and is applicable to large CAD models that can be naturally partitioned into rectangular cells. El-Sana et al. [2001] and the iWalk system [Corrêa et al. 2003] combined view-dependent rendering with approximate occlusion culling for highly occluded scenes. Chen and Nguyen [Chen and Nguyen 2001], Cohen et al. [2001] and Guthe et al. [2004] presented methods for combining multiresolution polygon and point rendering applicable densely sampled surfaces. These systems are optimized for particular model classes, and are hardly applicable to extremely detailed models with variable degrees of occlusion. A few systems based on volumetric representations have been proposed. Wimmer et al [2001] followed an approach similar to ours for data resampling, but focused on encoding the appearance of pre-shaded models as seen from a view cell. Livnat and Tricoche [2004] presented an isosurface extraction technique that, similarly to our work, approximates internal nodes of a LOD BSP tree with a point when a node projects to less than a pixel. Decaudin and Neyret [2004] also used a volumetric approach, but their system is specialized to procedural forest scenes obtained by repeated instancing of volumetric texture tiles.

Interactive ray-tracing approaches. An alternative to rasterization is to use ray tracing techniques. Thanks to spatial indexing, ray queries can be answered in logarithmic time, and only those parts of the scene that are visible need to be accessed. A number of authors have thus designed raytracing systems for massive model visualization. Pharr et al. [1997] proposed a caching scheme that reorders the rays in a way that minimizes disk I/O. DeMarle et al. [2004] presented parallel techniques for efficiently distributing models and tasks on a large distributed cluster. Wald et al. [2004] presented a heavily optimized system that exploits SIMD instructions for rendering large polygonal models on a dual Opteron PC with 6GB RAM. A small number of in-core volumetric proxies is exploited for representing not-yet-loaded geometry in order to hide disk I/O latencies. While the results obtained with this method are impressive, achieving 3-7 fps at video resolution for the Boeing 777 dataset, the lack of multiresolution data forces the algorithm to access large parts of the dataset for producing a single frame (e.g., over 2GB for a 640x480 1 ray/pixel overview of the Boeing 777 dataset). Moreover, the method does not employ GPU acceleration. By fully exploiting current GPUs, we demonstrate that it is possible to achieve nearly one order of magnitude higher frame rates on lower end 32 bit machines. Given current CPU/GPU performance trends, this gap is likely doomed to widen.

3 Multiresolution model overview

Our approach, that exploits the programmability and batched rendering performance of current GPUs, is based on the idea of moving the grain of the multiresolution surface model up from points or triangles to small volumetric clusters, which represent spatially localized dataset regions using groups of (procedural) graphics primitives. The clusters provide the capability of performing coarse-grained view-dependent refinement of the model and are also used for on-line visibility culling and out-of-core rendering.

Figure 3 provides an overview of the approach. To generate the clusters, the model is hierarchically partitioned with a axis-aligned BSP tree. Leaf nodes partition full resolution data into fixed triangle count chunks, while inner nodes are discretized into a fixed number of cubical voxels arranged in a regular grid.

Finding a suitable voxel representation is challenging, since a voxel region can contain arbitrarily complex geometry. To simplify

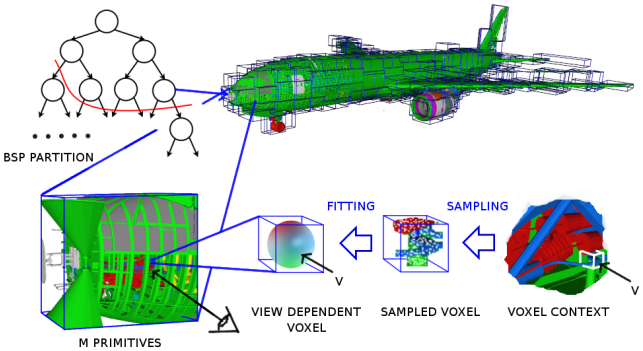


Figure 3: **Multiresolution structure overview.** The model is hierarchically partitioned with a axis-aligned BSP tree. Leaf nodes are rendered using the original triangles, while inner nodes are approximated using view-dependent voxels.

the problem, we assume that each inner node is always viewed from the outside, and at a distance sufficient to project each voxel to a very small screen area (say, below one image pixel). This constraint can be met with a suitable view-dependent refinement method, that refines the structure until a leaf is encountered or the image of each voxel is small enough (see section 5). Under this condition, a voxel always subtends a very small viewing angle, and a purely direction dependent representation of shading information is thus sufficient to produce accurate visual approximations of its projection. This approximation is similar in spirit to the lumispheres [Wood et al. 2000] employed for surface light field rendering, except that we do not associate shaded illumination samples to every ray originating from a surface, but rather prefiltered voxel shading information.

To construct a view-dependent voxel representation, we employ a visibility aware sampling and reconstruction technique detailed in section 4. We first acquire a set of shading information samples by ray casting the original model from a large number of appropriately chosen viewing positions. Each sample associates a reflectance and a normal to a particular voxel observation direction. We then compress these samples to an analytical form that can be compactly encoded and rapidly evaluated at run-time on the GPU to compute voxel shading given a view direction and light parameters.

4 Construction

The off-line component of our method constructs a space partitioned multiresolution structure starting from the full resolution model, that we assume, without loss of generality, represented as a *triangle soup*, i.e., a flat list of triangles with direct vertex information.

4.1 Mesh partitioning and deep BSP construction

The first preprocessing phase consists in spatially partitioning the scene according to an axis aligned BSP tree, whose root coincides with the mesh bounding box and whose leaves contain less than a small predefined number of mesh triangles. The BSP tree constructed in this phase will be exploited to accelerate the ray casting process and serves as a basis for constructing the multiresolution volumetric structure. The BSP is constructed out-of-core according to the surface-area heuristic [MacDonald and Booth 1990], which produces subdivisions that closely encompass the model, and stored in memory coherent order [Havran 1999]. Each BSP leaf points to the list of triangles that pass trough it, in a format optimized for ray-triangle intersection [Arenberg 1988]. The final structure, stored on disk in binary form, can be used for out-of-core sampling of the scene, by mapping it to the user address space and letting the OS manage demand loading on a per-page basis.

4.2 Level-of-detail hierarchy construction

Once the deep BSP tree is available, we exploit it for constructing the level-of-detail structure for the scene. We first generate the final

multiresolution structure layout by constructing a coarse hierarchical structure on top of the deep BSP hierarchy. In this layout, empty nodes are removed, while leaf nodes are associated with subtrees of the deep BSP that approximately contain a number of triangles corresponding to the specified cluster size. We then traverse the coarse hierarchy and generate for each node the final representation, which is immediately stored on disk. Nodes are visited level by level, and, at each level, in order of geometric proximity. To obtain that, the nodes at a given level are sorted by increasing Morton code of their center point. This traversal order optimizes memory coherence at construction time, and even more importantly, at rendering time. To further increase memory coherence, the hierarchical data structure is split on disk into two files: an index tree and a data repository. The index tree has a small footprint, since it contains, for each node, just the data required for traversal (bounding box, voxel size, and index of the two children), and refers to the associated rendering data through a 64 bit offset into the repository and a 32 bit size. The repository contains all the data required for rendering the node.

Generating rendering data is the main construction task. In our approach, each leaf node is encoded as a generalized triangle strip covering the geometry contained in it, thus retaining all the original topological and geometrical detail. To generate the strip, triangles are extracted from the deep BSP subtree associated to the generated node and clipped to the node's bounding box. Connectivity is reconstructed from independent triangles by vertex position hashing. Non-leaf nodes contain instead a volumetric simplification, generated from a discretization of the bounding box into a fixed number of (approximately) cubical voxels arranged in a regular grid.

Sampling process. To construct a view-dependent voxel representation, shading information samples are acquired by casting rays against the deep BSP of the original model from a large number of possible viewing position. To generate the samples, we exploit the fact that a given node will be always viewed from a distance sufficient to have each voxel subtend a very small viewing angle. We provide the preprocessor with a worst case value θ_{max} for this angle, which will correspond to the coarsest possible accuracy for view-dependent rendering (see section 5). Given the voxel radius r_{vox}, we derive the minimum viewing distance $d_{min} = r_{vox}/\tan(\theta_{max}/2)$, and cast against the deep BSP a large number of random rays originating on the surface S at distance d_{min} from the node's bounding volume B and randomly directed toward it (see figure 4). For each hit position inside B, we compute the voxel index, and store in an associated list a record containing the intersected surface reflectance, its normal, and the ray direction. Hits outside B are simply discarded. At the end of the process, each potentially visible voxel will be associated to a list of samples covering all the unoccluded directions. An important property of the sampling strategy is that, given a sufficiently large number of rays, visibility problems are conservatively solved for arbitrary view positions outside S by using information from the entire environment enclosed by S. In particular, voxels that do not contain surface boundaries, as well as voxels that are fully occluded, will be empty. At the same time, for each voxel, only visible surfaces are sampled, and only from those directions where they can be seen. As demonstrated in section 6, this visibility aware sampling strategy is extremely effective for complex models, as environmental occlusion leads to eliminate a large portion of the voxels (over 40% for the Boeing 777 dataset) and minimizes artifacts due to leaking of occluded objects colors through nearby occluding surfaces.

This ray casting based sampling strategy is very general and supports many model kinds, but at the expense of high preprocessing costs or aliasing problems if not enough rays are shot. For simplicity of implementation, we decided to cast a fixed number of rays per volume, even though classic raytracing sampling methods that cast more rays in high variance regions could definitely be used to improve accuracy. A promising direction to reduce processing times while keeping aliasing problems under control would be to construct the structure in a bottom-up fashion, and cast rays against the already constructed multiresolution structure instead of sampling the original model at each node.

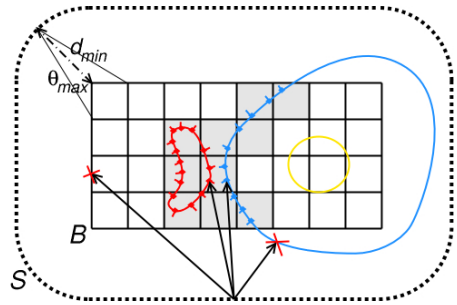


Figure 4: **Sampling process.** Shading information samples are acquired by ray casting the model from a large number of possible viewing positions at distance d_{min} from the volume. Environmental occlusion is taken into account to remove always occluded voxels arnd to restrict the sampling to potentially visible surfaces. In the image, the blue object hides the yelloow one, and only gray voxels are considered non-empty.

Parametric shader compression. The sample list for each voxel provides all the information for computing view-dependent shading. To generate a more compact and efficient representation, we compress these samples by fitting them to simple parameterized shader models, and choosing the shader that provides the best approximation. Each shader consists in a function that returns a color attenuation given its internal parameters, a view direction $\mathbf{v}$ and a light direction $\mathbf{l}$, i.e., $Shader_i(\mathbf{v},\mathbf{l}) = BRDF_i(\mathbf{v},\mathbf{l})\max(\mathbf{n}(\mathbf{v})\cdot\mathbf{l}),0)$, where $\mathbf{n}(\mathbf{v})$ is the surface normal seen from $\mathbf{v}$. Instead of deriving a general purpose shader, we assume that a small number of shader classes can be used to model common situations (see figure 5). In our current prototype, we have implemented the following classes:

K1a A flat shader, parameterized by a plane normal, a front material, and a back material, that implements the standard two-sided Lambert reflection model. The normal is found by averaging all sampled normals, while the colors are found as the front and back average.

K1b The same as above, with the normal computed by principal component analysis of the sampled hit positions.

K2 A smooth shader, parameterized by 6 reflectance and 6 normal control points associated to the main viewing directions $(\pm x, \pm y, \pm z)$, found by averaging sampled values according to the associated ray direction. Lambert shading parameters are found from v by summing the values at the three nearest reference directions weighted by the respective direction cosine.

The shader selected for a particular voxel is found by constructing an instance of each shader class k using a random subset of the samples. We then use the remaining samples to measure the difference in shading for a number of random light directions $\mathbf{l}_j$: $\varepsilon^{(k)} = \sum_i \sum_j \left(BRDF_i^{(sampled)}(\mathbf{v}_i,\mathbf{l}_j)\max(\mathbf{n}_i\cdot\mathbf{l}_j,0) - Shader^{(k)}(\mathbf{v}_i,\mathbf{l}_j)\right)^2$. The shader instance with minimum error is then selected and its representation is stored.

Even though the shader sampling and fitting method is very general, the particular shaders described here are phenomenological and appropriate mostly for diffuse materials. They have been chosen mostly because of ease of implementation and efficiency of computation. Generalizing them is an important avenue of future work.

Output data encoding. Since our current renderer is based on voxel splatting and vertex shaders (see 5), we only store non-empty voxels, and encode each shader parameter in a specific vertex array,

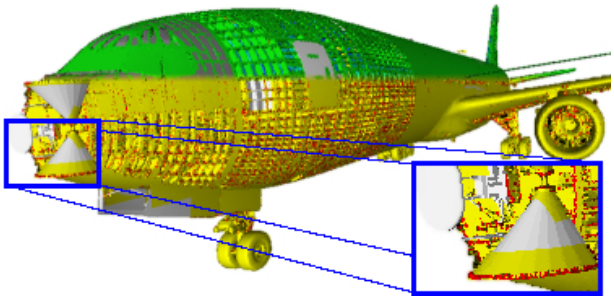


Figure 5: **Primitive distribution.** The top part of the image shows the model as presented to the viewer. The bottom part illustrates the primitive class distribution using a color code: red for K2 shaders, which tend to accumulate on complex voxels, yellow for K1 shaders, which tend to accumulate on almost planar surfaces, and white for triangles, used for full resolution leafs.

much in the same way we encode triangle strip data for leaf nodes (see figure 6). In our current implementation, we do not compress data to a compact external format, but rather directly encode each value in OpenGL format, using 4 bytes for colors, 6 bytes for normals, and 6 bytes for relative positions inside a box. Exploiting compression techniques for reducing disk usage and I/O needs is a major avenue for future work.

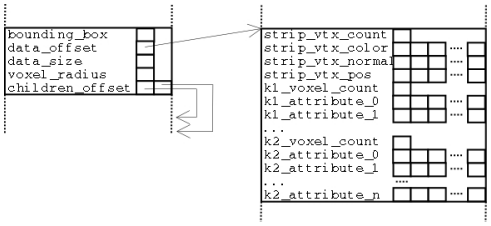


Figure 6: **Output data encoding.** All data required for rendering is encoded in vertex array format.

Parallel out-of-core construction. The hierarchy construction phase dominates, by far, the overall processing cost, mainly because of the scene sampling process. The whole process is however inherently massively parallel. For this paper, we have adopted the simple solution of separating the input models into chunks of 20-30M triangles each with a very coarse BSP partitioning and to distribute the chunks to N machines that execute in parallel, and out-of-core, the rest of the preprocessing. As a result, we obtain a forest instead of a single tree.

5 View-dependent rendering

At rendering time, the volumetric structures, maintained off-core, are refined and rendered in front-to-back order, exploiting vertex programs for GPU evaluation of view-dependent voxel representations, hardware occlusion queries for culling occluded subtrees, and asynchronous I/O requests for avoiding out-of-core data access latency.

Refinement algorithm. The user selected pixel threshold is the value that drives the refinement: this value represents the maximum required projected voxel size on the screen. The algorithm takes as input a forest of multiresolution hierarchies and performs a breadth-first front-to-back traversal, making use of the following data structures: a *node priority queue*, for sorting visited nodes in front-to-back order; an *occlusion query queue*, for storing pending occlusion queries; a *GPU cache*, based on OpenGL's *Vertex Buffer Objects* extension, for storing the most recently *rendered* nodes; a *RAM cache*, for storing the most recently *visited* nodes; a *fetch request priority queue*, for storing asynchronous I/O requests for nodes not yet available.

The traversal is initiated by inserting the roots of each tree into the node priority queue, and, if this is the first frame, into the RAM cache. We then iteratively remove and process the highest priority node from the node priority queue or from the occlusion query queue until both queues are empty.

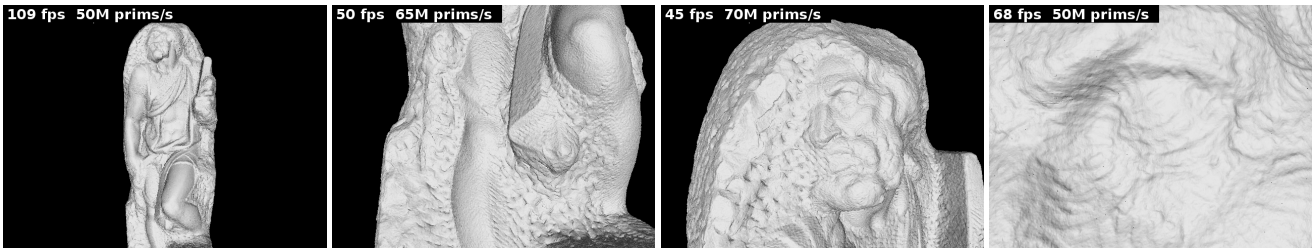
Each time a node is extracted from the node priority queue, we mark it by default as invisible for the current frame. We then check whether its bounding box falls totally outside the view volume. If so, we simply stop with this node, culling away its entire subtree. If instead the node is at least partially within the view frustum, we test whether to stop refinement, either because the node is a leaf, or its projected voxel size falls below the screen space threshold, or its children are not yet present in the RAM cache. In that case, the node is rendered, while issuing a hardware occlusion query and storing it in the occlusion query queue. If data is missing, fetch requests for missing children are pushed in the fetch request priority queue. When continuing refinement, we test whether the node was marked visible at the previous frame. If so, we avoid testing for occlusion and immediately push its children in the node priority queue. For previously invisible nodes, on the other hand, we issue an occlusion query for their bounding box, which is then stored in the query queue.

The nodes queried for occlusion are processed only as soon as the result for their occlusion query is available or there are no other nodes to traverse. Each time an item is extracted from the query queue, we check the occlusion query result. If the number of visible pixels returned by the query is zero, we simply stop with this node, culling away its entire subtree because of occlusion. If instead some pixels were visible, we mark the node and its ancestors as visible. If the node was not already rendered, it is because it is an inaccurate non-terminal node with RAM cached children. We thus continue refinement by pushing its children in the node priority queue.

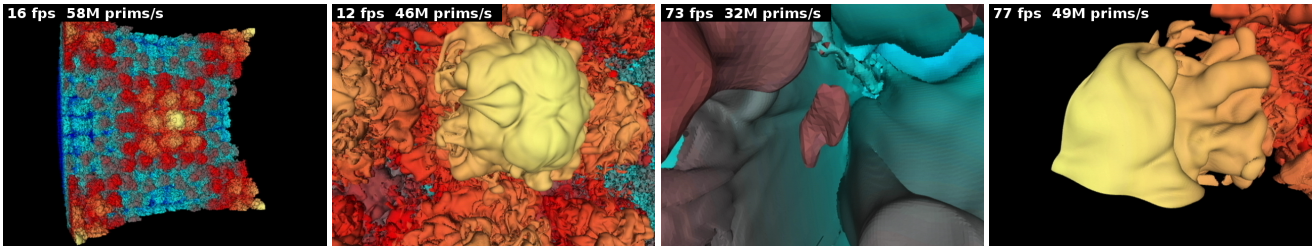
Since the structure is coarse grained and the refiner never stalls because of I/O requests the method is GPU bound.

Node rendering. Our current implementation renders view-dependent voxels using a splatting method that draws an antialiased OpenGL point primitive per voxel. At renderer initialization, vertex shader programs specific to each primitive class are compiled and loaded on board. At node rendering time, we iterate on all possible primitive representations. If the number of voxels/vertices for the given primitive class is non-null, we bind the associated program, load the vertex attribute data into the appropriate program parameters, and issue a `glDrawArrays` to draw all voxels at once. Triangle strip rendering code for leaf nodes follows the same pattern. To minimize bus traffic, each time a node is rendered, we reuse the GPU cached version if present, otherwise we render it and cache its representation in place of the oldest one. In the current implementation, the primitives that compose a particular node are rendered in an arbitrary order, and without taking into account their opacity. This provides a rendering quality similar to that of one-pass point splatting methods, which is sufficient for our target applications, but limits our ability to correctly treat high resolution textures and transparency. We are currently exploring ways to use a texture-based volume rendering approach to solve this problem.

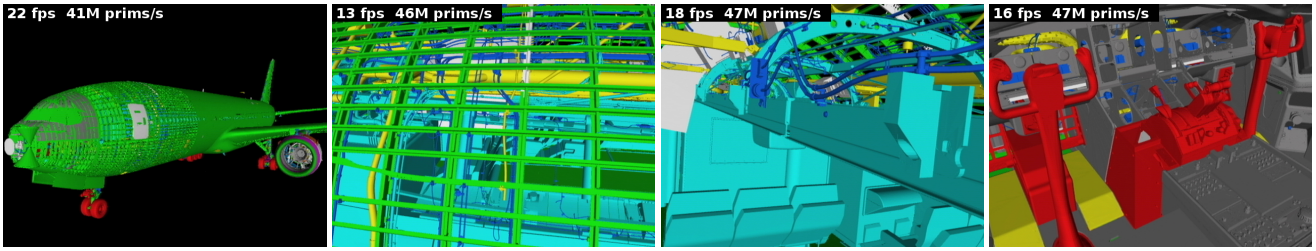
Asynchronous I/O. Similarly to Streaming QSplat [Rusinkiewicz and Levoy 2001], the fetch request queue is traversed in order of priority at the end of the frame, issuing only as many requests as those allowed by the estimated I/O bandwidth, and ignoring the remaining ones. In our current implementation, the priority of a fetch request is given by the node's parent projected voxel size. Access to the data repository is made through a data access layer, that hides from the renderer whether data is local or remote. This layer internally uses memory mapping primitives for local data, and a TCP/IP protocol for remote data. It makes it possible to asynchronously move in-core a node by fetching it from the repository, to test whether a node is immediately available, and to move available nodes to the RAM cache.



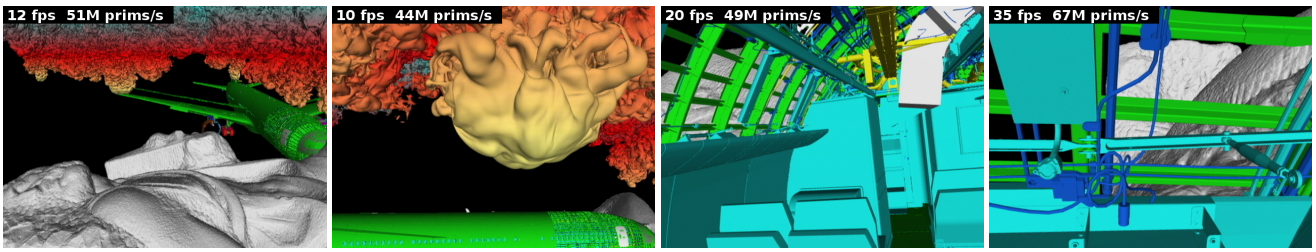
(a) St. Matthew 0.25mm (372M triangles)



(b) Richtmyer-Meshkov Isosurface (472M triangles)



(c) Boeing 777 (350M triangles)



(d) All models at once (1.2G triangles)

Figure 7: **Inspection sequences: selected frames.** All images were recorded live on a Xeon 2.4 GHz PC with 1GB RAM and a NVIDIA GeForce 6800GT AGP8X graphics board using a 1 pixel tolerance.

6 Results

An experimental software library and a rendering application supporting the technique have been implemented on Linux using C++ with OpenGL. We have extensively tested our system with a number of large surface models. The quantitative and qualitative results discussed here are restricted to the three models of figure 7, that possibly represent the most complex benchmark test cases in their respective domains: the St. Matthew 0.25mm dataset (372M triangles) is a very dense high resolution laser scanning model; the Richtmyer-Meshkov Isosurface dataset (472M triangles) is a very convoluted mesh with holes, a huge depth complexity, and a high genus generated from a very high resolution 3D simulation of Richtmyer-Meshkov instability and turbulence mixing; the Boeing 777 dataset (350M triangles) is an exceptionally complex CAD model, composed of many loosely connected interweaving detailed parts of complex topological structure represented as meshes of colored triangles with widely varying aspect ratios.

Preprocessing. Table 1 lists numerical results for our out-of-core preprocessing method for all the test datasets. The tests were executed on a moderately loaded network of 16 PCs running Linux 2.4. Each PC has two CPU Athlon 2200+ CPUs, 1GB DDR memory, a 70GB ATA 133 hard disk, and a Ethernet 100 Mb/s network connection. We constructed all multiresolution structures with a prescribed maximum leaf size of 8K triangles/node, a prescribed non-leaf discretization size of 16K voxels/node, 256K rays/node for sampling, and a maximum voxel viewing angle $\theta_{max} = 0.5$ degrees.

Models	Input file		Deep BSP		Far Voxels			
	M Tri	Size GB	Time sec	Size GB	Time sec	Size GB	Avg.Tri/Leaf	Avg.Vox/Node
St. Matthew	372	14.5	4706	18.6	14592	10.6	4876	1953
Boeing 777	350	13.7	8201	14.9	16461	14.9	5328	2401
Isosurface	472	18.4	5648	26.1	23751	16.1	5822	3425

Table 1: **Numerical results for out-of-core construction.** Tests performed on a network of 16 PCs.

Overall processing times range from about 1K input triangles/s for 1 CPU to 20K input triangles/s for 16 CPU. By contrast, processing times for competing multiresolution approaches based on geometric simplification range from 3K triangles/s [Yoon et al. 2004; Cignoni et al. 2004] for 1 CPU to 30K triangles/s on 16 CPUs [Cignoni et al. 2004]. The Far Voxels method is thus slower, even though it seems to scale better with the number of CPUs.

Our implementation requires on average 70MB per million

vertices. This is comparable to QVDR (88MB) [Yoon et al. 2004], but sensibly higher than the TetraPuzzles representation (32MB) [Cignoni et al. 2004]. Out-of-core model compression is a main avenue of future work.

To test the effectiveness of the strategy employed to cull occluded voxels at preprocessing time, we have also reconstructed the models without taking into account environment occlusion, i.e., by sampling each node separately without considering θ_{min}. The strategy proved very successful, since it removes over 25% of voxels for the isosurface dataset and 43% for the Boeing 777. The scanning model is essentially unaffected due to the low depth complexity.

Adaptive rendering. We evaluated the rendering performance of the technique on a number of inspection sequences on all test datasets, using a Linux PC with a Intel Xeon 2.4 GHz, 1GB RAM, two 70 GB ULTRA SCSI 320 hard drives, AGP 8x and NVIDIA GeForce 6800 GT graphics. The qualitative performance of our adaptive renderer is illustrated in an accompanying video.

The sessions were designed to be representative of typical inspection tasks and to heavily stress the system, and include rotations, rapid changes from overall views to extreme close-ups, and forced visibility discontinuities. To further illustrate the scalability of the method, we have included a test case showing the inspection of a scene containing all three models, for a total of over 1.2GB triangles and 41GB of out-of-core data distributed among the two PC disks. Table 2 lists numerical results for all the sessions.

Input			Window size	Pixel tolerance			Frames/s		Mprim/s	Max Resident Set Size MB
Models	M tri	Size GB		Target	Avg	Max	Min	Avg	Avg	
St. Matthew	372	10.6	640x480	1	0.8	2.4	9	45	51	102
Boeing 777	350	14.9	640x480	1	0.9	8.1	8	44	42	151
Isosurface	472	16.1	640x480	1	1.0	6.6	7	34	41	229
All	1194	41.6	640x480	1	0.9	12.2	6	20	42	172
All	1194	41.6	2x1024x768	1	1.1	18.0	3	20	40	218

Table 2: **Numerical results for out-of-core rendering.** Tests performed on a Linux PC with a Intel Xeon 2.4 GHz, 1GB RAM, two 70 GB ULTRA SCSI 320 hard drives, AGP 8x and NVIDIA GeForce 6800 GT graphics.

As illustrated in this table and in the accompanying video, our system is able to maintain interactivity in all test cases, while producing detailed images. As explained in section 5, our current renderer favors interactivity rather then ensuring strict bounds on accuracy, since we stop refinement and issue anynchronous I/O requests when data has to be fetched from disk. However, even though the maximum projected size has peaks caused by the rendering of coarser nodes when data is not immediately available, the average projected voxel size is, as expected, close (or below) the target tolerance (1 pixel), since due to temporal coherence only few nodes per frame cause RAM/GPU cache misses. The highest maximum values are for the larger models, which have longer load times due to increased disk footprint. The popping artifacts caused by rendering inaccurate nodes could be reduced by incorporating compression, speculative prefetching, or using disk RAIDs to serve the data.

During the entire inspection sequences, the resident set size of the application never exceeded 1.5% of the out-of-core data size, demonstrating the effectiveness of out-of-core data management.

We can sustain an average rendering rate of around 45M primitives/s independently of the model, counting as one primitive a single voxel or triangle. Throughput comparisons with classic tessellation techniques are difficult, since we employ heavier primitives than pure Gouraud shaded triangle strips. In our current unoptimized implementation, shading requires 19–28 vertex program instructions depending on primitive type. For simple models, such as the laser scanning test-case, it is likely that methods such as TetraPuzzles, which was able to sustain 70M tri/s on a GeForce Ultra FX 5800 GPU thanks to cache-coherent triangle strips [Cignoni et al. 2004], would provide better performance. Multiresolution mesh techniques would however be hardly applicable to the other examples demonstrated here. Even in the current implementation, the primitive rate is high enough to render over 4.5M primitives/frame at interactive rates. It is thus possible to use very small pixel thresholds, virtually eliminating popping artifacts without the need to resort to costly geomorphing techniques. The raw performance of the system is particularly useful for large scale display situations, where interactive raytracing solutions have problems meeting real-time constraints because of the large number of pixels to be covered. Figure 8 shows the largest scene examined on a large scale stereoscopic display assembled from off-the-shelf components, i.e., two 1024x768 DLP projectors connected to two outputs of the graphics card, polarizing filters with matching glasses, and a backprojection screen that preserves polarization. In this setting, a single PC is able to render two 1024x768 images per frame at an average of 20 Hz with a 1 pixel rendering tolerance, with a worst case of 3 Hz. Lowering the tolerance to 2 pixels increases the minimum frame rate to 9 Hz with little visual impact.

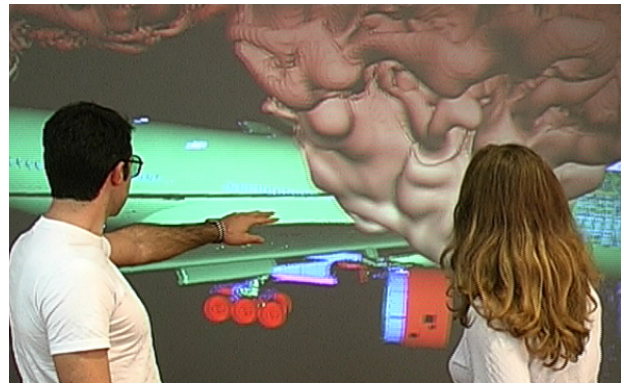


Figure 8: **Large scale stereoscopic display.** The largest test case (1.2G triangles) interactively inspected on a large scale stereoscopic display driven by single PC, which renders two 1024x768 images per frame with a 1 pixel tolerance.

Network streaming. Some network tests have been performed on all test models, on a local area network at 100Mb/s using the TCP/IP protocol to access the data. As illustrated in the video, rendering rate remains the same as that of the local file version, but updates asynchronously arrive with increased latency. The effect is illustrated in figure 9, which shows the progressive refinement of the Boeing 777 dataset on a machine connected to a moderately loaded Linux box serving the models. Even though our current uncompressed model encoding is far from being optimal for the task, the application remains usable even for very large models on standard network connections.

7 Conclusions

We have presented an efficient technique for end-to-end out-of-core construction and view-dependent rendering of very large heterogeneous surface models on commodity graphics platforms. The main benefit of the method lies in its performance and applicability to a wide variety of model classes. As a result, we obtain an unprecedented spatiotemporal quality in the interactive inspection of massive models that exhibit complicated geometry and topology, heterogeneous material attributes, as well as large variations in depth complexity.

Besides improving the proof-of-concept implementation, we plan to extend the presented approach in a number of ways. In particular, we are currently working on the following aspects: compression of the output data representation to reduce disk usage and I/O latency; exploration of alternate view-dependent voxel representations; implementation of multi-resolution sampling methods; exploration of higher quality texture-based volume rendering methods in place of the current point splatter; prefetching and prediction of occlusion/visibility events to reduce artifacts due to the display of coarser than needed newly visible objects.

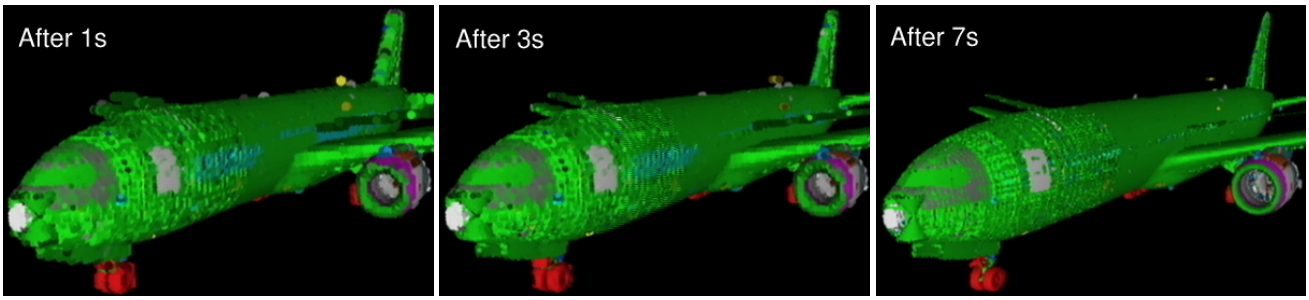


Figure 9: **Streaming.** Progressive refinement of the Boeing 777 dataset (350M triangles) on a 100Mb/s connection.

Acknowledgments. The source 3D datasets are provided by and used with permission of the Boeing Company, the Digital Michelangelo project, and the Lawrence Livermore National Laborarories. The authors would also like to thank Fabio Bettio (CRS4), Paolo Cignoni (ISTI-CNR), Francesca Frexia (CRS4), David Kasik (Boeing), Gianni Pintore (CRS4), and Alan Scheinine (CRS4).

References

ALIAGA, D., COHEN, J., WILSON, A., BAKER, E., ZHANG, H., ERIKSON, C., HOFF, K., HUDSON, T., STÜRZLINGER, W., BASTOS, R., WHITTON, M., BROOKS, F., AND MANOCHA, D. 1999. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *1999 Symposium on Interactive 3D Graphics*, 199–206.

ANDUJAR, C., SAONA, C., NAVAZO, I., AND BRUNET, P. 2000. Integrating occlusion culling and levels of detail through hardly-visible sets. *Computer Graphics Forum 19*, 3, 499–506.

ANDUJAR, C., BRUNET, P., AND AYALA, D. 2002. Topology-reducing surface simplification using a discrete solid representation. *ACM Trans. Graph. 21*, 2, 88–105.

ARENBERG, J., 1988. Re: Ray/triangle intersection with barycentric coordinates. Ray Tracing News, 1.

BITTNER, J., WIMMER, M., PIRINGER, H., AND PURGATHOFER, W. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3, 615–624.

CHEN, B., AND NGUYEN, M. X. 2001. Pop: a hybrid point and polygon system for large data. In *Proc. IEEE Visualization*, 45–52.

CHIANG, Y.-J., EL-SANA, J., LINDSTROM, P., PAJAROLA, R., AND SILVA, C. T. 2003. Out-of-core algorithms for scientific visualization and computer graphics. IEEE Visualization 2003, Tutorial 4 Course Notes.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2003. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3, 505–514.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2004. Adaptive TetraPuzzles – efficient out-of-core construction and visualization of gigantic polygonal models. *ACM Transactions on Graphics 23*, 3 (August), 796–803. Proc. SIGGRAPH 2004.

COHEN, J. D., ALIAGA, D. G., AND ZHANG, W. 2001. Hybrid simplification: combining multi-resolution polygon and point rendering. In *VIS '01: Proceedings of the conference on Visualization '01*, IEEE Computer Society, 37–44.

COHEN-OR, D., CHRYSANTHOU, Y. L., AND SILVA, C. T. 2003. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9*, 3, 412–431.

CORRÊA, W. T., KLOSOWSKI, J. T., AND SILVA, C. T. 2003. Visibility-based prefetching for interactive out-of-core rendering. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 1–8.

DECAUDIN, P., AND NEYRET, F. 2004. Rendering forest scenes in real time. In *Proc. Rendering Techniques*, 93–102.

DEMARLE, D. E., GRIBBLE, C., AND PARKER, S. 2004. Memory-savvy distributed interactive ray tracing. In *Proc. Eurographics Symposium on Parallel Graphics and Visualization*, 93–100.

EL-SANA, J., SOKOLOVSKY, N., AND SILVA, C. T. 2001. Integrating occlusion culling with view-dependent rendering. In *VIS '01: Proceedings of the conference on Visualization '01*, IEEE Computer Society, 371–378.

ERIKSON, C., MANOCHA, D., AND BAXTER, W. 2001. HLODs for faster display of large static and dynamic environments. In *Proc. ACM Symposium on Interactive 3D Graphics*, 111–120.

GOBBETTI, E., AND MARTON, F. 2004. Layered point clouds – a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics 28*, 6 (December).

GOVINDARAJU, N. K., SUD, A., YOON, S.-E., AND MANOCHA, D. 2003. Interactive visibility culling in complex environments using occlusion-switches. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, 103–112.

GUTHE, M., BORODIN, P., BALÁZS, A., AND KLEIN, R. 2004. Real-time appearance preserving out-of-core rendering with shadows. In *Proc. Eurographics Symposium on Rendering*. June, 69–79 + 409.

HAVRAN, V. 1999. Analysis of cache sensitive representations for binary space partitioning trees. *Informatica 29*, 3, 203–210.

KLOSOWSKI, J. T., AND SILVA, C. T. 2001. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics 7*, 4, 365–379.

LEVENBERG, J. 2002. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings IEEE Visualization '02*, IEEE, 259–266.

LINDSTROM, P. 2003. Out-of-core construction and visualization of multiresolution surfaces. In *ACM 2003 Symposium on Interactive 3D Graphics*, 93–102,239.

LIVNAT, Y., AND TRICOCHE, X. 2004. Interactive point based isosurface extraction. In *Proc. IEEE Visualization*, 457–464.

MACDONALD, J. D., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 6, 153–165.

PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. 1997. Rendering complex scenes with memory coherent ray tracing. In *Proc. SIGGRAPH*, 101–108.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. QSplat: A multiresolution point rendering system for large meshes. In *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH 00)*, ACM Press, 343–352.

RUSINKIEWICZ, S., AND LEVOY, M. 2001. Streaming QSplat: A viewer for networked visualization of large, dense models. In *Symposium for Interactive 3D Graphics Proceedings*, 63–68.

STAMMINGER, M., AND DRETTAKIS, G. 2001. Interactive sampling and rendering for complex and procedural geometry. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, Springer-Verlag, 151–162.

WALD, I., DIETRICH, A., AND SLUSALLEK, P. 2004. An interactive out-of-core rendering framework for visualizing massively complex models. In *Proc. Eurographics Symposium on Rendering*, 81–92.

WAND, M., FISCHER, M., PETER, I., AUF DER HEIDE, F. M., AND STRASSER, W. 2001. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *SIGGRAPH 2001 Proceedings*, 361–370.

WIMMER, M., WONKA, P., AND SILLION, F. 2001. Point-based impostors for real-time visualization. In *Proc. Rendering Techniques*, 163–176.

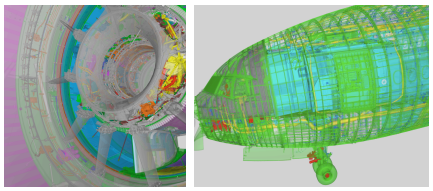
WOOD, D. N., AZUMA, D. I., ALDINGER, K., CURLESS, B., DUCHAMP, T., SALESIN, D. H., AND STUETZLINGER, W. 2000. Surface light fields for 3d photography. In *Proc. SIGGRAPH*, 287–296.

YOON, S.-E., SALOMON, B., GAYLE, R., AND MANOCHA, D. 2004. Quick-vdr: Interactive view-dependent rendering of massive models. In *VIS '04: Proceedings of the IEEE Visualization 2004 (VIS'04)*, IEEE Computer Society, 131–138.

ZHANG, E., AND TURK, G. 2002. Visibility-guided simplification. In *Proc. IEEE Visualization*, 267–274.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF III, K. 1997. Visibility culling using hierarchical occlusion maps. In *Proc. SIGGRAPH*, 77–88.

9 Ray Tracing with Multi-core/Shared Memory Systems



Ray Tracing on Multiprocessor Systems

State-of-the-art in Massive Model Visualization
Session 7
Abe Stephens

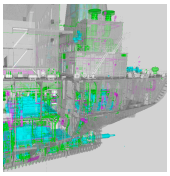
Scientific Computing and Imaging Institute
University of Utah

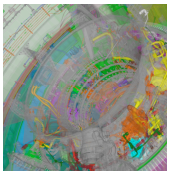
Overview

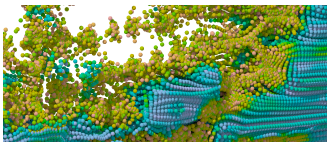
- Reasons for ray tracing massive models.
  - Massive model visualization.
  - Scientific visualization.
- Ray tracing fundamentals.
- Manta Ray Tracer software architecture.
- Parallel software programming & performance.

Massive Model Ray Tracing

- Basic intersection & shading techniques shared with serial renderers.
- No precomputed visibility.
- Transparency, hiding, or culling on the fly.
- Large data sets still in-core on moderate compute machines.
- Scalable software architecture from multi-core desktop to large server.
- Auto-realism, physically based shading.



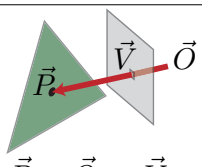




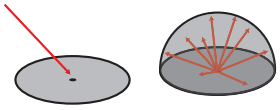
Gribble et al. *Interactive Particle Visualization Using Advanced Illumination Models.*

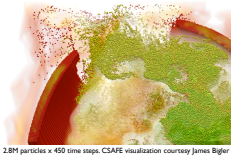
Ray Tracing Fundamentals

- Recursive algorithm.
- Basic operation:
  - Find the closest intersection point along a ray.
  - Compute normal, texture, material at intersection point.
  - Send secondary rays for shadows, reflections, other effects.
- Want to minimize the number of ray-object intersections necessary.
- Details on acceleration structures will be discussed in a subsequent session.
- Multi-modal visualization possible.



$$\vec{P} = \vec{O} + t\vec{V}$$

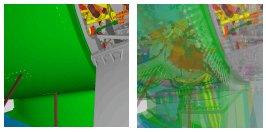


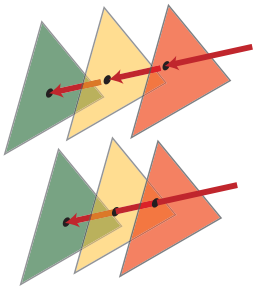


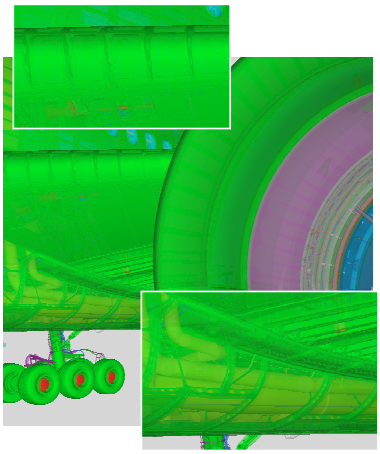
2.8M particles x 450 time steps. CSAFE visualization courtesy James Bigler

Transparency

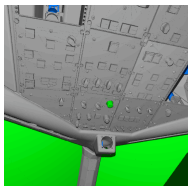
All Boeing 777 images rendered in Manta Interactive Ray Tracer. 3D data courtesy Boeing Company.

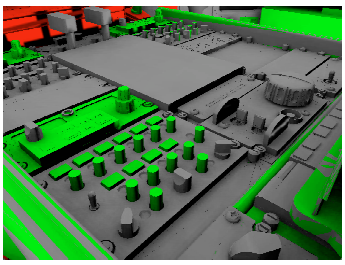


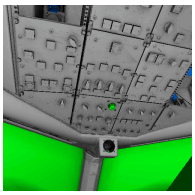


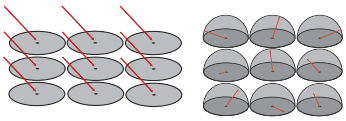


Controllable Automatic Realism





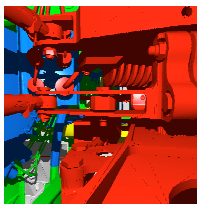




Interleaved sampling

Manta Software Design

- Portable interactive ray tracer.
- Shared memory systems (laptops to multiprocessors).
- Ray Packet based renderer.
- Modular design based on *parallel pipeline*.
  - Massive triangle models.
  - Massive volumes.
  - Glyph rendering.
  - Scriptable/Embeddable.
  - Shading language support.
- Open Source: http://code.sci.utah.edu/Manta/

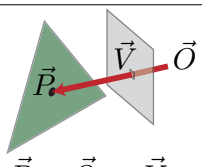


CSAFE visualization courtesy James Bigler

Stephens et al. *An Application of Scalable Massive Model Interaction using Shared Memory* Systems
Bigler et al. *Design for Parallel Interactive Ray Tracing Systems*

Rays & Ray Packets

- Single ray IRT systems less common.
- Packet based ray tracers operate on batches of rays.
  - Utilize aligned SIMD instruction sets.
  - Lazy evaluation of derived quantities.
  - Packet characteristics may be tracked and exploited using flags.



$$\vec{P} = \vec{O} + t\vec{V}$$

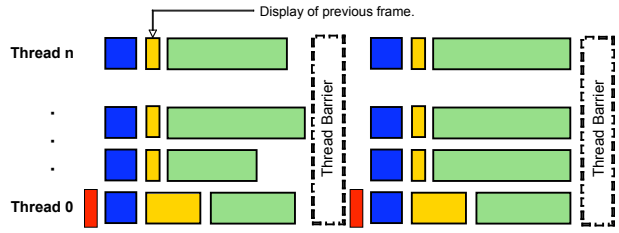
```
Real origin[3][MaxSize];
Real direction[3][MaxSize];
Real inverseDirection[3][MaxSize];
Real minT[MaxSize];
Real image[2][MaxSize];
Real normal[3][MaxSize];
Real ffnormal[3][MaxSize];
Real hitPosition[3][MaxSize];
Real texCoords[3][MaxSize];
Color color[Manta::Color::NumComponents][MaxSize];
int signs[3][MaxSize]; // 1=negative, 0=zero, positive
char scratchpad_data[MaxSize][MaxScratchpadSize];
```

$\vec{O}$ $\vec{V}$ t $\vec{P}$

```
enum RayPacketFlags {
  ConstantOrigin        = 0x0001,
  ConstantEye           = 0x0002,
  HaveImageCoordinates  = 0x0004,
  NormalizedDirections  = 0x0008,
  HaveHitPositions      = 0x0010,
  HaveUnitNormals       = 0x0100,
  HaveNormals           = 0x0200,
  HaveFFNormals         = 0x0400,
  HaveInverseDirections = 0x0800,
  HaveSigns             = 0x1000,
  ConstantSigns         = 0x2000,
  HaveCornerRays        = 0x4000,
};
enum PacketShape {
  LinePacket, SquarePacket,
  UnknownShape
};
```


Manta Parallel Pipeline

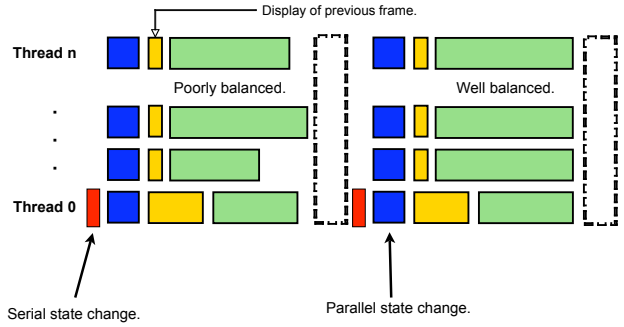
- Thread 0 performs display of the previous frame.
- State changes (red, blue) are performed at the beginning of the pipeline.
- Threads synchronize at barrier each frame.



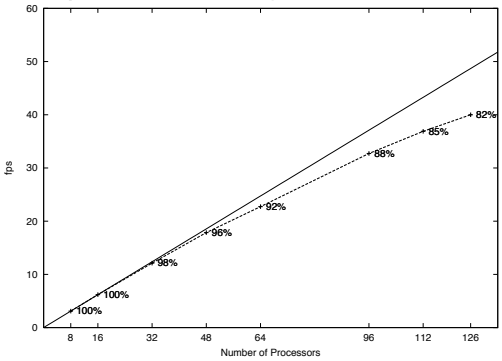
Manta Parallel Pipeline

- Events are specifically ordered:
  - Inherently balanced.
  - Imbalanced
  - Actively load balanced.
- Parallel inefficiency comes from serial portions of the code or from instances where the load balancer doesn't do a good job.

Manta Parallel Pipeline

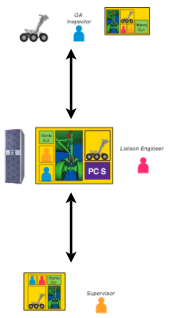


Manta Multiprocessor Scaling



SGI Altix 1.6 Ghz Itanium 2 (Fall 2005)

Remote / Collaborative Scenario



- Possibly dynamic data stored and rendered at one location.
- Visualization participants geographically separated.
- Each participant connects through an interactive session with aggressive compression.
  - e.g. SGI Vizserver application.
  - Application level frame compression possible.
- Remote visualization necessary more due to data management than rendering performance.

Parallel Ray Tracing Practices

➡Reduce high-level bottlenecks.

- Synchronous display
  - Causes other threads to block!
  - Easy solution: pipeline display with tracing.
- Shared read/write data structures in high performance code.
  Examples:
  - Locking during lazy accel. update.
  - Progressive refinement structures.
  - Producer/consumer queues.

Parallel Programming Practices

➡Reduce high-level bottlenecks.

- Use multi-processor summarization tools like *profile.pl*

```
=====================================================================
                              User

      Ticks     Percent  Cumulative   Routine
                          Percent
---------------------------------------------------------------------
        899       29.36    29.36      _ZNK5Manta6Kdtree6KDTree14intersect_node
        849       27.73    57.09      SCIRun::Barrier::wait(int)
        397       12.97    70.05      _ZN10V3C1Worker3runEv_LEH_1_48
        283        9.24    79.29      Manta::Kdtree::KDTree::intersectTriangles
```

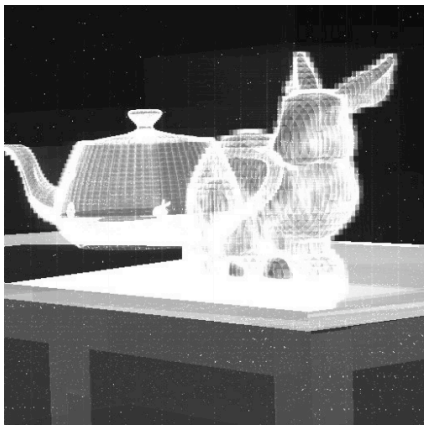

Parallel Ray Tracing Practices

➡Shared read/write data structures.

- Update during own pipeline stage.
- Per-thread copy of structure.
- Several threads share copy in neighborhood.
- Choice depends on application.
- Unsafe practice is not an option.

Load Balancing

- Work distribution across processors.
- Rendering cost may vary widely.
- Traversal and shading both effect ray cost.



Cost visualization courtesy Steven Parker

Load Balancing

- Coarse Load Balancing.
- Choose a strategy for assigning tiles to threads.
- Implement it as efficient as possible (hw intrinsics)
- Fine-grained Task Assignment
- Share read data, avoid common write data.
- How does ray coherence effect each level of parallelism for read/write?
- Complications: Lazy evaluation policies, Multi-thread scheduling.

10 Massive Model Visualization Using Realtime Ray Tracing

Massive Model Visualization using Realtime Ray Tracing

Andreas Dietrich Philipp Slusallek

Computer Graphics Group
Saarland University

1 Introduction

In the last years real-time ray tracing has become an attractive alternative to rasterization based rendering, particularly for highly complex datasets including both surface and volume data. Ray tracing [7, 15] is a much more flexible rendering algorithm than triangle rasterization found in most of todays graphics cards. Employing it in a real-time context might at first sound a bit surprising as ray tracing is mostly known for its application in high-quality off-line image generation, as e.g. in the motion picture industry. Infamous for its long rendering times, ray tracing was not used for interactive purposes until recently [13, 14, 19]. What makes it attractive for massive model rendering is not only its simplicity and robustness, but especially its versatility:

Support for advanced shading and lighting. Ray tracing closely models the physical process of light propagation by shooting mathematical rays into the scene. This enables the accurate computation of a broad spectrum of physical light scattering effects, ranging from simple specular reflection and refraction, up to more complicated cases such as glossiness or indirect light transport. Moreover, small custom programs that control illumination calculations, so-called *shaders*, can be deployed in a plug-and-play manner. This way characteristics of different surfaces can be described independently, and all optical effects are automatically combined in a physically correct way by the ray tracing engine.

Built-in visibility culling. Ray tracing starts by firing rays from the observer's eye through each pixel into a 3D scene in order to determine the closest intersection with the surfaces of the scene. As a result, visibility is evaluated on a per pixel basis, and typical types of *visibility culling* like *back-face culling*, *view-frustum culling*, and *occlusion culling* are handled implicitly.

High performance scalability. Because each pixel value can be computed independently from other pixels (if the scene is globally available), ray tracing is often called 'embarrassingly parallel'. Thus, image computation can be easily split between available processing units, which is even more important considering that multi-core processors are already becoming widely available even in consumer desktop systems.

Object instantiation. A very interesting property of ray tracing is support for simple instancing of objects. Here, an object including its geometry, textures, shaders, etc. is defined only once. Multiple instances can then be specified by simply storing transformation matrices. When the bounding box of an instance is hit, only the intersecting ray needs to transformed into the respective object coordinate system, where it is intersected with the object geometry.

Logarithmic scalability in scene size. Probably the most important feature of ray tracing when dealing with massive models is its logarithmic scalability with respect to scene size. In order to enable efficient ray–object intersection calculations, a ray tracer needs to organize the scene geometry in some sort of spacial index, which ensures that only the parts of the scene are accessed that are actually visible. As a consequence, for large scenes there is almost no performance impact if the geometric complexity is increased even further.

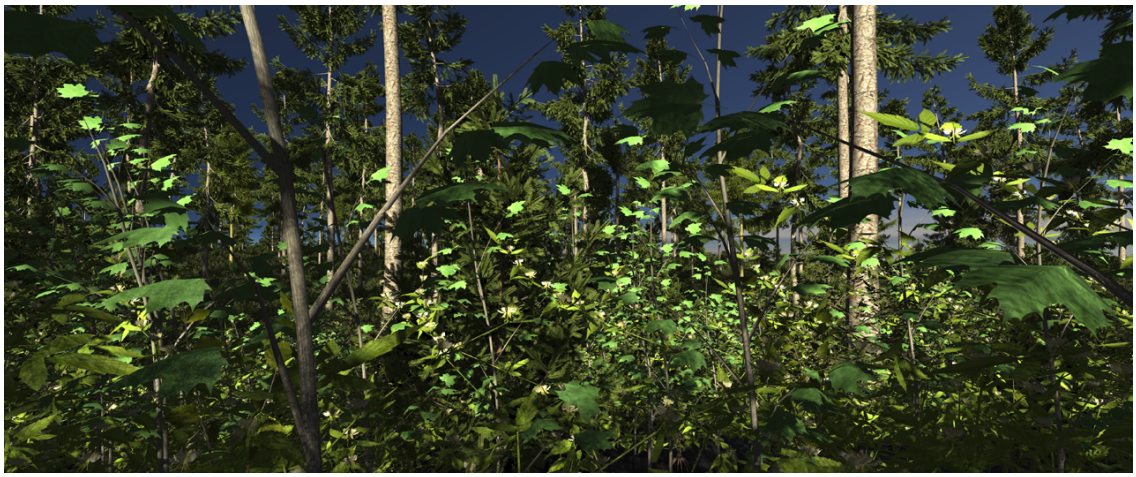


Figure 1: A complex plant ecosystem model containing more than 365,000 individual plants instances, resulting in a total of over 1.5 billion potentially visible polygons. The model can be ray traced interactively on a small number of desktop PCs.

Figure 1 shows an example of a visually complex scene, displayed using interactive ray tracing [5]. This model of a large natural ecosystem is made of 365,000 plant instances of 68 plant species. If we count every single primitive, the scene consists of more than 1.5 billion potentially visible triangles. Nevertheless, the scene can be interactively rendered using a small number of commodity PCs, even including optical effects like shadows, transparency, etc., and without any form of simplification.

While this example demonstrates that visual complexity, i.e., the number of visible primitives, is not really a problem for a ray tracer, it is in this case only possible because object instancing is used, which allows for storing the complete plant geometry on each render client PC. However, this is typically not the case for arbitrary massive models. Especially CAD models can be of tens to hundreds of gigabytes in size. Consequently, efficient data storage and access strategies are required to handle large scenes during preprocessing and rendering.

2 Data Preparation for Ray Tracing of Complex Scenes

Typically, a ray intersects only a few of the scene's objects. An ideal intersection algorithm would therefore only test these objects, or even better only the one with the closest hitpoint. In order to avoid unnecessary ray–primitive intersection tests, ray tracing algorithms employ spacial index structures or *acceleration structures*. During ray traversal, acceleration structures enable the quick identification of spatial regions that are not entered by the ray. This way large amounts of the scene's primitives can be effectively culled, and only for a limited number of primitives the actual geometric intersection calculation need to be performed.

2.1 Kd-Tree Construction and Traversal

Quite a variety of ray tracing acceleration techniques have been proposed in the past: For example, *octrees*, *binary space partition* (BSP) trees, *grids*, or *bounding volume hierarchies* (BVHs). (An overview can be found in Arvo and Kirk's chapter in [7].) In the following we will mainly focus on BSP trees, which are widely used in current ray tracing systems. More precisely, we describe axis-aligned BSP trees, so-called *kd-trees* [4, 18]. Construction of a kd-tree is very simple: We first start with the bounding box of the model and a list of contained primitives. The scene bounding box is then subdivided into two sub-boxes along one of the three primary coordinate axes. In the simplest cast we just select the longest side of the box, and split it in the middle. After that, the list of primitives is sorted into the two halves, creating two primitive

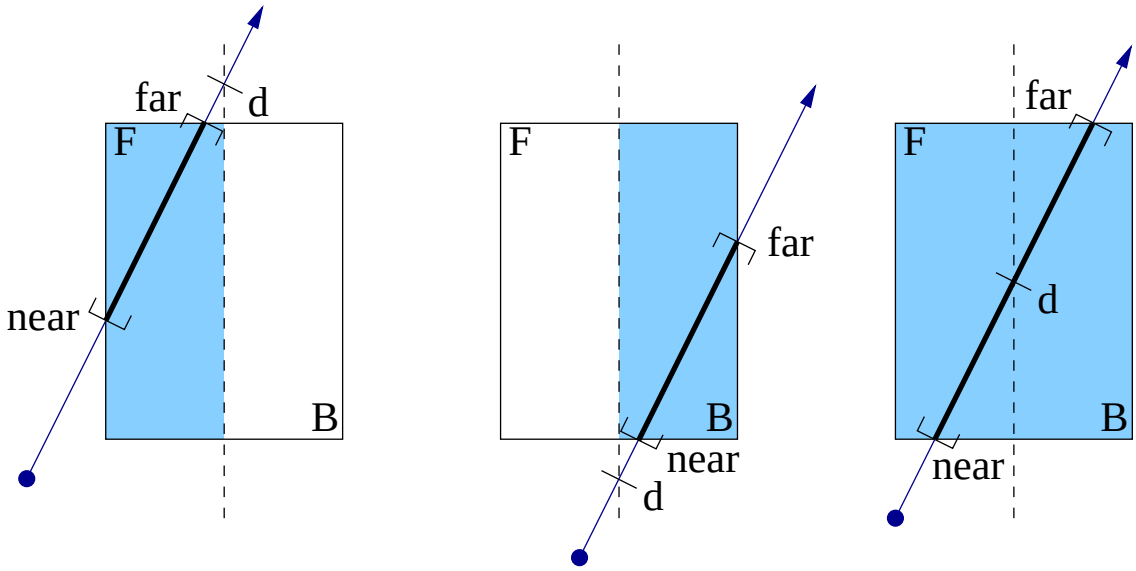


Figure 2: The three cases encountered during kd-tree traversal. Left: the far value of the ray interval is smaller than the distance d to the splitting plane, only the front child (F) is traversed. Middle: *near* is greater than d, only the back child (B) is entered. Right: d lies in-between *near* and far, first F then B is traversed. For both children the ray interval needs to be adjusted accordingly.

lists, one for each half. Polygons that lie in either half are simply replicated. The process is recursively continued for both sub-boxes and their respective primitive lists. This way a binary tree is constructed, where each node corresponds to a spatial region (called *voxel*), and its children to a binary space partition of their parent region. (The root node corresponds to the complete scene bounding box.) The recursive subdivision terminates if either a box contains not more than a predefined minimum number of primitives (typically 2–3), or if a maximum tree depth is reached. For each inner node we have to store an index indicating the axis affected by the splitting plane, the position of the splitting plane (one float value), and pointers to the two children. For each leaf of the kd-tree a list of indices of the contained primitives needs to be stored.

Traversal of such a kd-tree is also performed in a recursive manner: Again, we first start with the scene bounding box, i.e., the kd-tree root node, and clip the ray to be traced against it. This results in a ray segment $[near, far]$, where *near* is the distance from the ray origin to the bounding box entry point, and far the distance to the exit point. In addition, we compute the distance d from the ray origin to the splitting plane of the box. Depending on these value, three traversal cases can be distinguished (illustrated in Figure 2):

- $d > far$: The ray only crosses the voxel of the front child (F) of the node. Traversal continues with F, using the same ray segment as for its parent.
- $d < near$: The ray only crosses the voxel of the back child (B) of the node. Traversal continues with B, using the same ray segment as for its parent.
- $near \leq d \leq far$: The ray crosses the voxels of both children. The address of B, as well as far are pushed onto a stack, and traversal continues recursively with F with $[near, d]$ as new ray segment. As soon as the front node F has been fully traversed, B and far are pulled from the stack and B is traversed recursively with $[d, far]$ as ray interval.

This simple algorithm realizes an efficient front-to-back space traversal. The decision, whether a child node is in the front, or in the back can be based on the sign of the ray direction. Once a leaf is encountered, all included primitives are intersected sequentially with the ray, where the hitpoint closets to the ray's origin

is determined. In case a hitpoint *inside* the leaf voxel can be found, traversal is stopped, and the hitpoint together with the respective primitive ID is returned. Otherwise, the next node and the according far value are pulled from the stack, and traversal continues recursively with this node with the old far value becoming the new *near* value of the ray segment. More details can, e.g., be found in [18].

The quality of kd-trees, i.e., the placement of splitting planes, is crucial for interactive performance. Instead of recursively partitioning space into two equally sized halves, cost prediction functions can be applied to guess optimal splitting plane positions. For example, *surface area heuristics* (SAHs) [12] can serve as cost functions, which estimate the impact of a certain splitting plane position on ray traversal performance.

2.2 Multi-Level Acceleration Structures

Instead of using a single kd-tree for a complete scene, it can be often beneficial to use a hierarchy of kd-trees. Figure 3 shows an example of a two-level variant [20]. In this example, a scene can consist of several independent objects composed of geometric primitives, each object having its own local index structure. The bounding boxes of those objects are themselves organized in a top-level kd-tree. There are two intentions behind this setup:

First, it allows for efficient object *instancing* by simply keeping object references in combination with corresponding transformation matrices. If a ray hits an object in the top-level index, it gets transformed into object-space, where it then traverses the object's internal kd-tree structure. [1] Let's assume we have a ray $\mathbf{r} = \mathbf{o} + t\mathbf{d}$ (with origin $\mathbf{o}$ and direction $\mathbf{d}$ in homogeneous coordinates) and a transformation matrix $\mathbf{T}$ that transforms an object's geometry from its local coordinate system into world coordinates. We then simply need to intersect the ray $\mathbf{r}' = \mathbf{T}^{-1}\mathbf{o} + t\mathbf{T}^{-1}\mathbf{d}$ with the original object, i.e., traverse its local kd-tree and intersect its primitives. If a hitpoint in object-space is found, the resulting hit distance value t is inserted into the original ray equation, which yields the hitpoint coordinates in world-space.

Second, rigid-body motion can be handled. Typically, in dynamic scenes large parts often remain static over long periods of time, other parts of a scene undergo well-structured transformations such as rigid motion. Yet other parts might change in a totally unstructured way. This can be exploited by decomposing a scene into separate objects according to their dynamic properties. In the case of rigid-body motion only the top-level index structure needs to be rebuild once an object changes it position or orientation, while object kd-trees can be fully reused.

Instancing can be particularly effective for scenes with many equal or similar objects. Some examples where instancing has been applied to form highly complex natural scenes can be seen in Figure 4. The human eye can be fooled by sheer geometric complexity such that a scene can appear authentic even if all

[1] Typically, object bounding boxes will intersect more than one voxel of the top-level tree. In order to avoid having to intersect an object multiple times, *mailboxing* [2] can be used to record if an object has already be intersected by the current ray.

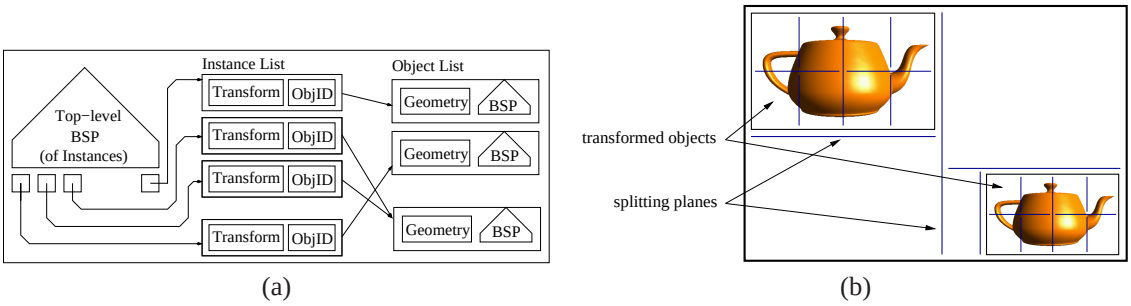


Figure 3: Two-level kd-tree hierarchy. (a) A top-level BSP tree contains references to instances, each containing a transformation and a reference to an object. Objects themselves consist of their geometric description plus a local BSP tree. Multiple instances can refer to the same object with different transformations. (b) A Simple example with a single teapot object instantiated twice.

(a) (b)

Figure 4: Highly complex natural scenes generated using instancing. (a) Another shot of the forest from Figure 1. (b) A model of the Puget Sound area, instancing is performed at multiple levels. The ground terrain is covered with billions of plant instances. Each plant model is made of several thousand polygons [6].

plants are absolutely identical. Empirical investigation has shown that individually bent and scaled copies of a few plant species are usually sufficient to generate a plausible impression of a complete landscape.

2.3 Streaming Kd-Tree Construction

Construction of a (multi-level) spatial index poses no challenging problem for source models that are already spatially grouped or partitioned into individual objects – and that the data for a single object fits into main memory. In this case we just build the object kd-trees one by one. We load the geometric source data for each object in turn into memory, construct the tree, and write the complete object (see Figure 3a) to disk. (We will see in Section 3 how models can be rendered out-of-core.)

Unfortunately, typical massive models are usually not that trivially to handle. Often the source data already requires many gigabytes of hard disk space, and does not completely fit into memory. Moreover, objects are often grouped based on their function. As a result their bounding boxes cover large parts of the scene space and overlap heavily. This would cause extensive sequential testing and severe performance degradation if we would follow the above approach. Finally, large models often come as a 'soup of triangles', i.e., without any mesh connectivity information.

However, there is a very easy way to construct a kd-tree index structure for such models efficiently in an out-of-core fashion [21]: We just take a stream of source triangles (usually contained in one large file, which is generated from the original model description), and first compute its bounding box. Then we split this parent box into two disjunct child boxes. Finally, we go over the initial triangle stream again, and sort every primitive into two new files, one for each sub-box. By applying this scheme recursively, we can generate a binary space partition. As soon as a triangle file can be processed in-core, we construct a binary representation that contains the local kd-tree for this subset of primitives plus all geometry and shading information. This binary data is afterwards stored as a single file. Details on how to efficiently generate such a representation can be found in [19]. The generated objects are finally organized by a top-level kd-tree (see above) that can be generated by keeping track of the split operations during preprocessing.

This approach is easily to parallelize in shared-memory environments, but also in a PC cluster setting with a network file system. After each splitting step, we simply start an additional splitting process, resulting in two separate processes working independently on the two children. Furthermore, several optimizations are possible in the preprocessing step: A straightforward enhancement is to reduce the costs of I/O operations by working only on the geometric description, i.e., on triangle corner points. All additional data contained in the source model, like shading normals, texture coordinates, etc., is separated from the triangle stream before splitting, and then re-added once the binary files are generated. Another obvious optimization would be to apply SAH cost functions already during preprocessing in order to determine advanced splitting plane positions.

3 Adapting Ray Tracing to Complex Models

As motivated in Section 1, interactive ray tracing of visually complex scenes is uncomplicated as long as the scene geometry and acceleration structures fit completely into main memory. For very large scenes (ranging from tens to hundreds of gigabytes of data) not only kd-tree generation, but also rendering must be performed out-of-core. This means that the system memory has to serve as a cache, where only the actually needed data remains in RAM.

3.1 Memory Management

Memory management of complex scenes can be done in a very straightforward manner on 64-bit architectures, which today are widely available, even in commodity PCs. The whole dataset, including acceleration structures, geometric descriptions, normals, texture coordinates, etc., is simply mapped from disk into virtual address space using the operating system's memory mapping facilities (e.g., `mmap()`). The following examples will assume Linux, but similar mechanisms exist in all modern operating systems. [2]

Making use of the OS memory mapping system has a number of advantages: First, it provides an automatic demand paging scheme, taking care that data is loaded into main memory as soon as it is accessed. Second, there is no memory fragmentation since we operate in virtual address space, and address translation is handled by the memory management unit (MMU) of the CPU. Third, the OS works on a per-page basis (typically pages are of 4 KByte size), which allows for a very fine cache granularity. Moreover, this way a unified cache is realized, i.e., it does not matter which data is contained in which physical page (e.g., geometric, shading, or texture information).

Although an OS-based memory management system has many advantages over manual caching, it also has the major drawback in that we lose control over what data is loaded into or discarded from memory at what time. Even though data is automatically paged in on demand upon accessing it, the resulting page fault stalls the rendering process until the data is available. To retain control over the caching process a hybrid memory management system can be implemented, which uses the operating system to perform demand paging, but which detects and avoids potential page faults before they occur, and which manually steers page loading and eviction.

In order to avoid page faults during ray traversal and intersection, it is necessary to detect whether or not memory referenced by a pointer is actually in core memory. Though Linux for this purpose offers the `mincore()` function, performing an OS call on *each* memory access obviously is not affordable. When taking a closer look at the Linux memory mapping implementation, however, there are several important observations to be made: First, after having once loaded a page, it will stay in memory at least for a limited amount of time. Second, pages in memory will not be overwritten as long as there is some unused memory available. Thus, as long as we know that there is some memory left, we can mark once-accessed pages, and can be (almost) sure that the respective page will still be in memory later on. Obviously, this only works as long as we do not try to page in more data than fits into physical memory. Fortunately, this can be easily guaranteed: By using the Linux `madvise()` call, it is possible to advise the OS kernel to free pages, thereby guaranteeing that some free memory is available at any time, and that no pages become unavailable without us knowing it. Of course, this assumes that no other processes start using up memory.

3.1.1 The Tile Table

In order to mark pages as either available or missing, one bit per page would have to be stored. Keeping an entry for each potential 4 KB page in a 64-bit address space would require 2^{52} entries and is not affordable. It is therefore necessary to group several pages into one *tile*, and keep the tiles organized in a hash table of tile addresses. If the hash table is large enough to minimize hash collisions, hashing is quite efficient, and can be implemented with a few bit operations on the address pointer. Furthermore, a hash table is quite memory efficient: For handling 128 GB RAM composed of 4 KB sized tiles (one page per tile) we only need 2^{25} entries. Using a larger cache granularity of 16 KB or 64 KB, this reduces even more to 2^{23} and 2^{21} entries, respectively. If the size of the tile table is a power of two, all addressing and hashing operation can be performed efficiently by simple binary `and` and `shift` operations.

[2] Readers interested in OS technology can find an in-depth description in [8].

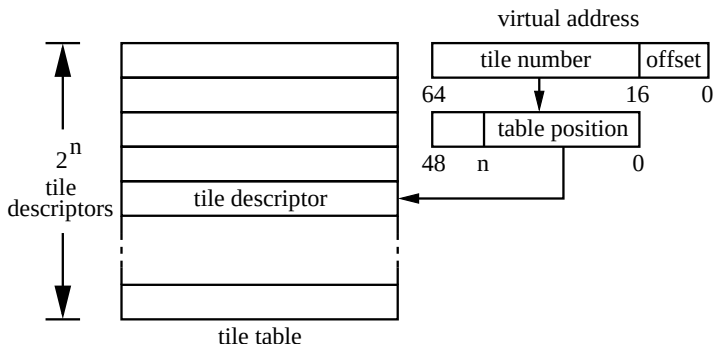


Figure 5: Use of a hash table to record which memory page/tiles are available in RAM. The tile descriptor is determined from the virtual address by simple bit masking operations.

Figure 5 illustrates how the tile table can be realized. Each entry contains a 64-bit pointer with the virtual base address of the tile for detecting hashing collisions. The lower 12–16 bits of this entry are always zero, and can thus be used for other purposes, i.e., for marking whether the page is available (bit 0), and whether it has recently been referenced (bit 1). In order to check if a page is in memory, we simply have to find its entry in the tile table (one `shift` operation), validate there is no hash collision (one `and`), check bit 0 for availability (one more `and`), and, if required, set bit 1 (one `or`) to mark an access.

3.1.2 Tile Fetching

In case a tile is found that is not marked as available, we cancel the respective ray, and schedule the tile's for asynchronous loading by putting its address into a request queue. To take care of loading requested tiles, a separate *fetcher* thread should be set up that asynchronously checks the request queue, and triggers IO operations if necessary. In an infinite loop, this thread in each iteration takes one request from the request queue, reads in the page via `madvise()`, and then marks the tile as available. Though reading the page obviously stalls the fetcher thread, the ray tracing threads are not affected at all, and image display remains interactive.

Missing data leads to cancelation of rays, so missing data that cancels many rays should be fetched faster than data affecting only a single ray. It can be observed that the number of affected rays is proportional to the size of the BSP voxel they are about to enter, and inversely proportional to its distance to the camera. We can use this value for prioritizing fetch requests. To avoid searching for the most important requests, we map the priority to 8 discrete values, and keep one request queue for each of them. The fetcher then always takes the first entry out of the queue with highest priority. This mapping is performed linearly, relative to the minimum and maximum priorities of the previous frame.

3.1.3 Tile Eviction

As mentioned before, the tile fetcher should only load new tiles if some unused memory is available. Otherwise, the OS overwrites tiles/pages without us even noticing it (i.e., in the table they are still being marked available). Therefore, the `madvise()` function is used to discard mapped pages from main memory. This obviously should be done only for pages that are likely not needed any longer. As a full *least recently used* strategy would be too expensive, we follow the same strategy as the Linux kernel, and use a *second chance* strategy. A separate tile evictor thread slowly but continuously cycles through the tile table, and resets the tile's *referenced* bit to zero (the page is still marked as present). If the tile is still needed, this bit will soon be re-set by a rendering thread. If, however, the evictor visits a tile a second time with the referenced bit still zero, it evicts the tile, and marks it as missing. Similar to the Linux kernel swapper, tile eviction only starts once memory gets scarce, e.g., at a memory utilization of $\sim$80%.

3.1.4 Optimizing Table Lookups

Apart from the time consumed by the asynchronous fetcher and evictor threads, the main ray tracing threads have to constantly check before each memory access for availability of the data. To minimize this overhead, it is advantageous to first check each pointer dereference if it crosses a tile boundary (with respect to the previous access). This can be done quite efficiently by simple bit operations, already reduces most of the tile table lookups, and does not require any costly semaphore operations.

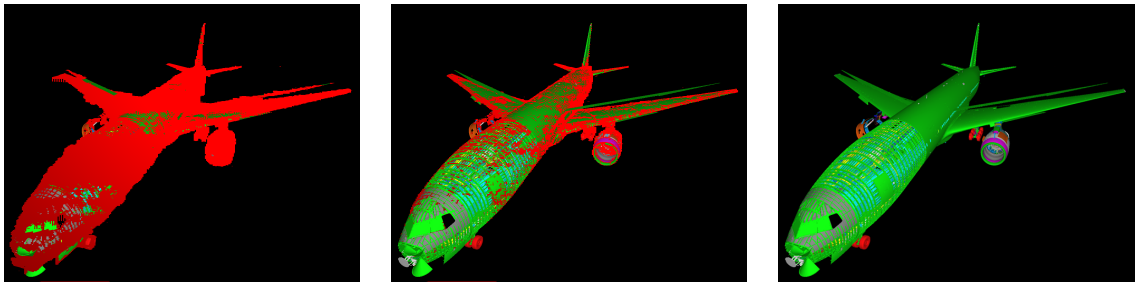
Even in the case the tile table has actually to be accessed, we can often get away without having to perform locking operations. If the tile is marked as available, or is marked as already being fetched, we can immediately return. Though this can result in a race condition – e.g., the evictor might evict the tile at exactly this moment – this event is extremely improbable. Even if it occurs, in the worst case it can lead to either a single, page fault, or to scheduling a tile twice for being loaded. There are only two cases where a ray tracing thread has to use a mutex look. Once it adds a previously unvisited tile to the tile table, and when the tile is added to the request queue. We also have to lock a mutex every time the tile fetcher or tile evictor want to modify the tile table or request queues. These threads, however, are not performance critical.

3.2 Bridging Loading Time

The above described hybrid memory management scheme allows for efficiently detecting potential page faults. Thus, stalling of the ray tracing process is avoided, which ensures interactivity. However, the question arises what to do in case traversal of a ray has actually to be canceled.

3.2.1 Canceling Rays

A trivial procedure to deal with canceled rays is to just skip shading, and assign a fixed color to affected pixels. Figure 6 shows the effect for a CAD model of a Boeing 777 consisting of more than 350 million triangles. In this setup, the dataset is visualized on a dual-processor PC, fitted with two AMD Opteron CPUs (1.8 GHz), and with 6 GB of physical main memory. In (a) the model is displayed directly after startup. After a few seconds, large parts of the model have be loaded (b), shortly afterwards the full airplane is visible (c). While this simple procedure causes artifacts as soon as moving the camera triggers new loading operations, a user can still interactively navigate through the model. After camera motion stops, a complete image gets available after a few seconds, since only the actually visible parts of the model are loaded.



(a) (b) (c)

Figure 6: Detecting potential page faults. Tracing of rays that require to access not-yet-loaded data is canceled, and the respective pixels are colored red. (a) Directly after start-up. (b) After a few seconds. (c) The complete model data for this particular view (more than 2 GB) has been loaded.

3.2.2 Suspending Rays

For smaller scenes (i.e., in the order of tens of millions of triangles), where the working does not change drastically during movements, there is another alternative to deal with rays that encounter not-yet-loaded parts of a models [21]. When a potential page fault is detected, the respective ray is suspended, and the responsible render thread tries to continue with another pixel, while the requested scene data is loaded asynchronously. Once everything has been loaded, the render thread resumes ray tracing of suspended pixels. Unfortunately, this approach has the disadvantage that it can only hide loading latency within a single frame. For larger models (like the Boeing 777) even small changes of camera position and orientation can trigger reading large amounts of model data that cannot be loaded during one frame.

3.3 Simplified Model Representations

An approach better suited for larger models is to bridge loading time by displaying a coarser version of the model until the fully detailed geometry has been fetched from disk. The idea is to have at any time a simplified, but small-sized, version of the scene in-core that can serve as a replacement.

During the last years a plethora of methods and algorithms have been proposed dealing with the generation of simplified representations from highly detailed source models. These approaches range from polygonal simplifications methods, over alternative geometric descriptions like point-based or volumetric representations, up to image-based methods, like e.g., impostors that act as replacement for complex geometry. In the following example we applied polygonal simplification methods. For models consisting of a 'soup of triangles', so-called *vertex clustering* methods can be recommended, which merge vertices inside a specific region into a single representative vertex. Such methods do not require well behaving polygon meshes as input, and can cope with non-manifold surfaces. An excellent survey of polygonal simplification algorithms can be found in [11].

From the original triangle model we create n coarser levels-of-detail (LODs). Level 0 relates to the original geometry, level n to the coarsest representation. Ideally, data reduction should be adjusted so that level n fits completely into main memory. For each ray to be trace we start traversal of level 0 using the hybrid memory management system described in Section 3.1. If memory requests for a certain piece of data returns false, we try to continue the ray using level 1. In case the needed part is not available in level 1, we iteratively proceed until the coarsest level n is reached. Because level n is already in memory, it can be directly accessed without risking a page fault, which would otherwise stall rendering. The job of the intermediate levels (level 1 up to level $n-1$, with $n > 1$) is to replace the coarsest level as soon as possible in order to achieve a better model approximation, as coarser levels can be loaded faster than finer levels. In the worst case always level 0 is available.

An example application of this method can be seen in Figure 7. From the original Boeing 777 model we created two simplification levels, containing about 25% and 5% of the number of triangles of the original high-detail model. It can be observed that LOD rendering based on vertex clustering delivers a very good approximation immediately after start-up. After a few seconds the artifacts of the most drastically simplified version are replaced by the intermediate level, where a difference to the original model can hardly be seen. Eventually, the finest level is loaded, and the model is directly rendered without any form of simplification.

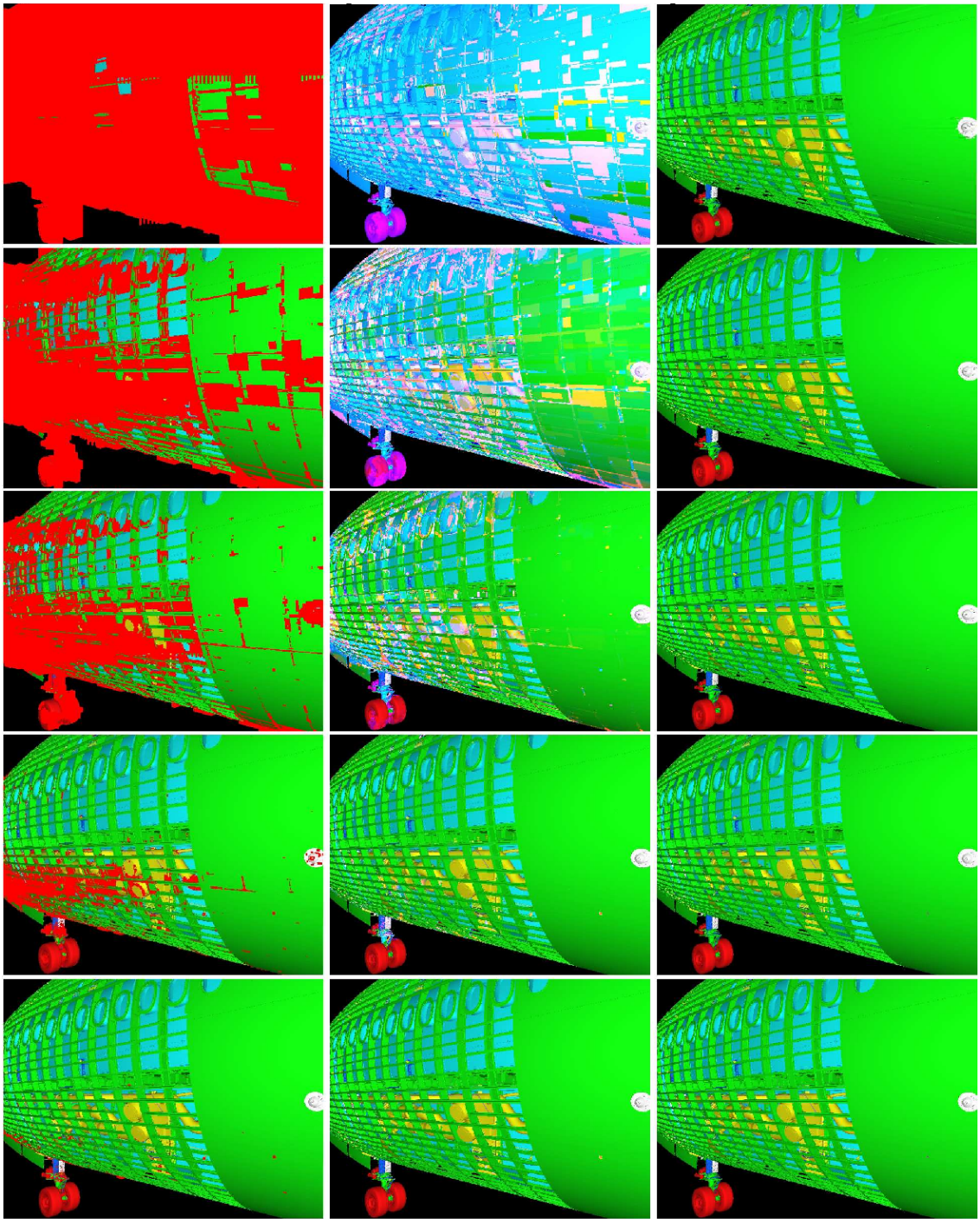


Figure 7: LOD rendering of the Boeing 777 model. From top to bottom screen-shots from frame 3, 9, 13, 25, and 37 are shown. Left: Unavailable data is represented by coloring the respective pixels red. Middle: False-color display, with the intermediate level (25%) highlighted red, and the coarsest level (5%) blue. Right: LOD display without false-colors.

4 Photorealistic Rendering and Lighting of Complex Models

An attractive aspect of ray tracing is the flexibility and power of shading operations. On the one hand this provides ways and means to generate visually convincing and highly realistic images. On the other hand it allows for conceiving all sorts of high-level abstraction visualization techniques that can aid humans in grasping the information contained within complex designs.

4.1 Advanced Shading

The impact of enhanced visual realism can be observed in Figure 8a. A planar area light source is attached to the camera position, which enables the computation of smooth shadows and specular highlights (note that the 777 model does not include any material or shading data, except a single color value per surface). In particular for complex geometry, the projection of 3D data onto a 2D display often incurs an undesired loss of depth impression. In that case, shadows often significantly help in the perception of the relative positions of objects. Especially *soft* shadows ease judging the distance between shadow caster and receiver.

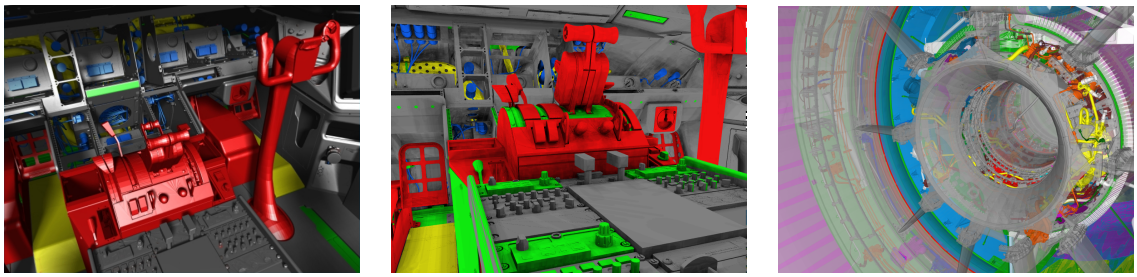
Such effects work fully automatic, e.g., a surface shader simply has to fire shadow rays towards light sources if it wants find out how the surface is lit. No multi-pass rendering operations are required, and even totally diverse surface types – glass maybe – can be added without having to rework already present shaders.

Other techniques such as *ambient occlusion* [22] take a step closer to photo-realism. This shading technique estimates the ambient light at every point in the model based on occluding geometry in the point's neighborhood (Figure 8b). This emphasizes intricate details, which are not always easily detected with lights and shadows. The technique can be implemented using an interleaved sampling scheme, which distributes secondary rays used by the shading technique between a neighborhood of pixels. The results from this neighborhood are then filtered to produce the final image [17].

Figure 8c is an example of a quite different style, oriented more towards abstraction and illustration than towards realism. Transparency is simply implemented by tracing the rays coming from the eye further into the model. To prevent a major frame rate drop, we track the amount of light energy that is absorbed by the affected surfaces. If a certain opacity threshold is reached, ray traversal can be discontinued. Thus, the maximum recursion depth can be controlled adaptively.

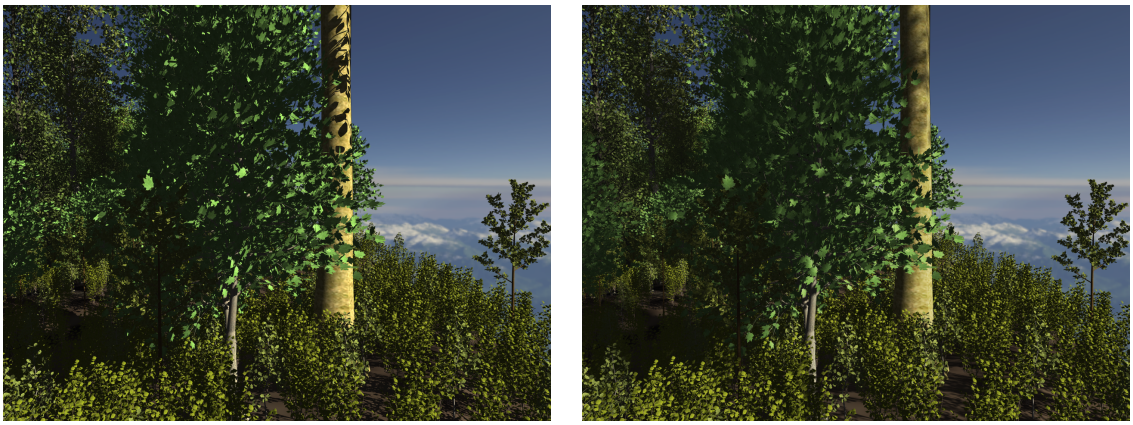
4.2 Realistic Environmental Lighting

Realistically structured natural outdoor scenes are in contrast to CAD models much more difficult too handle. Typically, they are composed of many vegetation layers, which consist of countless irregular structured plants models (as could be seen in Figures 1 and 4). In addition to a high degree of detail, a realistic representation of natural outdoor scenes heavily depends on the incident illumination, i.e., the



(a) (b) (c)

Figure 8: Some examples of advanced surface shading effects. (a) Soft shadow effects in the 777 cockpit providing a better impression of the relative placement of components. (b) Ambient occlusion techniques result in an even more accurate lighting by computing the ambient light at visible surface points. (c) Adding transparency significantly helps in understanding the complex part arrangement in the engine.



(a) (b)

Figure 9: Visual influence of environmental high dynamic range lighting. (a) Approximate sun light using a single directional light source. (b) Illumination based on an HDR environment map. Note particularly the soft shadows on the tree trunk and the tree foliage.

light coming from the environment. Approximation of natural lighting with e.g., a single directional light, is usually not sufficient (see Figure 9) as it cannot capture the subtle but important effects like smooth shadows.

The cost of pre-calculating light transport, e.g., using *precomputed radiance transfer* (PRT) [16] methods, is not affordable in scenes of such a high complexity, in particular when making heavy use of instantiation. An alternative is therefore to directly sample the incident radiance emanating from a *high dynamic range* (HDR) environment map. In order to avoid noise, resulting from random sampling of such a map, it is better to use a large but fixed number of virtual directional light sources in the spirit of [1, 10] that act as an approximation of the environment illumination. Unfortunately, it is too costly during shading to fire as many shadow rays as there are virtual light sources. For that reason we are picking only a small number of random samples from the full set of virtual directional lights.

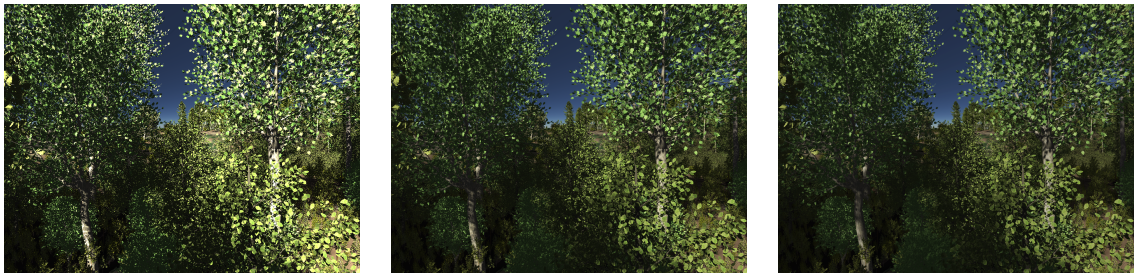
Although the addition of environmental illumination provides significantly more realism, it is commonly not affordable to interactively compute a full global illumination solution for such scenes. Because of this, important effects like diffuse leaf transparency, as well as inter-object light scattering are not accounted for in this example.

4.3 Interleaved Sampling

Aliasing poses a difficult problem in scenes with the geometric complexity of the described examples. This does not only account for primary rays, but also for shadow rays – rays that sample the environment. To make best use of a given ray budget, it is advantageous to combine anti-aliasing caused by geometry and illumination [9, 3]. Rather than using the same set of virtual directional lights for every primary ray, we can break up the light source set into subsets, each of which applies to a different set of primary rays. That way we can increase the number of light samples while simultaneously improving pixel-anti-aliasing.

4.4 Progressive Refinement

The above techniques allow for quickly generating high-quality images at interactive rates on a set of PCs. However, this visualization still requires significant compute power. In particular high-resolution images are quite demanding. However, it is possible to produce an image in a progressive way: As soon as camera motion stops, we perform progressive improvement of image quality by computing successive images of the same viewpoint with new random samples, and accumulate the resulting images. This makes it possible to generate high-quality images of high-resolution even on a single PC in just a couple of seconds. During



(a) (b) (c)

Figure 10: Progressive refinement: An image with an increasing accumulation of intermediate results. (a) No accumulation. (b) After 4 accumulation steps. (c) After 10 accumulation steps.

user interaction pictures with reduced quality can still give a very good impression of the final result. Figure 10 shows an example of interleaved sampling combined with progressive rendering. In this setting one primary ray per pixel was fired, as well as two shadow rays per hitpoint. During image accumulation, for each new frame primary rays were jittered, and new virtual directional light sets were used.

References

[1] Sameer Agarwal, Ravi Ramamoorthi, Serge Belongie, and Henrik Wann Jensen. Structured Importance Sampling of Environment Maps. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, pages 605–612, 2003.

[2] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Computer Graphics Forum*, pages 3–10, 1987. (Proceedings of Eurographics).

[3] Carsten Benthin, Ingo Wald, and Philipp Slusallek. A Scalable Approach to Interactive Global Illumination. In *Computer Graphics Forum*, pages 621–630, 2003. (Proceedings of Eurographics).

[4] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[5] Andreas Dietrich, Carsten Colditz, Oliver Deussen, and Philipp Slusallek. Realistic and Interactive Visualization of High-Density Plant Ecosystems. In *Natural Phenomena 2005, Proceedings of the Eurographics Workshop on Natural Phenomena*, pages 73–81, 2005.

[6] Andreas Dietrich, Gerd Marmitt, and Philipp Slusallek. Terrain guided multi-level instancing of highly complex plant populations. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 169–176, 2006.

[7] Andrew Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.

[8] Mel Gorman. *Understanding the Linux Virtual Memmory Manager (Bruce Perens' Open Source Series)*. Prentice Hall, 2004.

[9] Alexander Keller and Wolfgang Heidrich. Interleaved Sampling. In *Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering)*, pages 269–276, 2001.

[10] Thomas Kollig and Alexander Keller. Efficient Illumination by High Dynamic Range Images. In *Rendering Techniques 2003 (Proceedings of the 14th Eurographics Workshop on Rendering)*, pages 45–50, 2003.

[11] David P. Luebke. A Developer's Survey of Polygonal Simplification Algorithms. *IEEE Computer Graphics and Applications*, 21(3):24–35, 2001.

[12] J. David MacDonald and Kellogg S. Booth. Heuristics for Ray Tracing using Space Subdivision. In *Proceedings of Graphics Interface '89*, pages 152–163, 1989.

[13] Michael J. Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium '95*, 1995.

[14] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics*, pages 119–126, 1999.

[15] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing, Second Edition*. A K Peters, 2003.

[16] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, pages 527–536, 2002.

[17] Abraham Stephens, Solomon Boulos, James Bigler, Ingo Wald, and Steven Parker. An Application of Scalable Massive Model Interaction using Shared-Memory Systems. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, pages 19–26, 2006.

[18] Kelvin Sung and Peter Shirley. *Ray Tracing with the BSP Tree*. Academic Press, 1992.

[19] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[20] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[21] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering)*, pages 274–285, 2001.

[22] Sergey Zhukov, Andrei Iones, and Grigorij Kronin. An Ambient Light Illumination Model. In *Rendering Techniques 1998 (Proceedings of the 9th Eurographics Workshop on Rendering)*, pages 45–56, 1998.

Massive Model Visualization using Real-time Ray Tracing

SIGGRAPH2007

Course
State of the Art in Massive Model Visualization

Andreas Dietrich Philipp Slusallek
Saarland University & inTrace GmbH

Overview

- Simplicity of data preparation for ray tracing
- Adapting ray tracing to complex models
- Photorealistic lighting of highly complex models
- Review of hardware-trends for real-time ray tracing

Ray Tracing

- Simple algorithm, with many advantages
  - Support for advanced shading and global illumination
    - Not *directly* related to massive model rendering…
  - Visibility culling built-in
    - Includes view-frustum, back-face, and occlusion culling
    - Per pixel visibility
  - Supports object instantiation
  - Trivially parallelizable
  - Logarithmic scalability in scene size
    - Due to traversal of (hierarchical) spatial index structures

Ray Tracing

- Simple algorithm, with many advantages
  - Support for advanced shading and global illumination
  - Visibility culling built-in
  - Supports object instantiation
  - Trivially parallelizable
  - Logarithmic scalability in scene size

➔ Complex models: „log scalability“ most important!

Ray Tracing for Massive Models

- Visibility not the main problem

- Proof by example: “Forest” scene
  - 1.5 billion triangles
  - Plus shadows, textures, transparency, …
  - Rendered interactively on few PCs

Storage Problem

- Number of visible triangles not main problem
  - ➔ Main problem: Efficient scene storage and access !
- The storage problem
  - Logarithmic cost: Assumes all data is in memory
  - “Forest” example:
    - Only possible through instantiation → special case !
  - For general complex models usually not the case
    - Boeing 777: 30 - 40 GB on disk (including spatial index)

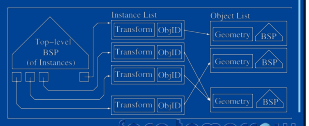
➔ Need efficient data handling for preprocessing and rendering

Part I

Simplicity of Data Preparation for Ray Tracing

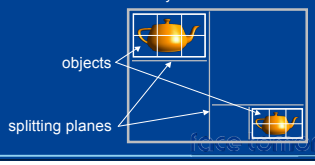
Spatial Index
Hierarchy and Instancing

- Two-level kd-tree scheme [Wald et al. 2003]
  - Accelerates ray-object intersection computation
  - Object-level kd-tree for each object type
  - Top-level kd-tree organizes instances
    - Object references
    - Object bounding boxes
    - Transformation matrices



Spatial Index
Hierarchy and Instancing

- Two-level kd-tree scheme enables
  - Rigid-body dynamics
    - Only top-level kd-tree has to be rebuilt
  - Efficient instancing
    - objects can be reused with little memory overhead



Index Generation

- Simple case
  - Source model grouped/partitioned into individual objects
  - Object boxes are not extensively overlapping
  - Data for single object fits into memory

➔ Build kd-trees independently
  - Build object-level kd-trees one after another
  - Build top-level scene kd-tree based on objects boxes

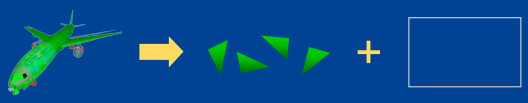
Index Generation
Streaming Approach

- Massive models require many GB of data
  - Data often too large to build spatial index fully in-core
  - Objects typically grouped functionally not spatially
  - Model often exported as "soup of triangles"

➔ Divide and conquer strategy

Index Generation
Streaming Approach

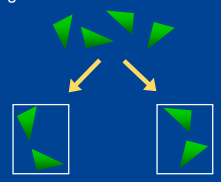
- Offline index generation
  1. Produce triangle stream (file) from source data and compute bounding box



  2. Split bounding box into two halves along longest side

Index Generation
Streaming Approach

- Offline index generation
  3. Go through triangle stream and sort each triangle into the new bounding boxes → build two new files

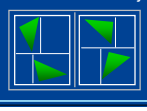


  4. Repeat process recursively with new streams (files)

Index Generation
Streaming Approach

- Offline index generation
  5. If number of triangles small enough → build object kd-tree in-core

  6. Build top-level kd-tree based on object bounding boxes



Index Generation
Streaming Approach

- Optimizations
  - Remove vertex shading data from triangle stream
    - Normals, UV-coordinates, vertex colors, etc.
    - Do sorting only with vertex position data
    - Reconstruct full triangles after sorting
  - Compute better splitting planes
    - Use cost functions to determine plane position e.g., Surface Area Heuristics (SAH) [McDonald et al. 1989]
  - Parallelize sorting
    - Start extra processes for new streams

Part II

Adapting ray tracing to complex models

Out-of-Core Ray Tracing

- Ray tracing capable of handling massive models
  - Multi-level kd-trees as hierarchical spatial index
- E.g., „Boeing 777“ model requires 30 – 40 GB

➔ Out-of-core mechanism needed

  - Build index structures offline on disk
  - Map disk data into 64-bit address space (`mmap()`)

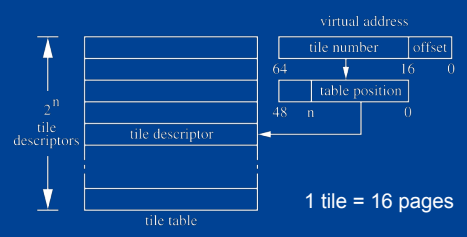
Memory Management
OS Based Memory Mapping

- Advantages of OS based memory mapping
  - Automatic demand paging
  - Address translation and I/O handled by CPU and OS
  - Fine cache granularity (page size 4 KB)
- Problems
  - Access to unavailable data causes page faults
  - Stalling of rendering process inhibits interactivity

➔ Manually check data availability
Detect and prevent page faults using tile table

Memory Management
Tile Table

- Simple hash table to efficiently check tile availability



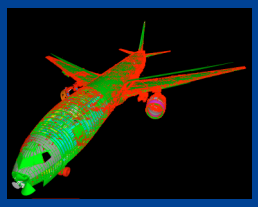
1 tile = 16 pages

Memory Management
Tile Handling

- Tile descriptor
  - Bits 64 – 16 : Tile base address (detect hash collisions)
  - Bit 1 : Referenced bit
  - Bit 0 : Availability bit
- Tiles loaded by *asynchronous* fetcher thread
  - Cache miss: Add tile ID to request queue
- Asynchronous tile eviction
  - Free memory using „Second Chance“ algorithm (`madvise()`)

Bridging Load Time

- What happens if data is not yet in main memory ?



Rays trying to access not loaded data colored red

Fully loaded data - but only for this particular view

Bridging Load Time
Ray Reordering

- Ray reordering [Wald et al. 2002]
  1. Suspend rays that try to access not yet loaded data
  2. Fetch missing data asynchronously
  3. Immediately continue with other ray
  4. Resume stalled rays after data is available

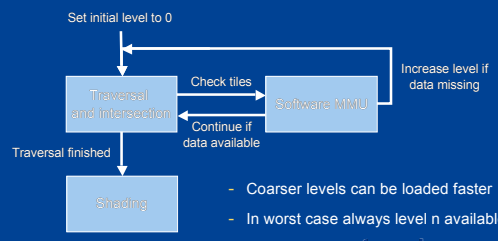
➔ Only possible for smaller models with not drastically changing working sets

Bridging Load Time
Level-Of-Detail

- Use simplified data as replacement
  - Polygonal simplification
    - See e.g., “A Developer’s Survey of Polygonals Simplification Methods” [Luebke 2001]
    - For “soup of triangles” typically use vertex clustering
  - Voxel representation
    - See e.g., “Far Voxels” [Gobetti et al. 2005]
- Generate n+1 model detail levels
  - Level 0: Original un-simplified model
  - Level n: Coarsest simplification → should fit into memory

Bridging Load Time
Level-Of-Detail

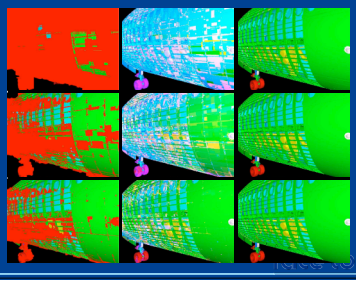
- Switch to simplified representation during loading



- Coarser levels can be loaded faster
- In worst case always level n available

Bridging Load Time
Level-Of-Detail

- Progressive loading (3 levels, 100%, 25%, 5%)

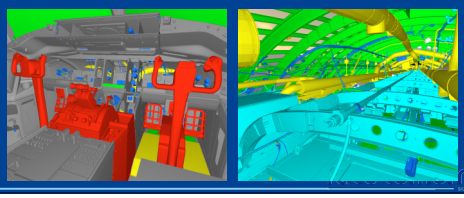


Part III

Photorealistic lighting of highly complex models

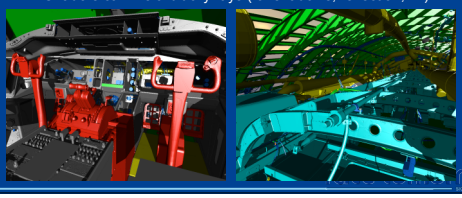
High-Quality Shading
Shadows

- Without shadows and highlights



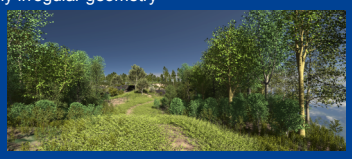
High-Quality Shading
Shadows

- Pixel-accurate shadows and highlights
  - Simple integration into a ray tracer
    - Shader (plug-in) is called when a ray hits a surface
    - Shaders can fire arbitrary rays (for shadows, reflection, …)



Photorealistic Rendering

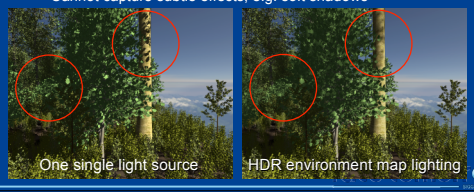
- More complicated example:
  - Realistically structured plant ecosystem
  - Many plants and vegetation layers
  - Highly irregular geometry



➔ Much more difficult than CAD models

Realistic Lighting
Environmental Illumination

- Realistic Illumination of outdoor scenes
  - Depends heavily on environmental illumination
  - One single directional light not sufficient
    - Cannot capture subtle effects, e.g. soft shadows



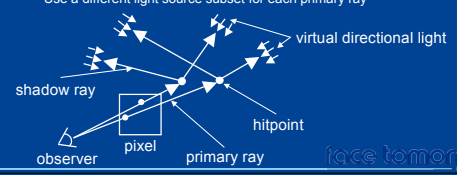
Realistic Lighting
HDRI Approximation

- Pre-computation, e.g. PRT, not practical
  - Scene too complex → memory limitations
  - Difficult to use with instantiation

➔ Approximate HDR environmental illumination
  - Approximate with large number of directional lights
    - Generated from HDR environment maps
      e.g., similar to [Kollig et al. 2003] or [Agarwal et al. 2003]
  - Randomly pick subsets from these virtual lights
    - Use as targets for shadow rays
    - Interleave shadow rays with primary rays

Interleaved Sampling

- Interleaved Sampling [Keller et al. 2001]
  - Combination of geometric and illumination anti-aliasing
    - Split up set of virtual directional lights into subsets
    - Fire a number of primary rays per pixel
    - Use a different light source subset for each primary ray



Interleaved Sampling
Example

- 1 primary sample per pixel
- 1 light sample / hitpoint
  (1 virtual light source)

32 CPUs: 6 fps (640×480 pixels)

- 4 primary samples per pixel
- 4 light samples / hitpoint
  (4 different sets of virtual lights)

32 CPUs: 1 fps (640×480 pixels)

Note: "sample" means sequence of ray segments

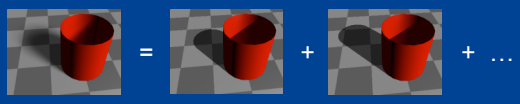
Handling Aliasing

- Brute-force pixel over-sampling
  - Simultaneously remove geometric and illumination aliasing (using interleaved sampling)
  - Trivial implementation
  - Scales better than linear in number of CPUs (→ coherence)
  - But still needs many samples for high-quality images
    - Especially for complicated geometry (e.g., plant leaves)
    - And complex illumination model (e.g., global illumination)

➔ Progressive image enhancement

Progressive Rendering

- Rendering in progressive mode
  - Activated as soon as camera motion stops
  - Successive frames are accumulated
  - Use new random sample values each frame
  - Generates high-quality images in a few seconds



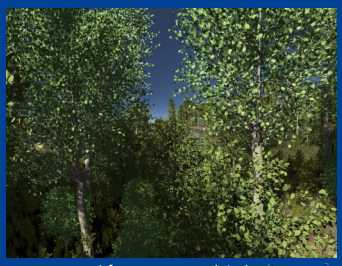
Progressive Rendering
Example

1 primary sample, 2 light samples, 48 CPUs: 2 fps (1270×960 pixels)

No accumulation

Progressive Rendering
Example

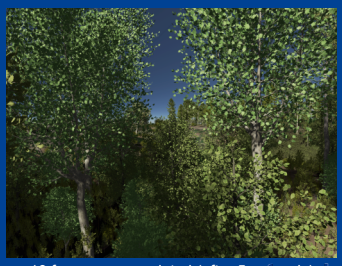
1 primary sample, 2 light samples, 48 CPUs: 2 fps (1270×960 pixels)



4 frames accumulated

Progressive Rendering
Example

1 primary sample, 2 light samples, 48 CPUs: 2 fps (1270×960 pixels)



10 frames accumulated (after 5 seconds)

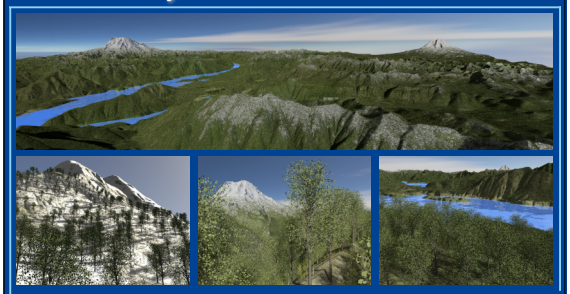
Part IV

Review of hardware-trends for real-time ray tracing

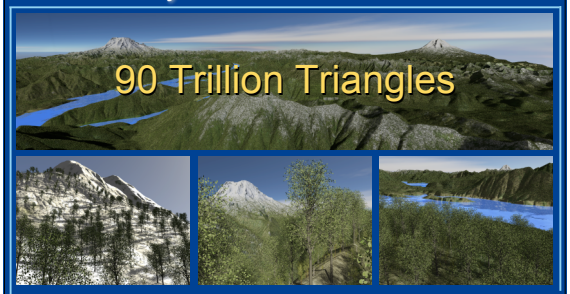
Hardware Trends For Real-time Ray Tracing

Philipp Slusallek
Saarland University, Germany

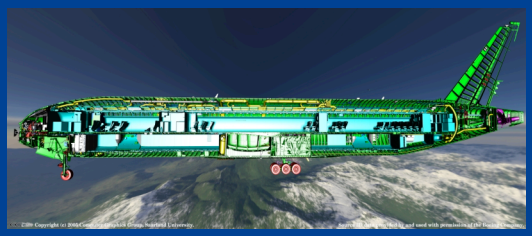
Outdoor Environments with full Sky Illumination



Outdoor Environments with full Sky Illumination



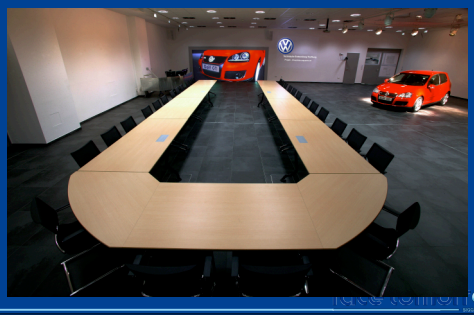
Large Model Visualization at Boeing



CATIA Model of Boeing 777:
350 million triangles, 30 GB on disk, 2-3 fps on Dual-Opteron

VW Visualization Center by inTrace GmbH

VW Visualization Center by inTrace GmbH



VW Visualization Center by inTrace GmbH

Lighting Simulation at EADS

Product Visualization at EADS

Ray Traced Games

Real-time Requirements

- Minimum Number of Rays
  - 1 megapixel screen
  - 30 frames per second
  - 10 rays per pixel (anti-aliasing, lighting, ...)
  - → 300 million rays per second
- But
  - Larger screens (2x), higher frame rate (2x)
  - Complex lighting (10x)
- Promising: Adaptive space-time sampling

Ray Tracing on Multi-Core

- Advantages
  - High-performance implementations are available
  - Highly flexible environment
  - Scales nicely with # of cores (~10 Mrays/s per core)
- Disadvantage
  - Need 30 cores for minimum requirements
- Not for the mass market any time soon

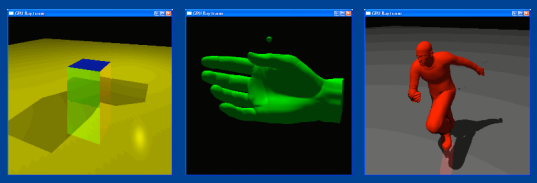
Ray Tracing on Multi-Core

- Advantages
  - High-performance implementations are available
  - Highly flexible environment
  - Scales nicely with # of cores (~10 Mrays/s per core)
- Disadvantage
  - Need 30 cores for minimum requirements
- Not for the mass market any time soon
  - But high-end systems are becoming available
    - Opteron-System (8 CPUs x Quad-Core) → 32 cores

Ray Tracing on GPUs

- Increasingly Implemented as an Add-On
  - Volume rendering by ray casting [Krüger '03]
  - Displacement mapping [Wang '04]
  - Approximate refractions on GPU [Weyman '05]
  - Screen space caustics [Krüger '06]
- Not well supported by GPUs
  - So far, less efficient than CPUs
    - Even though they have higher raw performance

Ray Tracing on GPUs: Performance @ 1024 x1024



Scene	Triangles	ATI x1900
Cube	16	5.0
Hand	17k	5.5
Ben	72k	1.1

Ray Tracing on Cell

- Advantages
  - Already 8 compact but powerful cores (SPUs)
  - Highly efficient SIMD instruction set
  - DMA and full control over caches in LS
  - C/C++ compiler
- Disadvantages
  - Still hard to program, non-optimal compilers
  - Needs another programming approach
    - No good, high-level data parallel languages available
  - Complex and costly memory handling

Ray Tracing on Cell: Performance @ 1024 x1024

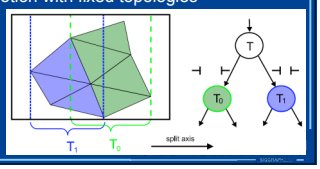
Scene	Triangles	Single-Cell	Dual
ERW6	800	58.1	110.9
Conference	280k	20.0	37.3
Beetle	680k	16.2	30.6

D-RPU Approach

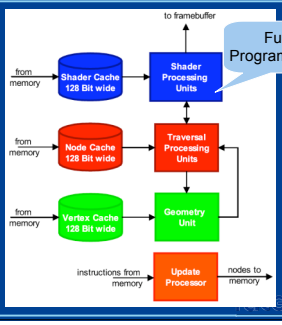
- Shading processor
  - Design similar to fragment processors on GPUs
  - Support for full recursion even with SIMD
  - Highly parallel, highly efficient
- Improved programming model
  - Add highly efficient recursion, conditional branching
  - Add flexible memory access (beyond textures)
- Custom traversal and intersection hardware
  - High-performance kd-tree traversal & triangle intersection

D-RPU: Dynamic Scenes [GH'06]

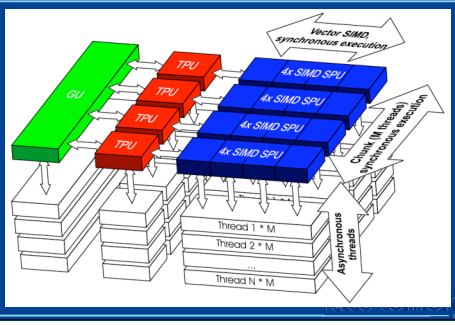
- Bounding KD-Trees (B-KD Trees)
  - Combining the best of two worlds
    - Traversal efficiency of kd-trees
    - Update efficiency of bounding volume hierarchies
  - Efficient for coherent motion with fixed topologies
  - Supports general rays
  - Good for empty space
- Implemented in HW
  - Traversal & update



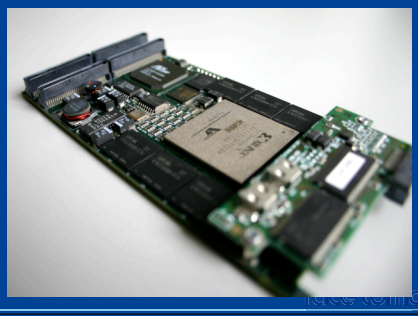
D-RPU: High-Level Architecture



D-RPU: Hardware Architecture

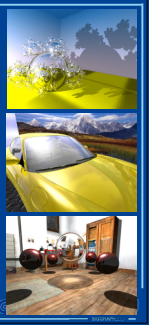


Hardware Implementation



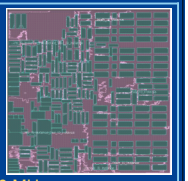
D-RPU Implementation

- Xilinx Virtex-4 LX160
  - 128 MB RAM, .5 GB/s @ 66 MHz
  - 7.5 GFLOP/s @ 24 bit
  - Usage: 99% logic, 60% memory
  - 32 threads per SPU 60% usage
  - Chunk size of 4 95% efficiency
  - 12 kB caches in total 90% hit rate
- Performance
  - 40-70% faster than OpenRT
  - OpenRT on CPU with 40x clock rate
  - ➔ 60x „more efficient“



D-RPU Implementation

- D-RPU ASIC
  - Synthesized from HWML
    - With HW evaluation for clock rate
  - Larger caches (3x 16 KB)
    - 4-way associative
  - 130 nm process from UMC: 49 mm2, 266 MHz
    - 30 GFLOP/s @ 32 bit (post-layout timing)
    - 2.1 GB/s required to external memory



Projections

- ATI R-520: 288 mm² in 90 nm process
- D-RPU-4: 196 mm², 130 nm
  - 120 GFLOP/s @ 266 MHz (constant field scaling)
  - 8.5 GB/s (DDR2 memory?)
- D-RPU-8: 186 mm², 90 nm
  - 361 GFLOP/s @ 400 MHz (constant field scaling)
  - 25.6 GB/s (multi-channel DDR-2 or XDR memory)

Performance @ 1024 x 768
(shadows, full Phong shading, textures)

Scene	triangles	objects	#rays	DRPU FPGA	DRPU ASIC	DRPU4 ASIC	DRPU8 ASIC
Shirley6	0.5k	1	1.5M	4.7 fps	18.8 fps	75.2 fps	225.6 fps
Conference	282k	52	1.5M	1.7 fps	6.7 fps	27.0 fps	81.2 fps
Office	34k	1	1.5M	3.6 fps	14.4 fps	57.6 fps	172.8 fps
Mafia Room	15k	1	1.5M	2.8 fps	11.2 fps	44.8 fps	134.4 fps
Mafia Spheres	20k	6	1.6M	1.8 fps	7.2 fps	28.8 fps	86.4 fps
Hand	17k	2	1.3M	5.0 fps	20.0 fps	80.0 fps	240.0 fps
Skeleton	16k	2	1.3M	5.9 fps	23.6 fps	94.4 fps	283.2 fps
Helix	78k	2	1.5M	3.5 fps	14.0 fps	56.0 fps	168.0 fps
Gael	52k	1	1.5M	1.9 fps	7.6 fps	30.4 fps	91.2 fps
DynGael	85k	4	1.5M	2.0 fps	8.0 fps	32.0 fps	96.0 fps

Outlook: Hardware for Ray Tracing

- Symmetric & Asymmetric Multi-Core CPUs
  - Current: ~10 Mrays/s (per core)
  - Future: many cores per chip, SHM
- High Performance Parallel GPUs
  - Not competitive (yet?), limited programming model
- Custom Ray Tracing Hardware
  - Current: 5-9 Mrays/s (FPGA, 66 MHz)
  - Future: >300 Mrays/s (ASIC, 285 MHz)

Interested?
Questions?
informatik
saarland.
Informatik Saarland http://www.informatik-saarland.de
Computergraphik http://graphics.cs.uni-sb.de
Ray Tracing http://www.OpenRT.de
Direct Email slusallek@cs.uni-sb.de
inTrace GmbH http://www.inTrace.com
info@inTrace.com
trace tomorrow

11 Putting Theory into Practice

Siggraph 2007

State-of-the-art inMassive Model Visualization

Putting Theory into Practice

(Inigo Quilez, VRcontext)

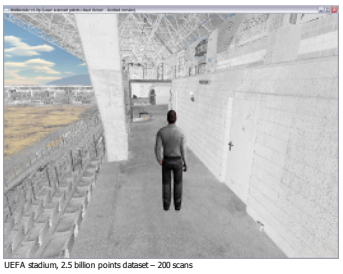
August 2007

Perspective

VR support, camera collision detection, real-time monitoring and controlling, database links, scene editing – scripting, super fast model loading, moving object, streaming

Lasser scanning - pointclouds

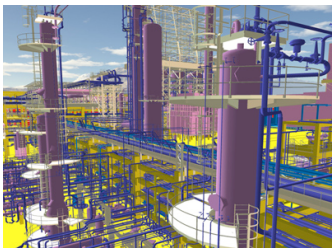
- About 100 billion points
- With collision detection

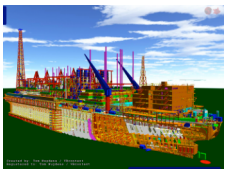


UEFA stadium, 2.5 billion points dataset – 200 scans

CAD File

- An FPSO with more than 1 billion polygons
- A refinery with 4 million objects (>1 billion polygons)
- A nuclear reactor room with more than 1 million primitives





Problems

- 1. Most real data is bad formed
- 2. About data set size
- 3. About quality
- 4. Other practial problems
- 5. Somehow incorrect assumtions on the research

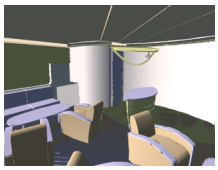
Problems

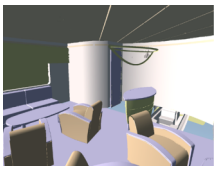
- 1. Most real data is bad formed
- 2. About data set size
- 3. About quality
- 4. Other practial problems
- 5. Dangerous assumptions on academic research

1. Most real data is bad formed

- CAD models:
  - T junctions
  - incorrectly oriented geometry
  - not orientable geometry
  - duplicated geometry
  - clashes
- Laser scanned point clouds
  - noise
- GIS
  - less than optimal triangulation

1. Most real data is bad formed





- Non orientable geometry. If back-face culling is disabled, lot of coplanar and inversely oriented face cause disturbing z fighting. If back-face culling is enabled, this artifacts dissapear but objects not all objects are rendered anymore.
- Some objects are two sided. Flip the normals per pixel or per vertex is mathematically innacurate, and indeed causes disturbing artifacts (for example in the borders of the objects). The only way to fix that is per face, but this approach doesn't fit in current GPU pipelines.

1. Most real data is bad formed

- Duplicated geometry is common, for many reasons.
- For example, often a visualization element has a cloned version for database support.
- Whatever the reason, the fact is that Z fighting is incredibly disturbing for the user.
- And shadows make this even worse!

Problems

- 1. Most real data is bad formed
- 2. About data set size
- 3. About quality
  massive model rendering techniques are not design for quality
- 4. Other practial problems
- 5. Somehow incorrect assumtions on the research

2. Data set size

- Gamming technology is unsuitable for massive models. For example
  - Shadows: shadow volumes are impractical in massive poly-soups. Shadow maps are better, but it's very hard to overcome resolution problems (note that we often have milimeter details in models of 4 kilometers). Vertex based no linear rasterization are also not affordable yet.
  - Advanced (global) lighting. Even precomputed lighting is non trivial: several gigabytes of textures, need for streaming + LOD, etc. We cannot afford neither to do it per vertex (we already have too polygons and vertices).
- Massive models often have also a massive amount of materials. This explosion can make difficult the clustering of geometry in cells (these cells are used atomic objects for visibility culling, rendering and so on, meaning one single VBO/IBO for the complete cell for example). Tricks like « virtual » materials don't work completely fine.

2. Data set size (demo)

2. Data set size (demo)

- Rasterization cannot do this yet with massive models.
- Even if we solve the polycount scalability problem (with the techniques explained in these course), we still have the rendering cost that is linear with the amount of « effects » we want to add, ie, shadows need redrawing the geometry, adding a reflection maps means a thirth rendering pass of the model…
- Regarding memory, it's not only about out of core geometry, but also out of core textures.

Problems

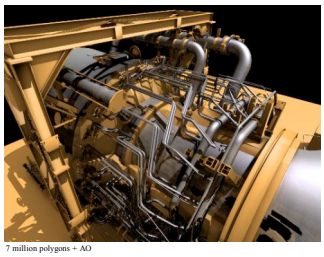
- 1. Most real data is bad formed
- 2. About data set size
- 3. About quality
- 4. Other practial problems
- 5. Somehow incorrect assumtions on the research

3. About quality

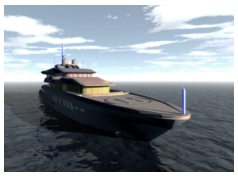
- Rendering quality has been a secondary goal in most massive model rendering techniques so far, of course. However customers demand something else than Phong shading. Shadows are a must, some kind of advanced shading (reflections, AO, GI) are very desirable.
- GPU approaches rely on huge amount of textures to handle this, plus multi-pass rendering. But multi-pass rendering conflicts with the presumed frame-to-frame coherence in many techniques.
- So, …

3. Raytracing – the solution for quality…

- It scales well with polycount
- Shading effort is the minimun possible

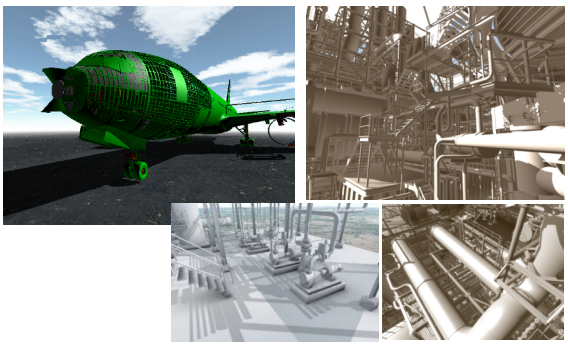


7 million polygons + AO



5 million polygons

… with massive models. Or may be not?



… with massive models. Or may be not?

- We still don't have the brute force power and/or the algorithms to allow for real time quality raytracing. Impressive advances has been made but we still need 8 cores to have 25 frames per second on a massive model.
- We can only afford simple shading in most cases, nothing to do with the high quality we want. (we cannot sell a technologie that makes uglier images than the previous ones).
- Fast raytracing relies too much on coherent rays.
- As a fact, even raytraced CG movies most hq-effects are still facked with maps (reflections, lightmaps, normalmaps…).
- In 2007, rasterization still more than competitive solution (it will take longer than expected to see the change)
- However…. there is no apparent way we will achieve the high quality in massive models with rasterization in the short term neither. Raytracing is the future… the question is when will that future be.

Problems

- 1. Most real data is bad formed
- 2. About data set size
- 3. About quality
- 4. Other practial problems
- 5. Dangerous assumtions on the research

4. Other problems

- SDKs
  - how to hide the implementation, and still keep optimal performance
- Moving objects
  - the evil of most algorithms
- Document/export management
  - delta exporting, incremental precomputations affects data structures
  - build acceleration structures on demand, at runtime (render time)
- File formats
  - too many standards, what means no standard
- Virtual Reality setup
- Marketing, the money

Problems

- 1. Most real data is bad formed
- 2. About data set size
- 3. About quality
- 4. Other practial problems
- 5. Dangerous incorrect assumtions on the research

5. Dangerous assumtions

- Frame to frame coherence
- Out of core?
- Density of models
- Occlusion degree

5.1. Frame to frame coherence

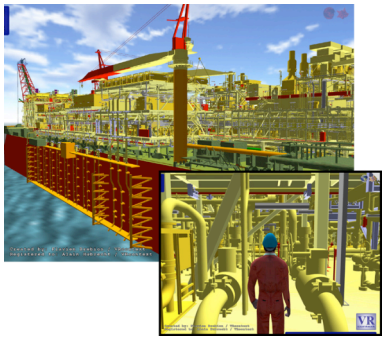
- Hardware occlussion queries and similar techniques, requiere frame to frame coherence on the camera movement to hide latencies, or improve cache performance.
- Frame to frame coherence may not exist in a client-rendering server cofiguration where a single machine renders frames for different concurrent users, ech with a different camera location.
- Multipipe configurations (several graphics cards rendering at the same the same model, common in virtual reality centers) are also not optimal. Each GPU renders from a very different frustum (frustums can even have opposite directions), and since all of them share the same system memory area, it can create a caching problem.
- Many users are not used at all to realtime walkthrus, and still prefer to set the camera position and orientation by clicking in a top view map of the model. This never creates smooths paths along the model, obviously.

5.3. Density of the models

- We often find what we call « low density massive models », and we have some troubles with them.
- It is not that easy to find a definition, but think on the amount of polygons per unit cube volume, the dynamic range of the dataset or even the fractal dimension of the model surface. It related to all that.
- Examples:
  - The Boeing 777 or the Powerplant are medium density models, and are the most common models used in research.
  - Happy Buddha,the Dragon and other scanned models are high density models, non of the techniques used to render those apply to our models.
- Most literature for massive models is about medium density models.

5.3. Density of the models

- High Density
  - D > 10k t/m³
  - DR ~ 1:10³
- Medium Density
  - D ~ 100-1k t/m³
  - DR ~1:10⁵
- Low Density
  - D < 1tri/m³
  - DR >1:10⁶

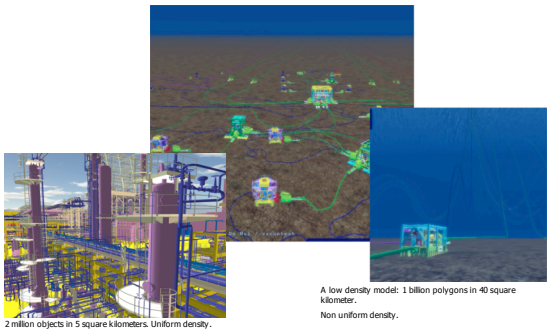


A typical medium density model (600 million polygons)

5.3. Density of the models

- Problems arrive with low density massive models.
- Models that are still massive (like one billion polygons), with very high dynamic ranges (features of half centimeter on a surface of 20x20 kilometers).
- Common to overflows the capacity of a floating point number.
- Spliting the model in cells with local frames works, but it's more expensive (think on a raytracer).
- These models can also suffer from other problems like low self occlussion degrees.

Low Density models



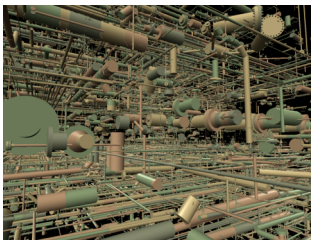
2 million objects in 5 square kilometers. Uniform density.

A low density model: 1 billion polygons in 40 square kilometer.

Non uniform density.

5.4. Occlusion degree

- There is not much occlusion in some models (because they are low density, or because user selects a subset of the model, like « water pipes only ».
- This makes them very sensitive to agressive geometry rejection.



Incomplete model, still 120 million poligons with no much occlusion

- Cannot relay on occlusion detection as major speed gain factor anymore.
- But still, we need to render fast.
- Raytracing handels this much better.

Conclusion

- Many models, not possible to tune for just one type
- Fast research often means simple solutions
- It's not clear what's the best technology, so the product has as much software engineering as pure visualization work
- Still need lot of work on (in order of importance)
  - low density models
  - dynamic scenes
  - high quality shading