High-Performance Visualization of Large and Complex Scientific Datasets

Course Notes for Tutorial M9/SC 2002
Baltimore, Maryland
November 2002

Organizers:

Dirk Bartz	University of Tübingen, Germany
Cláudio T. Silva	Oregon Health & Science University

Speakers:

Dirk Bartz	University of Tübingen, Germany
Peter Lindstrom	Lawrence Livermore National Laboratory
James T. Klosowski	IBM T.J. Watson Research Center
Will Schroeder	Kitware
Cláudio T. Silva	Oregon Health & Science University

Abstract

The goal of the tutorial is to introduce students, academics, and professionals to techniques for the visualization of very large and complex datasets, in particular those datasets which are too large to fit in main memory.

About 40% of the material are related to algorithms for large-data visualization, including external memory algorithms. About 40% of the material relates to parallel rendering and large displays. The other 20% of the time will be spent presenting the VTK system that implements several of the most useful visualization tools, and has been evolving towards being able to perform very large datasets.

Schedule (tentative)

08:30–08:45	Introduction to the course	Silva
08:45–10:00	Parallel Rendering Algorithms	Bartz
10:00–10:30	Break	
10:30–11:30	Building and Driving High-Resolution Displays	Klosowski
11:30–12:00	Out-Of-Core Scientific Visualization	Silva
12:00–01:30	Break	
01:30–02:15	Out-Of-Core Scientific Visualization (cont.)	Silva
02:15–03:00	Out-Of-Core Surface Simplification	Lindstrom
03:00–03:30	Break	
03:30–03:45	Out-Of-Core Surface Simplification (cont.)	Lindstrom
03:45–04:45	Parallel Processing With Vtk	Schroeder
04:45–05:00	Final Remarks & Questions	All

Speaker Biographies

Cláudio T. Silva
OGI/CSE
20000 N.W. Walker Road
Bearverton, OR 97006
E-mail: csilva@cse.ogi.edu
Cláudio T. Silva is an Associate Professor in the Department of Computer Science & Engineering of the OGI School of Science and Engineering at Oregon Health & Science University. Previously, he was a Principal Member of Technical Staff at AT&T Labs-Research.
His current research is on architectures and algorithms for building scalable displays, rendering techniques for large datasets, 3D scanning, and algorithms for graphics hardware. Before joining AT&T, Claudio was a Research Staff Member at IBM T. J. Watson Research Center. Claudio has a Bachelor's degree in mathematics from the Federal University of Ceara (Brazil), and MS and PhD degrees in computer science from the State University of New York at Stony Brook. While a student, and later as an NSF post-doc, he worked at Sandia National Labs, where he developed large-scale scientific visualization algorithms and tools for handling massive datasets. His main research interests are in graphics, visualization, applied computational geometry, and high-performance computing. Claudio has published over 40 papers in international conferences and journals, and presented courses at ACM SIGGRAPH, Eurographics, and IEEE Visualization conferences. He is a member of the ACM, Eurographics, and IEEE. He is on the editorial board of the IEEE Transactions on Visualization and Computer Graphics.

Dirk Bartz
Univ. of Tuebingen
WSI/GRIS
Auf der Morgenstelle 10/C9
D72076 Tuebingen, Germany
bartz@gris.uni-tuebinigen.de
Dirk Bartz is currently member of the research staff of the Computer Graphics Laboratory (GRIS) at the Computer Science department of the University of Tuebingen. His recent works covers interactive virtual medicine and thread-parallel visualization of large regular datasets. In 1998, he was co-chair of the "9th Eurographics Workshop on Visualization in Scientific Computing", and he is editor of the respective Springer book. Dirk studied computer science and medicine at the University of Erlangen-Nuremberg and the SUNY at Stony Brook. He received a computer science and medicine at the University of Erlangen-Nuremberg and the SUNY at Stony Brook. He received a Diploma (M.S.) in computer science from the University of Erlangen-Nuremberg. His main research interests are in visualization of large datasets, occlusion culling, scientific visualization, parallel computing, virtual reality, and virtual medicine.

James T. Klosowski
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
jklosow@us.ibm.com
James T. Klosowski is a Research Staff Member at the IBM Thomas J. Watson Research Center. His main research interests are in computer graphics, visualization and applied computational geometry. Klosowski received a BS in computer science and mathematics from Fairfield University in 1992, and an MS and PhD in applied mathematics from the State University of New York at Stony Brook in 1994 and 1998, respectively. His research interests in computer graphics include interactive visualization of large datasets, collision detection, volume rendering, and adaptive network graphics. Recently, his research has focused on visibility culling, simplification of complex geometric models, and parallel rendering of distributed data.

Peter Lindstrom
Lawrence Livermore National Laboratory
7000 East Avenue, L-560
Livermore, CA 94551
E-mail: pl@llnl.gov
Peter Lindstrom is a Computer Scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory (LLNL). His current research is focused on surface simplification and compression, view-dependent rendering and other level-of-detail techniques, and geometric and multiresolution modeling. As a member of the ASCI VIEWS visualization team at LLNL, he works on out-of-core methods for managing and visualizing terascale geometric data sets generated in large scientific simulations. Peter joined LLNL in 2000 after receiving his PhD in computer science from the Georgia Institute of Technology, and graduated with BS degrees in computer science, mathematics, and physics from Elon University, North Carolina, in 1994. Peter has published and presented papers at ACM SIGGRAPH, ACM TOG, IEEE Visualization, and IEEE TVCG, among other computer graphics conferences and journals. He is a member of ACM and IEEE.

William J. Schroeder
Kitware, Inc.
469 Clifton Corporate Parkway
Clifton Park, New York 12065
will.schroeder@kitware.com
William Schroeder is President and co-founder of Kitware, Inc. Kitware was incorporated in 1998 to support VTK, and build commercial applications with this open-source software foundation. Prior to his current position, Dr. Schroeder was computational scientist as the GE Corporate R&D Center where he developed software tools for mechanical system, medical, and computational visualization. Dr. Schroeder graduated in 1983 with a M.S. in applied mathematics, and in 1991

with a Ph.D. in mathematics from Rensselaer Polytechnic Institute. His Ph.D. was obtained part-time over an eleven year period while he worked full time at GE. Dr. Schroeder graduated summa cum laude from the University of Maryland in 1980 as a mechanical engineer. Dr. Schroeder continues to be active in the research community presenting papers, teaching courses, and participating on panel discussions at such conferences as Siggraph and IEEE Visualization. Dr. Schroeder has also been an invited speaker at conferences such as Pacific Graphics, the CINECA (Italy) Supercomputing Series, Visualization Development Environments 2000, and AT&T Visualization Days.

Contents

- Bartz
  - “Excerpts from the Course Notes of the tutorial on Rendering and Visualization in Parallel Environments”, D. Bartz, C. Silva, and B.-O. Schneider, 2001.
- Klosowski
  - “Deep View: High-Resolution Reality,” J. T. Klosowski, P. D. Kirchner, J. Valuyeva, G. Abram, C. J. Morris, R. H. Wolfe, T. Jackman, *IEEE Computer Graphics & Applications*, 22(3):12-15, 2002.
  - “Early Experiences and Challenges in Building and Using A Scalable Display Wall System,” Kai Li, Han Chen, Yuqun Chen, Douglas W. Clark, Perry Cook, Stefanos Damianakis, Georg Essl, Adam Finkelstein, Thomas Funkhouser, Allison Klein, Zhiyan Liu, Emil Praun, Rudrajit Samanta, Ben Shedd, Jaswinder Pal Singh, George Tzanetakis and Jiannan Zheng, *IEEE Computer Graphics and Applications*, 20(4):29-37, 2000.
  - “Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters,” Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter Kirchner, Jim Klosowski, *ACM Transactions on Graphics*, 21(3):693-702, 2002.
- Lindstrom
  - “A memory insensitive technique for large model simplification,” P. Lindstrom, C. Silva, *IEEE Visualization 2001*, pp. 121-126, 2001.
  - “External Memory Management and Simplification of Huge Meshes,” P. Cignoni, C. Rocchini, C. Montani, R. Scopigno, *IEEE Transactions on Visualization and Computer Graphics*, to appear.
- Schroeder
  - “A Multi-Threaded Streaming Pipeline Architecture for Large Structured Data Sets,” Charles Law, Kenneth M. Martin, William J. Schroeder, Joshua Temkin, *IEEE Visualization ’99*, pp. 225-232, 1999.
  - “Large-Scale Data Visualization Using Parallel Data Streaming,” James Ahrens, Kristi Brislawn, Ken Martin, Berk Geveci, C. Charles Law, Michael Papka, *IEEE Computer Graphics & Applications*, 21(4):34-41, 2001.

- Silva
  - “Interactive Out-Of-Core Isosurface Extraction,” Y.-J. Chiang, C. Silva, and W. Schroeder, *IEEE Visualization 1998*, pp. 167-174, 1998.
  - “Out-Of-Core Rendering of Large Unstructured Grids,” R. Farias and C. Silva, *IEEE Computer Graphics and Applications*, 21(4):42-50, 2001
  - “Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics,” C. Silva, Y.-J. Chiang, J. El-Sana, and P. Lindstrom, manuscript, 2002.

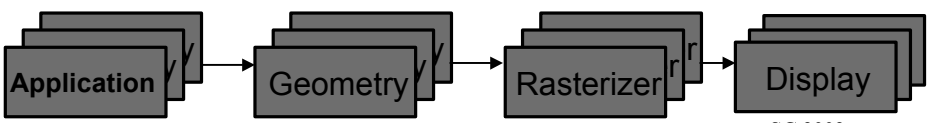
Parallel Graphics Concepts

Dirk Bartz
University of Tübingen, Germany
bartz@gris.uni-tuebingen.de

(Based in part on presentations by
Bengt-Olaf Schneider, NVIDIA and Claudio Silva)

The Problem

- **Render a computer graphics scene using multiple processors and/or threads**
- **Parallelization possible in every stage and overlapping execution of stages**
- **Each stage maybe implemented on the same or on separate processors**



Outline

- The Problem
- General Concepts
- Classification
- Load Balancing
- Practical Issues

Polygon oriented

- Parallel Volume Rendering
- Parallel Hierarchy Generation

Parallel Rendering Scenarios

- **Rendering**
  - Still-image rendering (single-frame)
  - Animation (multiple-frames)
  - Rendering algorithm and rendering quality
- **Parallel Architectures**
  - Tightly-coupled SMP (SIMD less often)
  - Loosely-coupled Clusters
  - Graphics hardware or software renderers

MIMD, SIMD, Pipelining (1)

Multiple-Instruction, Multiple-Data (MIMD)

- High level of independence between processors
- Requires synchronization between processors
- Best for task requiring many data-dependent branches
- Used for geometry processing

MIMD, SIMD, Pipelining (2)

Single-Instruction, Multiple-Data (SIMD)

- Tight coupling between processors
- Implicit synchronization, good scalability
- Best for tasks requiring same operations for many data
- Used for rasterization and fragment processing

MIMD, SIMD, Pipelining (3)

Pipelining a.k.a. Functional Parallelization

- Natural for many algorithms, i.e. graphics pipeline
- Limited scalability and difficult to load balance
- Some operations require pipeline flush

Outline

- **The Problem**
- **General Concepts**
- **Classification**
- **Load Balancing**
- **Practical Issues**
- **Parallel Volume Rendering**
- **Parallel Hierarchy Generation**

Classification of Parallel Renderers

Partitioning

- Object (MIMD)
- Image (SIMD)
- Temporal (pipelining)

Sorting

- First
- Middle
- Last

Object vs. Image Partitioning

- **Object-space partitioning**
  - Most often used for geometry processing
  - Each task handles part of the scene, i.e. high-level objects, triangles etc.
- **Image-space partitioning**
  - Most often used for rasterization
  - Each task processes parts of the image, i.e. pixels, scanlines, tiles, etc.

Temporal Partitioning (1)

- **Popular choice for rendering of animations**
- **Process**
  - Broadcast data to all processor
  - Assign frames (i.e. viewpoint etc.) to n processors
  - Render n frames
  - Display or store n frames

Temporal Partitioning (2)

- **Considered to be embarrassingly parallel**
  - Rendering performance could scale linearly with number of processors
  - But: Data distribution and collection (communication), forms serial overhead and may overshadow computation and rendering ...
    → no speedup & bad scalability

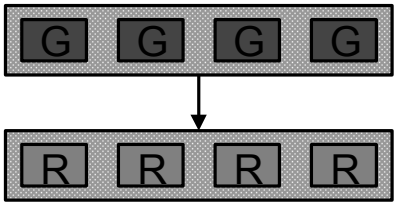
Classification by Sorting (1)

- **Typical parallel polygon rendering system**
  - Geometry processing performed in object space, i.e. each processor handles a subset of the scene
  - Rasterization done in image space, i.e. each processor handles a subset of the screen

  ➡ Objects must be sorted

Classification by Sorting (2)

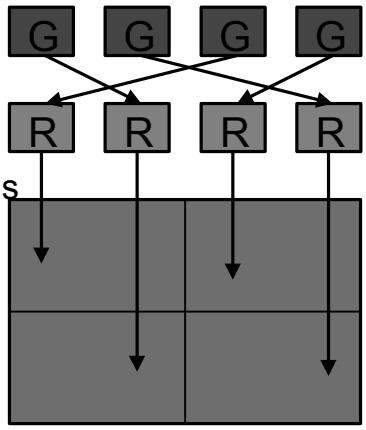
- **From object space into image space**
- **Classification by where the sorting occurs.**



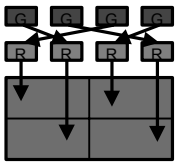
Sort-Middle Rendering (1)

Operation:

- Round-robin assignment of objects to geometry proc's
- Sort objects to screen regions
- Rasterizers responsible for screen regions



Sort-Middle Rendering (2)

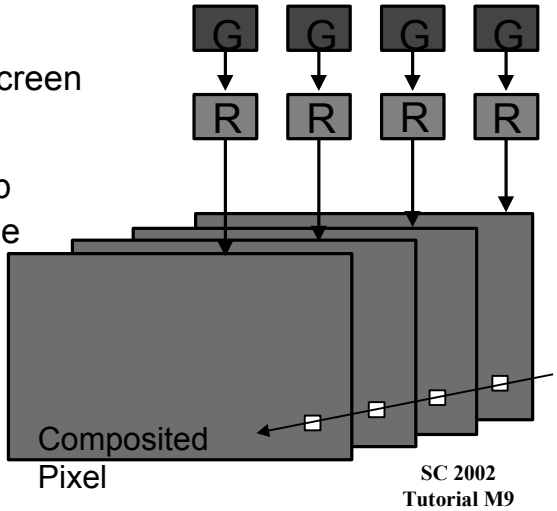


- **Properties**
  - Good load-balancing during geometry processing (clipping!)
  - Possible load-balancing problems in rasterizer
- **Most popular scheme**

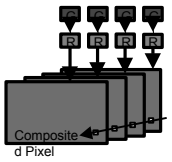
Sort-Last Rendering (1)

Operation

- GR-pair for full screen for all obj's
- Compositing step determines visible objects / pixels



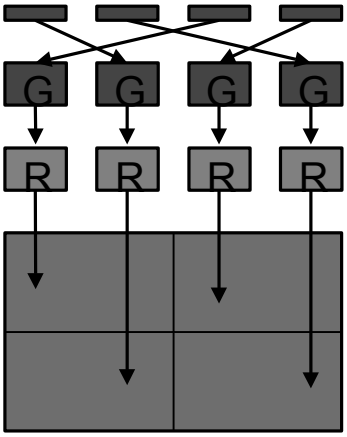
Sort-Last Rendering (2)



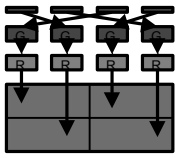
- **Properties**
  - Generally good load balancing (clipping, object size in screen space !)
  - Requires large bandwidth
  - No temporal order, transparency and anti-aliasing hard
- **PixelFlow**

Sort-First Rendering (1)

Quick mapping to screen regions and sorting



Sort-First Rendering (2)



- **Properties:**
  - Less sorting for highly tessellated objects
  - Potential for load-imbalance due to uneven spatial distribution of objects
- **Ie., Princeton Display WALL, WireGL**

Outline

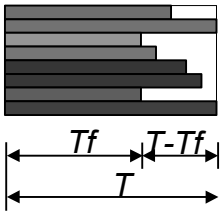
- The Problem
- General Concepts
- Classification
- Load Balancing
- Practical Issues
- Parallel Volume Rendering
- Parallel Hierarchy Generation

Load Balancing

- Objective is to ...
  - distribute work evenly among all processors
  - have all processors finish their work at the same time
- (One) Definition of load balance:

$$LB = \frac{Tf}{T}$$

  - ***LB*** **Load balance**
  - ***Tf*** **Time fastest processor finishes**
  - ***T*** **Total processing time**



Load Balancing Problems (1)

Geometry Processing

- Clipping: Trivial accept/reject simpler than actual clipping
- Vertex Count: Polygons have different number of vertices (e.g. clipping)
- Tessellation: Higher order primitives may be tessellated into different number of basic primitives (triangles)
- Rendering parameters: Number of lights, shading type, texturing algorithm etc.

Load Balancing Problems (2)

Rasterization

- Spatial distribution: Objects tend to be clustered in certain screen areas
- Primitive size: Rasterization time is proportional to primitive size
- Multipass: Some pixels require multipass rendering

Load Balancing Issues (1)

Terminology

- Task: Basic unit of work that can be assigned to a processor, e.g. objects, primitives, scanlines, pixels, etc.
- Granularity: Minimum number of tasks that are assigned to a processor,
  e.g. 10 scanlines or 128x128 pixel regions.

Load Balancing Issues (2)

Terminology, cont'

- Coherence:
  - Neighboring elements (in space or time) are similar, e.g. frames, scanline, pixels
  - Exploited to speed up rendering calculations, ie. for rasterization
  - Parallelization may destroy / hide coherence for a given processor

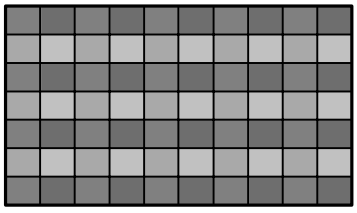
Load Balancing Strategies

- **Static**: Fixed assignment of tasks to processors
- **Dynamic**: On-the-fly assignment of tasks to processors
- **Adaptive**: Assign tasks such that all processors have approximately the same load

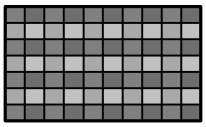
Static Load Balancing (1)

All tasks assigned before start of rendering, ie.

- Round-robin object assignment in sort-middle architectures
- Assignment of screen regions to rasterizers (SGI RE/IR, Pixel planes 4)



Static Load Balancing (2)



Relies on assumptions about heuristics of the model to achieve load balancing, e.g.

- Most objects requires same amount of work
- Interleaving of pixels will give each processor equal share of busy and less busy screen regions
  But: Reduces coherence between pixels within a processor
- All frames of an animation incur approximately same workload

Dynamic Load Balancing (1)

Task are assigned on demand, i.e. the next task goes to the first available processor:

- Assume that there are more tasks than processors
  Granularity Ratio = #tasks / #processors > 1
- Upper bound for load imbalance is difference between largest and smallest task
- Simple optimization: (if known) assign largest tasks first

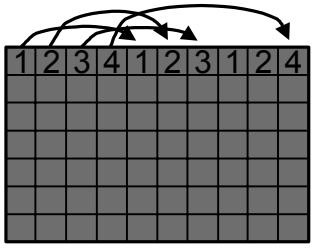
Dynamic Load Balancing (2)

Task/processor assignment not known a priori:

- Maintains a task list that is depleted by processors
- Requires dynamic (sic) distribution of tasks during runtime
- Tasks may not complete in same order as issued; some APIs require temporal ordering, e.g. OpenGL !

Example: Pixel-Planes 5

- Rasterization modules for 128 x 128 pixel regions
  - 80 regions for 1280 x 1024 display
  - Several modules in a system
- Idle rasterizers process next unprocessed region
  - More efficient than Pixel-planes 4
  - Scalability for cost, performance and display size



Adaptive Load Balancing (1)

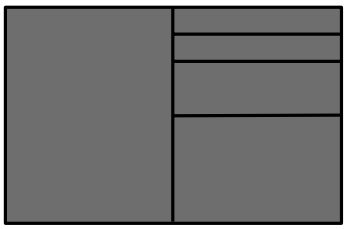
- Create tasks which will require (approximately) the same amount of processing time
- **Static adaptive** load balancing
  - **Predictive**: Estimate the processing time for each task
  - **Reactive**: Deduce processing time from previous frame
  - Requires separate step to determine task assignments

Adaptive Load Balancing (2)

- **Dynamic adaptive** load balancing
  - Monitor workload of processors
  - Reassign tasks from busy processors to idle processors
  - Requires concurrent monitoring process

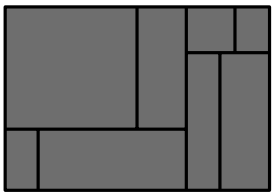
Example: Roble's Method

- Determine number of primitives per region
- Combine low-load regions
- Split high-load regions in half
- No control over location of split and hence chance for low/no effectiveness



Example: Whelan's Method

- Count number of centroids per region
- Split regions using distribution of centroids
  - Median-cut algorithm to subdivide such that both new regions contain same number of centroids
- Large effort for sorting primitives
- No accounting for primitive size

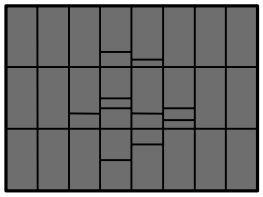


Example: Whitman's Method (1992)

- Tally primitives overlapping each region
- Build quadtree of all regions, assigning number overlapping primitives at each quadtree node
- Top-down subdivision of quadtree for nodes with highest primitive count
- May still leave unbalanced work distribution
  - Increased granularity ratio
- Big primitives are counted multiple times: Overestimation of work

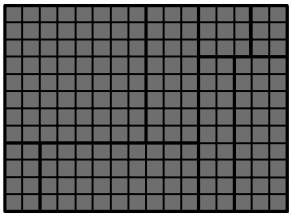
Example: Whitman's Method (1994)

- Initial subdivision into tiles for given granularity ratio (here 2)
- During rendering, processors who finish early "steal" work from busy processors by splitting work region
  - An idle processor finds the processor with most work left
  - Split only if remaining work exceeds a threshold



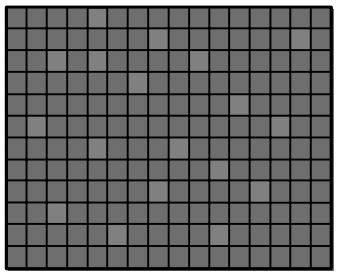
Example: Mueller's Method: MAHD

- Mesh-based adaptive hierarchical decomposition
  - Based on small screen cells, ie. a fine, regular mesh
- Count primitives overlapping each cell in inverse proportion to their size
  - Avoid accounting problems like in Whitman's method
  - Heuristic to balance constant geometry cost and size-dependent rasterization cost (experimentally justified)
- Build a summed-area table (SAT)
- Subdivide screen into regions along cell boundaries
  - similar to median-cut algorithm but cheaper because of SAT

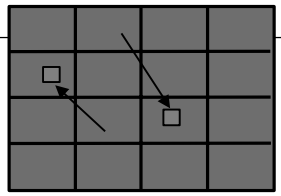


Example: Ellsworth's Method

- Reactive method, load balancing between frames
- Tally number of primitives overlapping each cell
- Estimate processing time for each cell
- Greedy, multiple-bin-packing algorithm to assign regions to processors
  - Sort regions by descending polygon counts
  - Assign next region to processor with lightest workload



Example: Princeton Display Wall (1)



Multi-projector system

- Each projector is driven by a dedicated computer
- Computers are connected through high-speed network
- Data replication through-out system (2000)
- Later partial replication: k-way replication (2001)

Example: Princeton Display Wall (2)

Object/pixel based partition for every view

- Hybrid sort first/last system
- Recursive binary partition (kd-tree)
- Goal to reduce load inbalance and com' overhead

Example: Princeton Display Wall (3)

Recursive binary partition

- Sweep planes assign primitives to partitions to render servers (sort first)
- Are sensitive to 2D Bounding Box size
- Partition overlaps are transmitted to participating render servers (which renders other partition) (sort last)
- Requires potentially slow framebuffer readback

Example: OpenSG (1)

- **Open source system: http://www.opensg.org**
- **Scene Graph is replicated through out parallel system**
- **Local copies of data fields**
- **Data modification at synchronization points**

Example: OpenSG (2)

- **Will have support for Volume Rendering (2D/3D texture based)**
- **Right now, only 2D slicing**
- **Current cluster model:**
  - Every node (PC) drives one display
  - Load balancing implemented,
    but currently no image composition available

Others

- **WireGL/Chromium: Jim will talk about it in the next session**
- **Various solutions of the National Labs**
- **....**

Outline

- The Problem
- General Concepts
- Classification
- Load Balancing
- ➔ Practical Issues
- Parallel Volume Rendering
- Parallel Hierarchy Generation

Temporal Ordering

- Often lost/difficult with parallel strategies
- Required by some APIs, ie. OpenGL
- Important for algorithms that rely on the order in which objects are drawn onto the screen
  - Painter's algorithm
  - Multi-pass algorithms, ie. transparency, overlays, solid-modeling, priority algorithm
- Possible solutions
  - Sequence numbers (time stamps) enforce strict ordering
  - Barriers to ensure ordering between groups of objects (ie., WireGL)

Graphics Hardware Concepts

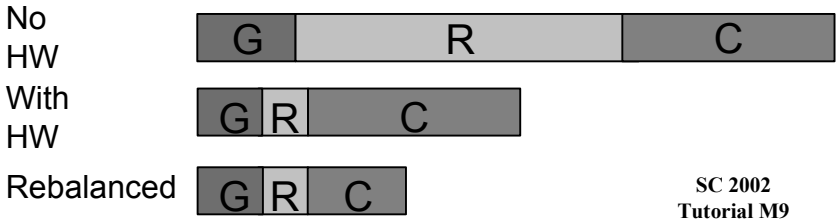
- Graphics hardware accelerators are very useful for single-processor rendering
- Parallel Polygon Rendering Hardware
  - Typically sort-middle architecture
  - By now, complete graphics pipeline in hardware
  - Most systems interleave pixels, scanlines, or stamps
  - Some systems overlay a coarser tiling scheme for virtual rasterizers, ie. Pixel-planes 5, Talisman

Exploiting Graphics Hardware

- Multiple CPUs and/or multiple graphics adapters are a challenge
  - SMP geometry pipelines are difficult for many APIs
  - In clusters, communication (in particular compositing) often overshadows computation and rendering
  - Communication overhead is often overlooked or excluded
  - Even "embarrassingly parallel" approaches (frame-parallel) are not trivially implemented
    - Require high-speed networks and high-speed disks
    - Poor scaling and chance for deceleration
    - Solution: Increase CPU load to reduce communication

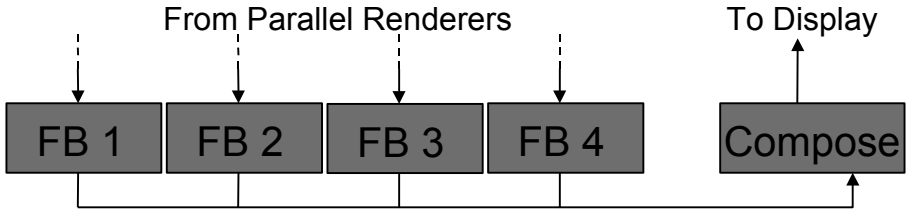
Communication Bottleneck

- Hardware usually shortens rendering time
  - Shift balance between computation & communication
- Rebalance by reducing communication overhead
  - Use more CPU, ie. compression, selective updates, ...
  - Use less bandwidth, ie. distributed frame buffers
  - Overlap communication and computation



Distributed Frame Buffer

- Avoid communication of final pixel values to a central storage or display
- Instead use dedicated compositing hardware to combine partial image
  - ie. Wei et al. PRS95, IBM Yotta, Compaq's Serpia, Lightning2



Summary

- Parallel polygon rendering classifications
  - Sort-first/middle/last
- Key problems in parallel polygon rendering
  - Load balancing
  - Communication overhead (in particular compositing
- Load Balancing
  - Required due to uneven work distribution
  - Static, dynamic, adaptive
- Communication overhead
  - Distribution and collection (compositing) of data
  - Minimize transfers or support in hardware

Outline

- The Problem
- General Concepts
- Classification
- Load Balancing
- Practical Issues
- ➔ Parallel Volume Rendering
- Parallel Hierarchy Generation

Overview

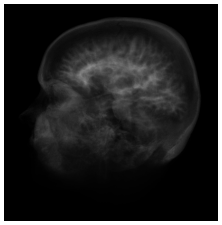
Introduction to Volume Rendering

Volume Rendering on

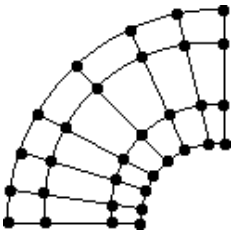
- Regular Grids
- Irregular Grids

Introduction to Volume Rendering (1)

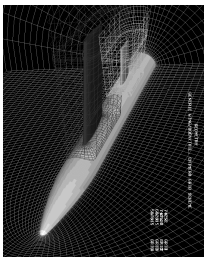
Regular Grids:
- **Fixed grid topology (ie., #neighbors, valence)**
- **Flexible grid geometry**

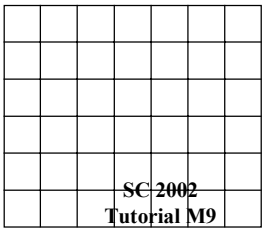


Cartesian/rectilinear



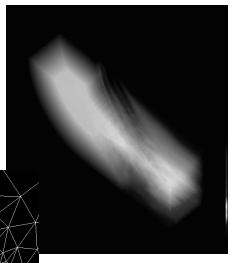
Curvilinear

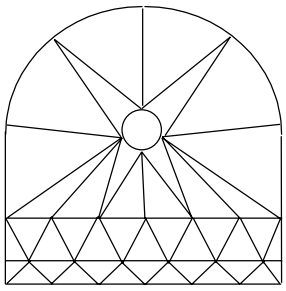


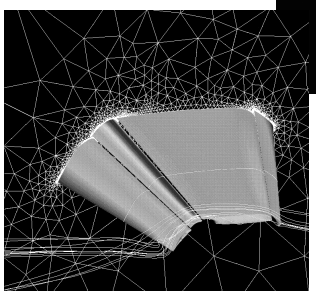


Introduction to Volume Rendering (2)

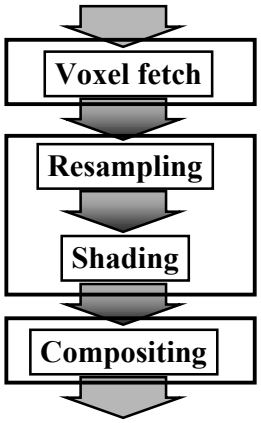
Irregular/Unstructured Grids:
- **Flexible grid topology**
- **Flexible grid geometry**





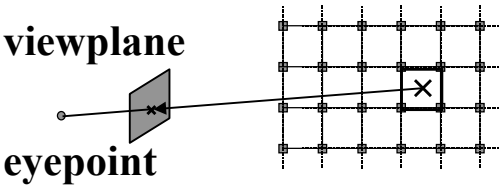


Introduction to Volume Rendering (3)



Volume rendering pipeline

$V_{xyz} \rightarrow$ (R, G, B, alpha)

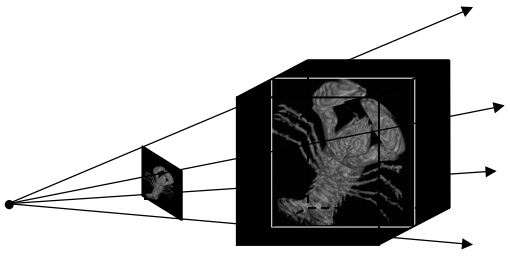


SC 2002
Tutorial M9

Introduction to Volume Rendering (4)

Optical model:

- **Absorption along the projection**
- **Receive light from light sources (emission)**
- **Associative composing**

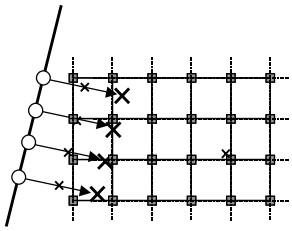


SC 2002
Tutorial M9

Introduction to Volume Rendering (5)

Ray Casting:

- Rays are cast from viewplane
- Samples are composed along the ray into viewplane pixels

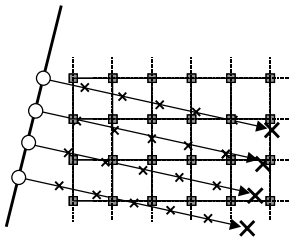


SC 2002
Tutorial M9

Introduction to Volume Rendering (5)

Ray Casting:

- Rays are cast from viewplane
- Samples are composed along the ray into viewplane pixels

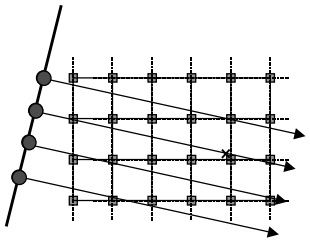


SC 2002
Tutorial M9

Introduction to Volume Rendering (5)

Ray Casting:

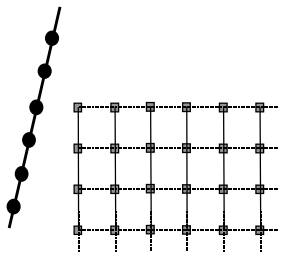
- Rays are cast from viewplane
- Samples are composed along the ray into viewplane pixels



Introduction to Volume Rendering (6)

Shear Warp

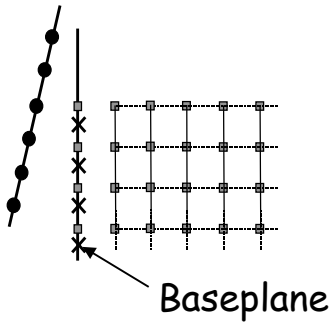
- Base plane ray casting
- Similar parallel approaches
- Lower quality (inter-polation,sampling)
- Fast
- Cartesian grids only



Introduction to Volume Rendering (6)

Shear Warp

- **Base plane ray casting**
- **Similar parallel approaches**
- **Lower quality (interpolation,sampling)**
- **Fast**
- **Cartesian grids only**

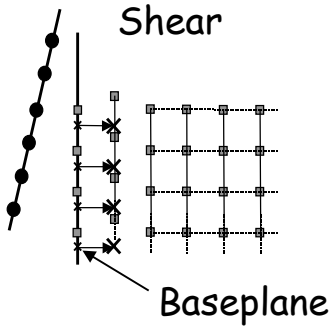


SC 2002
Tutorial M9

Introduction to Volume Rendering (6)

Shear Warp

- **Base plane ray casting**
- **Similar parallel approaches**
- **Lower quality (interpolation,sampling)**
- **Fast**
- **Cartesian grids only**

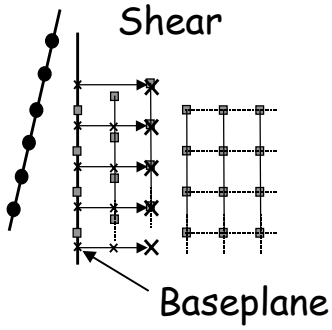


SC 2002
Tutorial M9

Introduction to Volume Rendering (6)

Shear Warp

- **Base plane ray casting**
- **Similar parallel approaches**
- **Lower quality (interpolation,sampling)**
- **Fast**
- **Cartesian grids only**

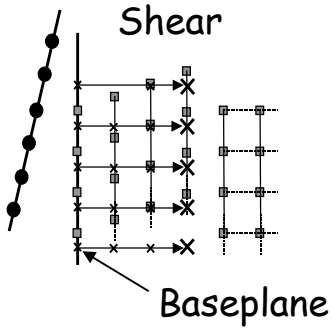


SC 2002
Tutorial M9

Introduction to Volume Rendering (6)

Shear Warp

- **Base plane ray casting**
- **Similar parallel approaches**
- **Lower quality (interpolation,sampling)**
- **Fast**
- **Cartesian grids only**

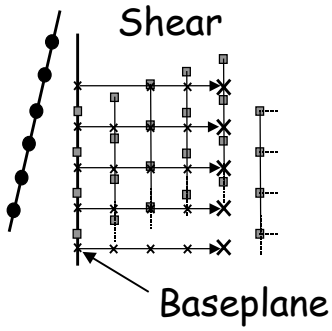


SC 2002
Tutorial M9

Introduction to Volume Rendering (6)

Shear Warp

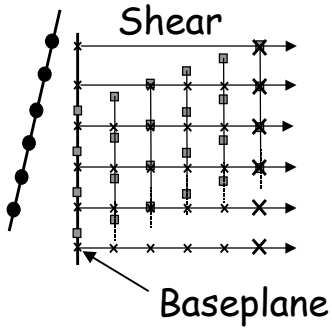
- **Base plane ray casting**
- **Similar parallel approaches**
- **Lower quality (interpolation,sampling)**
- **Fast**
- **Cartesian grids only**



Introduction to Volume Rendering (6)

Shear Warp

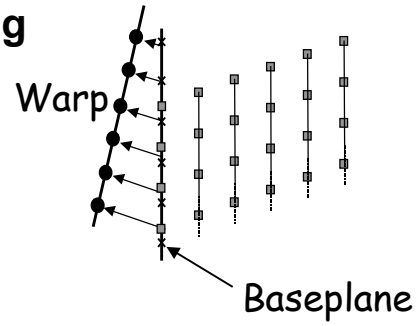
- **Base plane ray casting**
- **Similar parallel approaches**
- **Lower quality (interpolation,sampling)**
- **Fast**
- **Cartesian grids only**



Introduction to Volume Rendering (6)

Shear Warp

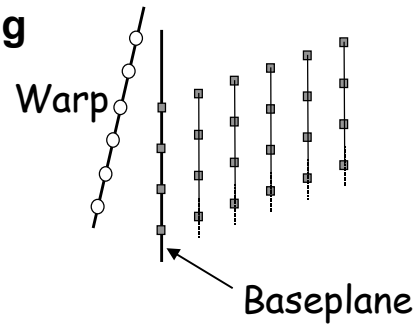
- **Base plane ray casting**
- **Similar parallel approaches**
- **Lower quality (interpolation,sampling)**
- **Fast**
- **Cartesian grids only**



Introduction to Volume Rendering (6)

Shear Warp

- **Base plane ray casting**
- **Similar parallel approaches**
- **Lower quality (interpolation,sampling)**
- **Fast**
- **Cartesian grids only**

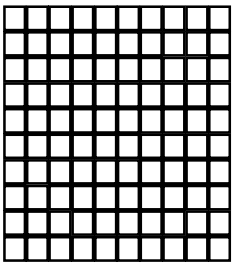


SC 2002
Tutorial M9

Overview

- Introduction to Volume Rendering
- Volume Rendering on
  - Regular Grids in
    - Shared Memory
    - Distributed Memory
  - Irregular Grids

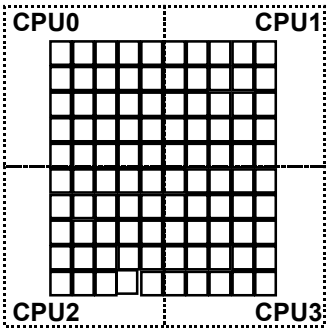
Parallel Ray Casting



Idea: Assign (group of) rays to separate processors

Issues:

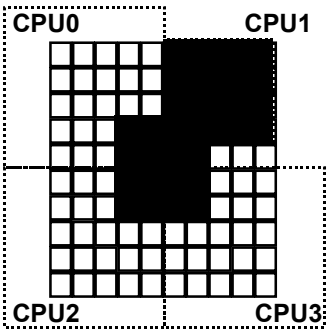
Parallel Ray Casting



Idea: Assign (group of) rays to separate processors

Issues:
- **Load Balancing: static**

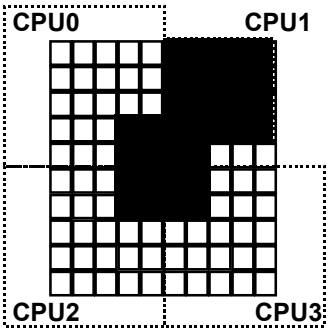
Parallel Ray Casting



Idea: Assign (group of) rays to separate processors

Issues:
- **Load Balancing: dynamic**

Parallel Ray Casting



Idea: Assign (group of) rays to separate processors

Issues:
- **Load Balancing**
- **Data locality and cache coherence (in particular on NUMA)**
- **Scalability**

Example: Nieh and Levoy 1992

- **Ray casting on the Stanford DASH**
- **Screen is sub-divided into titles**
- **Titles assigned dynamically**
- **14 MB dataset on 48 proc's:**
  - Maximum speedup 40, 1.4 fps (non-adaptive)
  - Maximum frame rate 3 fps, speedup 33 (adaptive rendering)

Example: Parker, Shirley, Livnat, Hansen, Sloan 1998

- Parallel ray casting of iso-surfaces on SGI O2K
- Screen is sub-divided into titles
- Dynamic load balancing of variable tile chunks
- Takes great care of cache coherence
- 900MB dataset on 128 proc's: speedups >100, 20 fps

(Similar techniques in Law,Yagel 1996 and Palmer et al. 1997)

Volume Rendering on Regular Grids

Introduction to Volume Rendering
Volume Rendering on
- Regular Grids in
  - Shared Memory
  - Distributed Memory
- Irregular Grids

Distributed-Memory Ray Casting

Issues:

- **Communication / Synchronization**
- **Data Distribution / Replication**
- **Sorting / Compositing**
- **Load Balancing**

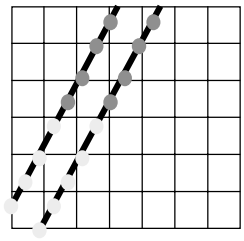
Distributed-Memory Ray Casting

Issues:

- **Communication / Synchronization**
- **Data Distribution / Replication**
  - Often replicated if local memory is large enough
  - Object space chunks depending on ray directions
- **Sorting / Compositing**
- **Load Balancing**

Sorting / Compositing (1)

How to compose partial results?



- **Sampling (partial) volume**
- **Compose (partial) samples**
- **Compose partial results**

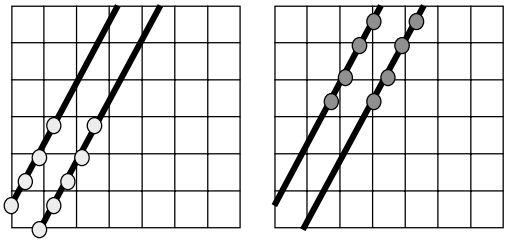
Important: Compositing is associative

Sorting / Compositing (2)

How to compose partial results?

Compositing depends strongly
on load balancing / data distribution

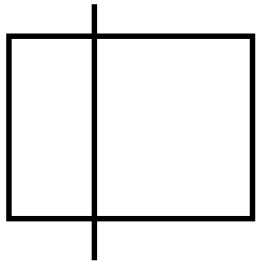
Processor 1 **Processor 2**



Sorting / Compositing (3)

Binary Swap (Ma, Painter, Hansen, Krogh 1993)

- **The data set partitioning based on a K-d tree**
- The image compositing based on a recursive image-subdivision scheme

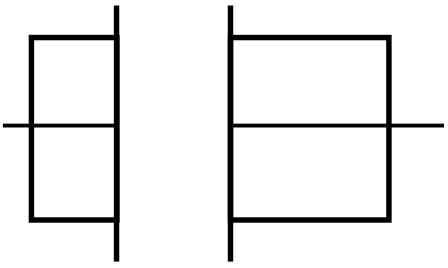


Also: Mackerras, TR ANU 1992

Sorting / Compositing (4)

Binary Swap (Ma, Painter, Hansen, Krogh 1993)

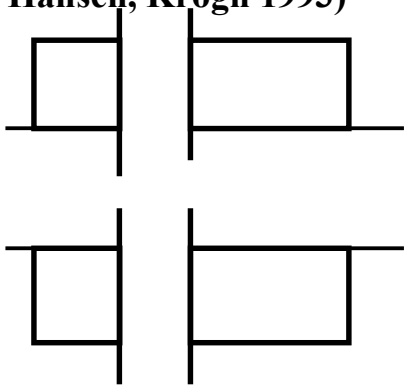
- **The data set partitioning based on a K-d tree**
- The image compositing based on a recursive image-subdivision scheme



Sorting / Compositing (5)

Binary Swap (Ma, Painter, Hansen, Krogh 1993)

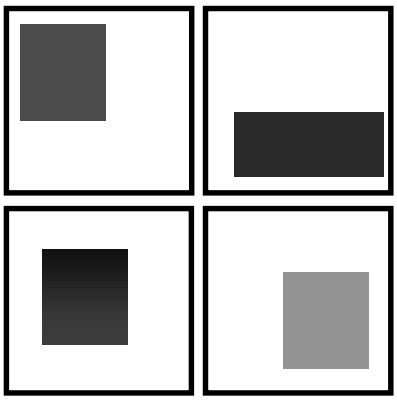
- **The data set partitioning based on a K-d tree**
- The image compositing based on a recursive image-subdivision scheme



Sorting / Compositing (6)

Binary Swap

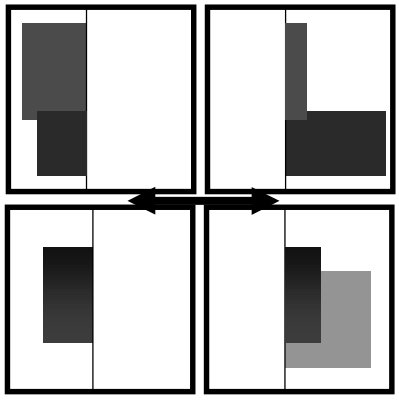
- **The data set partitioning based on a K-d tree**
- The image compositing based on a recursive image-subdivision scheme
  since the data is partioned into convex, non-overlapping regions of 3-space.



Sorting / Compositing (7)

Binary Swap

- **The data set partitioning based on a K-d tree**
- The image compositing based on a recursive image-subdivision scheme
  since the data is partioned into convex, non-overlapping regions of 3-space.

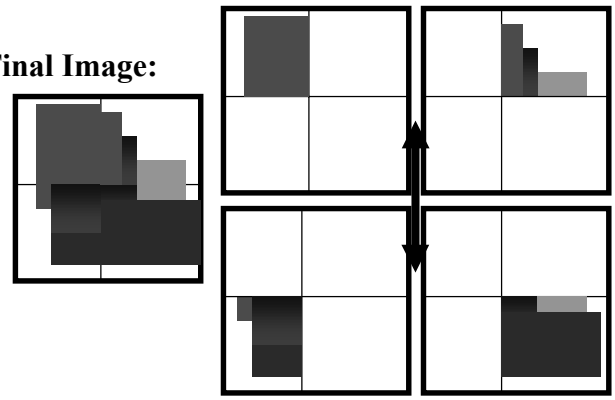


SC 2002
Tutorial M9

Sorting / Compositing (8)

Binary Swap

Final Image:



SC 2002
Tutorial M9

Sorting / Compositing (9)

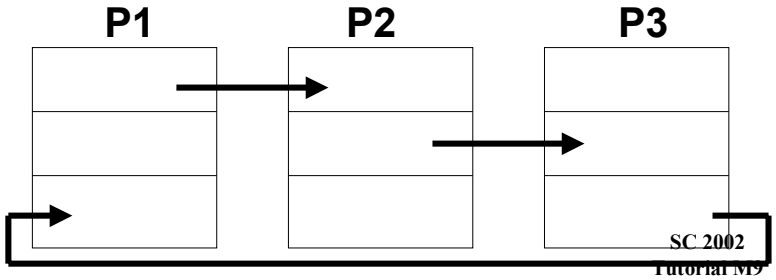
Binary Swap:

- **Compositing with p processors**
- **O(log p) compositing (p should be 2^n)**
- **Compositing of "empty regions"**

Sorting / Compositing (10)

Parallel Pipeline (Lee, Raghanvendra, Nicholsas, TVCG 1997):

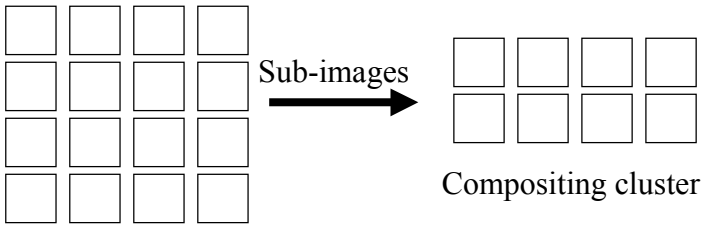
- **Image is split in p parts**
- **Compositing in ring buffer**



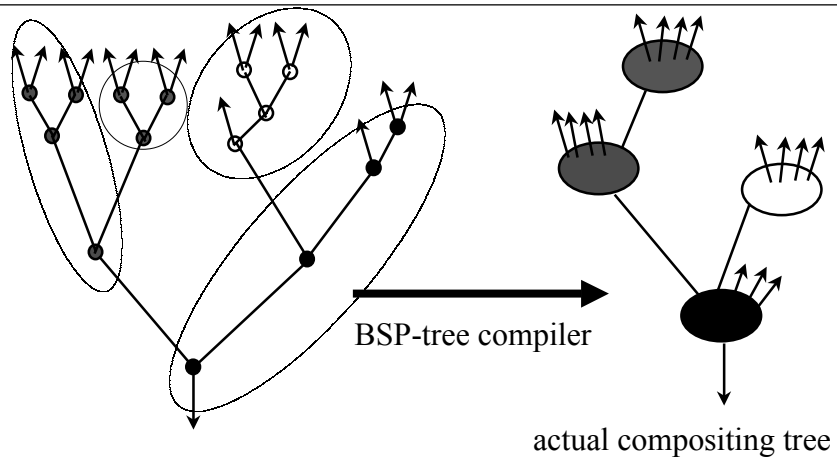
Sorting / Compositing (11)

Rendering/Compositing Clusters (Ramakrishnan, Silva 1999)

- **BSP-Tree compositing**
- **#subimages = #compositors**



Sorting / Compositing (12)

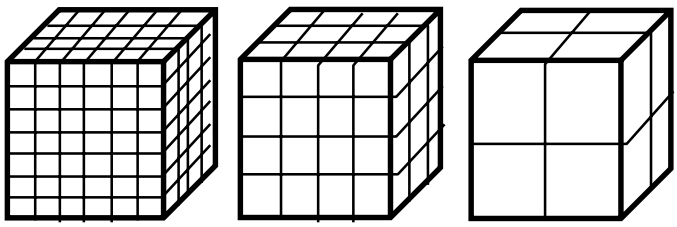


Load Balancing (1)

- Close related to Distribution / Compositing
- Tries to minimize communication
- Static / dynamic / adaptive
- Distribution also requires voxel extend for neighborhood information

Load Balancing (2)

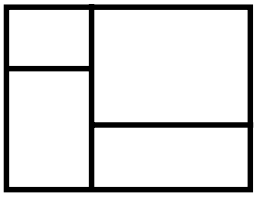
Subdivision into subvolumes
(Hsu, 1993, Westermann 1995, ...):

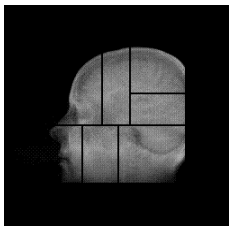


Load Balancing (3)

Content-Based Load Balancing (Silva'96)

- Uses Kd-tree
- Always subdivide along the longest "axis"
- Always subdivide number of full sub-cubes in half
- Static scheme





SC 2002
Tutorial M9

Overview

Introduction to Volume Rendering

Volume Rendering on

- **Regular Grids**
- **Irregular Grids**

SC 2002
Tutorial M9

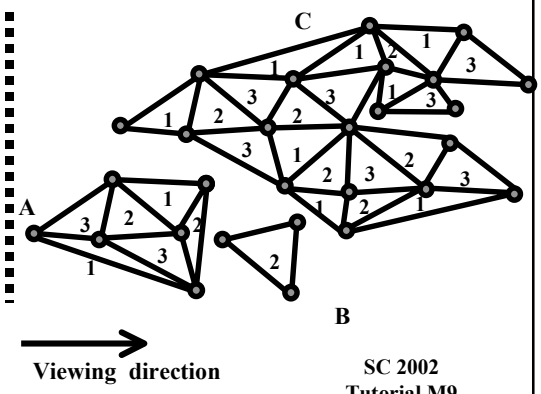
Volume Rendering on Irregular Grids

- Complex topology of grid
- Can have several disconnected components
- Load Balancing has to deal with cells of large size range
- Central problem: sorting of cells for compositing

Example: Ma and Crocket 97 (1)

Distributed-Memory Technique

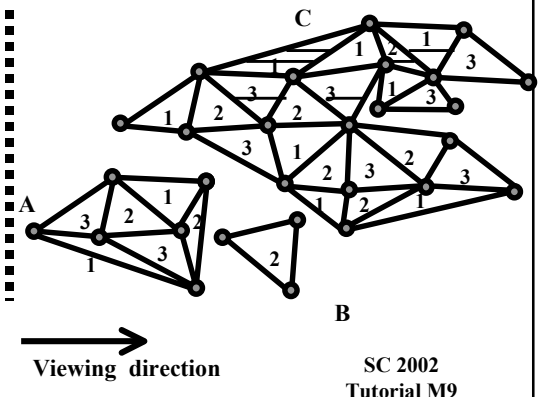
- Round-robin of the cell primitives for load balancing
- May create problems with caches/ local memory



Example: Ma and Crocket 97 (2)

Cell projection in two phases:

- Scan conversion of cells
- Compositing

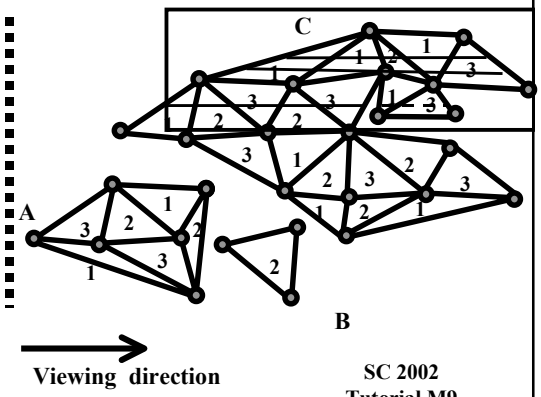


SC 2002
Tutorial M9

Example: Ma and Crocket 97 (3)

Uses k-d tree for sub-partitioning to compose sub-regions first

- Bounds the number of not-composed ray segments



SC 2002
Tutorial M9

Example: Chen, Fujishiro, Nakajima, EGPGV 2002 (1)

- **Target: Earth Simulator (NEC SMP-Cluster)**
- **Complex unstructured grids**
- **Various cell types**
- **Large data: 1-100 Petabytes (1PB = 1000 TB = 10^6 GB)**
- **For more details check website of 4th Eurographics Workshop on Parallel Graphics and Visualization 2002: http://www.gris.uni-tuebingen.de/conf/egpgv02**

Example: Chen, Fujishiro, Nakajima, EGPGV 2002 (2)

Hybrid parallelization

- Message passing: MPI for intra and inter node
- Threading: OpenMP for intra node
- Optimized for loop vectorization for PEs
- Difficulties to obtain good hybrid performance
- Data partitioned into sub volumes
- Dynamic load repartitioning depending on number of relevant voxels/cells

Example: Chen, Fujishiro, Nakajima, EGPGV 2002 (3)

Data partitioned into sub volumes

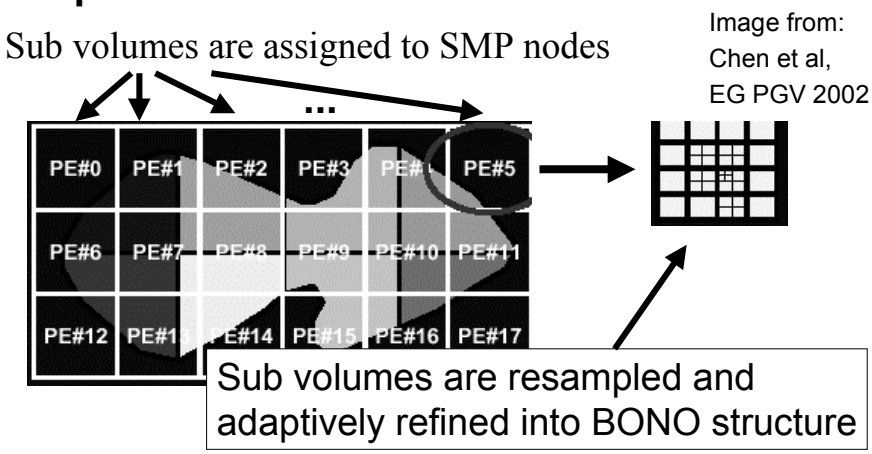


Image from:
Chen et al,
EG PGV 2002

Example: Chen, Fujishiro, Nakajima, EGPGV 2002 (4)

Data partitioned into sub volumes

- Boundary and inner elements
- Sub volumes are individually rendered and opacity is stored
- Sub images are composed using stored opacity
- Rendering by adaptive ray casting

Volume Rendering on Irregular Grids

Other approaches focus on cell sorting

- Giertsen CG&A 1992/93 (sweep plane)
- Williams TOG'92, Silva, et al. VolVis 1998
- Westermann, Vis'97, SIGGRAPH'98, VisSym'01: Hardware-based resampling of irregular grids

Challenges/Opportunities Ahead

- PC clusters and hardware-assisted rendering techniques
- Large- and high-resolution displays and clusters
- Out-of-core rendering
- Time-critical rendering

Outline

- The Problem
- General Concepts
- Classification
- Load Balancing
- Practical Issues
- Parallel Volume Rendering
- ➔ Parallel Hierarchy Generation

Parallel Hierarchy Generation (1)

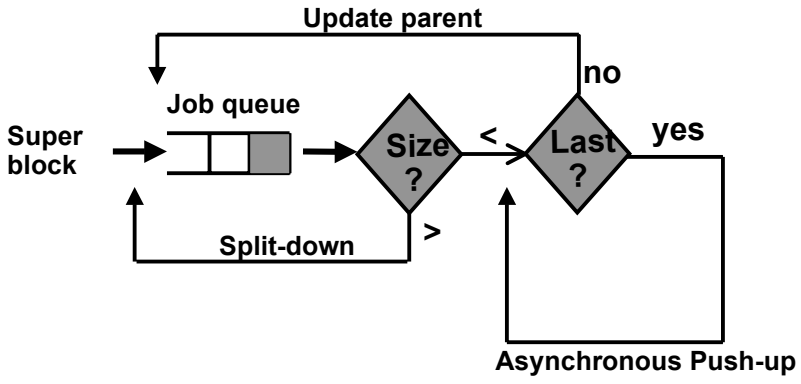
- Hierarchies are often used to handle large data
- Implement a multi-resolution representation
- Generation can be time consuming, in particular if needed often
- Sometimes dynamic generation needed
- Many hierarchical data structures are recursive (ie., Octrees/BONO, BSP-trees, kd-trees, ...)

Parallel Hierarchy Generation (2)

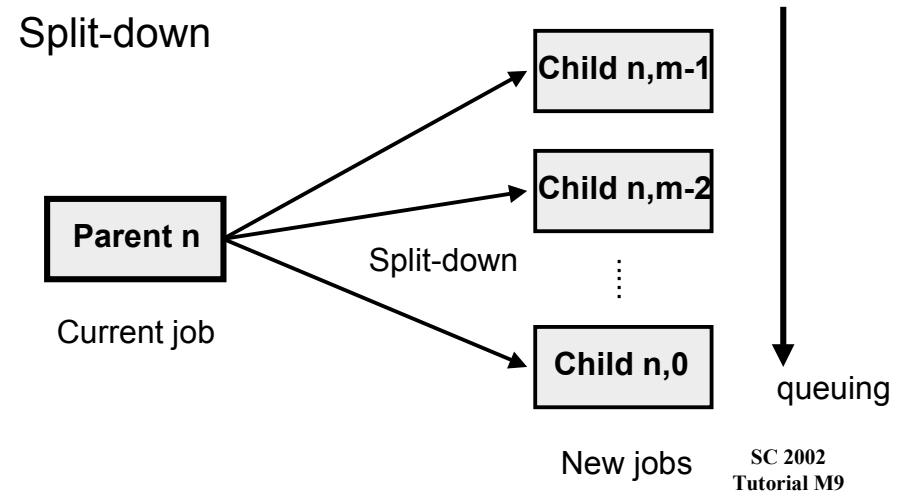
- **Balanced, asynchronous generation wanted**
- **Solution: Asynchronous Push-Up (Bartz et al., 1998-2000):**
  - Parallel split-down of blocks into queue
  - Parallel, asynchonous Push-Up

Parallel Hierarchy Generation (3)

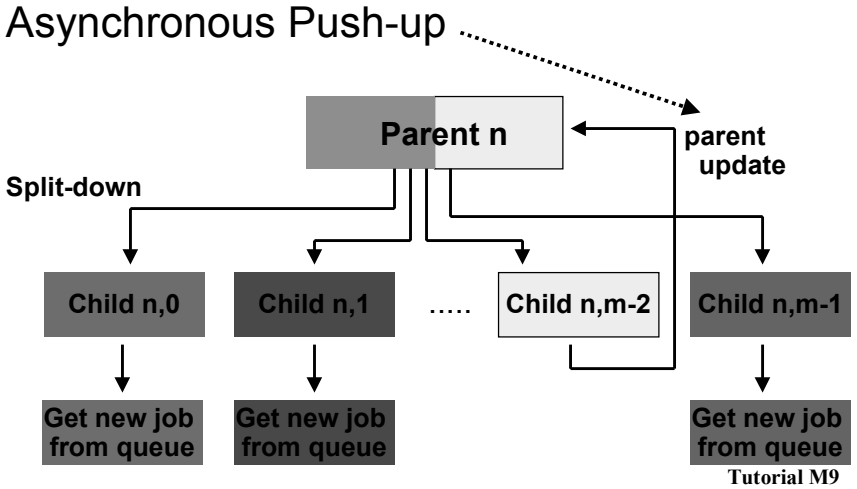
Overview of construction



Parallel Hierarchy Generation (4)

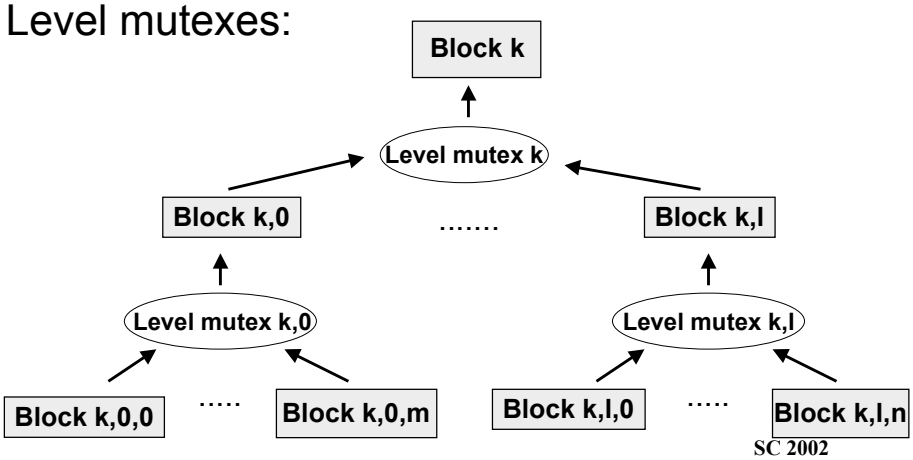


Parallel Hierarchy Generation (5)



Parallel Hierarchy Generation (6)

Level mutexes:



Parallel Hierarchy Generation (7)

Observations

- Test on SMP systems (NUMA, UMA)
- (p)Thread -based, but extendable to OpenMP or message passing
- Architecture sensitive memory managment needed
- No bottlenecks were
  memory access, job queues, level mutexes
  (might be different with more CPUs (>16) or
  distributed/cluster memory)

Summary

- **Parallel Graphics Concepts**
  - Scenarios, sorting, polygon oriented
  - Parallel volume rendering of regular and irregular grids
  - Parallel hierarchy generation
- **Check tutorial website for updated infos: http://www.cse.cgi.edu/~csilva/tutorials/sc2002**

References (which are not in the notes)

Princeton Display Wall:

- R. Samanta, J.Zheng, T. Funkhouser, et al.: Load Balancing for Multi-Projector Rendering Systems, EG/SIGGRAPH workshop on Graphics Hardware, 1999.
- R: Samanta, T. Funkhouser, K. Li, et al.: Hybrid Sort-First and Sort-Last Rendering with a Cluster of PCs, EG/SIGGRAPH workshop on Graphics Hardware, 2000.
- R. Samanta, T. Funkhouser, K. Li: Parallel Rendering with k-way Replication, Symposium on Parallel and Large Data Visualization and Graphics, 2001.

Earth Simulator:

- L. Chen, I. Fujishiro, K. Nakajima: Parallel Performance Optimization of Large-Scale Unstructured Data Visualization for the Earth Simulator, 4th Eurographics Workshop on Parallel Graphics and Visualization, 2002.

Parallel Hierarchies:

- D. Bartz: Optimizing Memory Synchronization for the Parallel Construction of Recursive Tree Hierarchies, 3rd Eurographics Workshop on Parallel Graphics and Visualization, 2000.

Excerpts for the Course Notes of the tutorial on Rendering and Visualization in Parallel Environments

Dirk Bartz
WSI/GRIS
University of Tübingen
Email: bartz@gris.uni-tuebingen.de

Claudio Silva
Oregon Health & Science University
Email: csilva@cse.ogi.edu

Bengt-Olaf Schneider
NVIDIA

The following sections contain parts from our tutorial on "Rendering and Visualization in Parallel Environments which was held in the past years at ACM SIGGRAPH, IEEE Visualization, and the Eurographics conferences. For the full notes, please check

For updates and additional information, see
http://www.gris.uni-tuebingen.de/~bartz/vis2001tutorial

.

Part I

Parallel Graphics Concepts

In this section, we describe several concepts for parallel graphics. Although this section mostly focusses on parallel polygon graphics, some of these concepts are also applicable to parallel volume rendering.

1 Rendering Pipeline

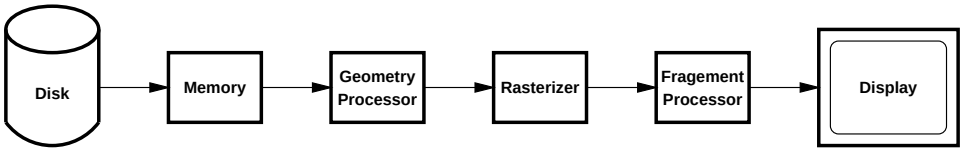


Figure 1: Simplified model of the rendering pipeline.

In this paper we will only consider rendering of polygonal models using the standard rendering pipeline, i.e. we will not discuss ray-tracing or volume rendering. Figure 1 shows the principal steps in rendering of a polygonal model. The description of the model is stored on disk in some file format such as VRML. Before commencing the actual rendering process, the model must be loaded from disk into main memory and converted into an internal representation suitable for rendering. All further processing steps are then memory-to-memory operations. It should be noted that the order of primitives on disk and in the in-memory representation is arbitrary and is usually determined by the application. In particular, the order of primitives in the should not be relied upon when trying to load-balance parallel processors.

Geometry processing forms the first stage of the rendering pipeline. It includes the steps of transforming objects from their intrinsic coordinate system, e.g. model coordinates, into device coordinates, lighting, computation of texture coordinates, and clipping against the view frustum. Except for clipping, all operations in this stage are performed on vertex information. (Clipping operates on entire polygons which is, in particular on SIMD computers, often disrupting the data flow. The steps in the geometry pipeline can be rearranged such that clipping is postponed until the very end when vertices are reassembled into triangles for rasterization [68, 78]. Geometry processing needs mostly floating point operations to implement the matrix multiplications required to transform vertices and to support lighting calculations. Depending on the number of lights and the complexity of the lighting model geometry processing requires between several hundred and a few thousand floating point operations per vertex.

Rasterization converts primitives (typically triangles) described as screen-space vertices into pixels. The resulting pixels are then subjected to various *fragment processing* operations, such as texture mapping, z-buffering, alpha-blending etc. The final pixel values are written into the frame buffer from where they are scanned out onto the display. Most graphics systems implement rasterization and fragment processing as a unit. One notable exception is the PixelFlow system [24].

Rasterization and fragment processing are use predominantly fixed-point or integer computations. Depending on the complexity of the fragment processing operations, between 5 and up to 50 integer computations per pixel and per triangle are required. Because rasterization is algorithmically simple yet requires such a huge number of operations it is often implemented in hardware.

More details on the computational requirements for the different stages in the rendering pipeline can be found for instance in [25, pp. 866-873].

Finally, the complete image is either sent to the screen for display or written to disk. In many parallel rendering algorithms this step forms a performance bottleneck as partial images stored on different processors have to be merged in one central location (the screen or a disk). (Although this step should be included when measuring the end-to-end performance of a parallel rendering system, some researchers explicitly exclude this step due to shortcomings of their particular target platform [20].)

1.1 Single-frame vs. multi-frame rendering

Rendering polygonal models can be driven by several needs. If the model is only used once for the generation of a still image, the entire rendering process outlined above has to be performed. The creation of animation sequences requires rendering of the same model for different values of time and consequently varying values for time-dependent rendering parameters, e.g. view position, object location, or light source intensities. Even though multi-frame rendering could be handled as repeated single-frame rendering, it offers the opportunity to exploit inter-frame coherence. For example, access to the scene database can be amortized over several frames and only the actual rendering steps (geometry processing and rasterization) must be performed for every frame. Other ways to take advantage of inter-frame coherence will be discussed below.

2 Algorithm Classification

For many years the classification of parallel rendering algorithms and architectures has proven to be an elusive goal. We will discuss several such classifications to gain some insight into the design space and possible solutions.

2.1 Pipelining vs. Parallelism

Irrespective of the problem domain, parallelization strategies can be distinguished by how the problem is mapped onto the parallel processors.

For *pipelining* the problem is decomposed into individual steps that are mapped onto processors. Data travel through the processors and are transformed by each stage in the pipeline. For many problems, like the rendering pipeline (sic!), such a partitioning is very natural. However, pipelining usually offers only a limited amount of parallelism. Furthermore, it is often difficult to achieve good load-balancing amongst the processors in the pipeline as the different functions in the pipeline vary in computational complexity.

To overcome such constraints pipelining is often augmented by *replicating* some or all pipeline stages. Data are distributed amongst those processor and worked on in parallel. If the algorithms executed by each of the processors are identical, the processors can perform their operation in lockstep, thus forming a SIMD (single-instruction, multiple-data) engine. If the algorithms contain too many data dependencies thus making SIMD operation inefficient, MIMD (multiple-instruction, multiple-data) architectures are more useful. SIMD implementations are usually more efficient as the processors can share instructions and require very little interprocessor communication or synchronization.

2.2 Object Partitioning vs. Image Partitioning

One of the earliest attempts at classifying partitioning strategies for parallel rendering algorithms took into consideration whether the data objects distributed amongst parallel processors belonged into object space, e.g. polygons, edges, or vertices, or into image space, i.e. collections of pixels such as portions of the screen, scanlines or individual pixels [1]. Object-space partitioning is commonly used for the geometry processing portion of the rendering pipeline, as its operation is intrinsically based on objects. Most parallelization strategies for rasterizers employ image-space partitioning [22, 94, 18, 4, 3, 62] A few architectures apply object-space partitioning in the rasterizer [92, 26, 77].

2.3 Sorting Classification

Based on the observation that rendering can be viewed as a sorting process of objects into pixels [23], different parallel rendering algorithms can be distinguished by where in the rendering pipeline the sorting occurs [27]. Considering the two main steps in rendering, i.e. geometry processing and rasterization, there are three principal locations for the sorting step: Early during geometry processing (*sort-first*), between geometry processing and rasterization (*sort-middle*), and after rasterization (*sort-last*).

Figure 2) illustrates the three approaches. In the following discussion we will follow [27] in referring to a pair of geometry processor and a rasterizer as a renderer.

Sort-middle architectures form the most natural implementation of the rendering pipeline. Many parallel rendering systems, both software and hardware, use this approach, e.g. [4, 22, 3, 17, 15, 96, 20]. They assign primitives to geometry processors that implement the entire geometry pipeline. The transformed primitives are then sent to rasterizers that are each serving a portion of the entire screen. One drawback is the potential for poor load-balancing among the rasterizers due to uneven distribution of objects across the screen. Another problem of this approach is the redistribution of primitives after the geometry stage which requires a many-to-many communication between the processors. A hierarchical multi-step method to reduce the complexity of this global sort is described in [20].

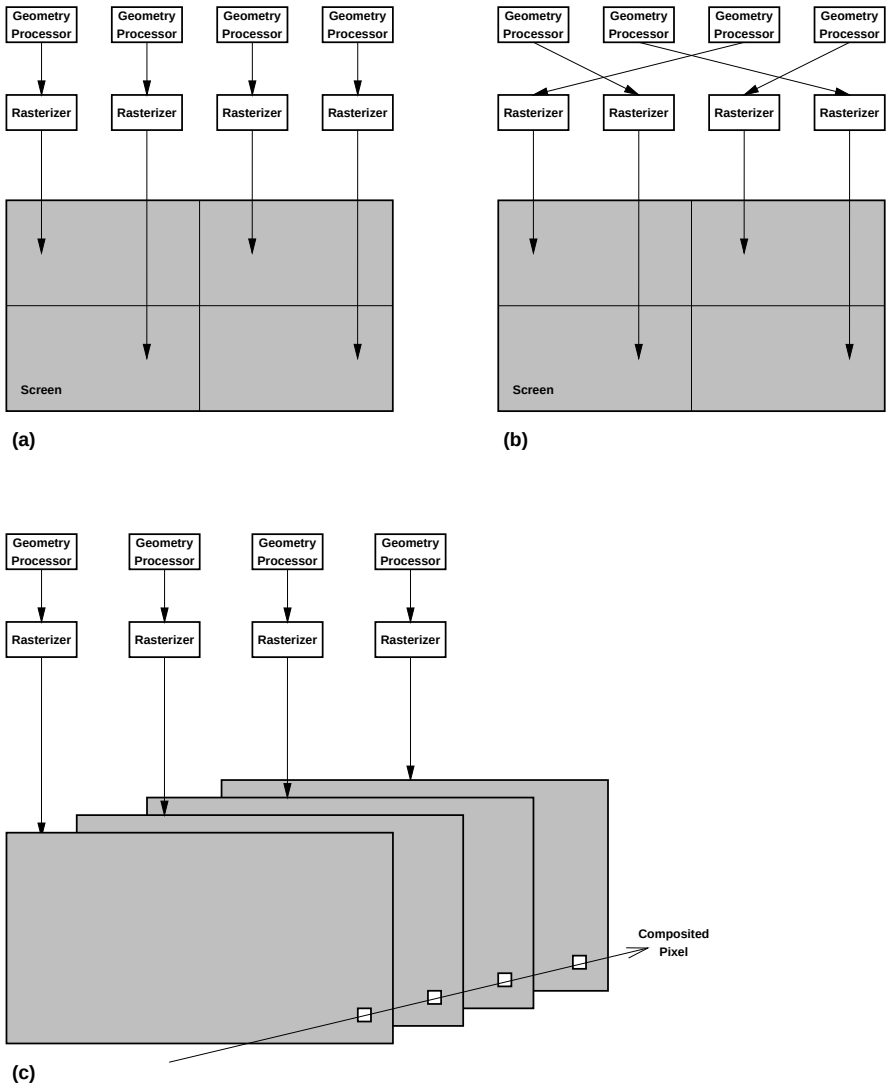


Figure 2: Classification of parallel rendering methods according to the location of the sorting step. (a) sort-first (b) sort-middle (c) sort-last.

Sort-last assigns primitives to renderers that generate a full-screen image of all assigned primitives. After all primitives have been processed, the resulting images are merged/composited into the final image. Since all processors handle all pixels this approach offers good load-balancing properties. However compositing the pixels of the partial images consumes large amounts of bandwidth and requires support by dedicated hardware, e.g. [24]. Further, with sort-last implementations it is difficult to support anti-aliasing, as objects covering the same pixel may be handled by different processors and will only meet during the final compositing step. Possible solutions, like oversampling or A-buffers [9], increase the bandwidth requirements during the compositing step even further.

Sort-first architectures quickly determine for each primitive to which screen region(s) it will contribute. The primitive is then assigned to those renderers that are responsible for those screen regions. Currently, no actual rendering systems are based on this approach even though there are some indications that it may prove advantageous for large models and high-resolution images [63]. [63] claims that sort-first has to redistribute fewer objects between frames than sort-middle. Similar to sort-middle, it is prone to suffer from load-imbalances unless the workload is leveled using an adaptive scheme that resizes the screen regions each processor is responsible for.

Part II

Parallel Rendering

In this part, we discuss important parallel rendering issues which are typical for parallel approaches, such as load-balancing, scheduling, or the foundations of the rendering approaches (for volume rendering).

3 Parallel Polygonal Rendering

As with the parallel implementation of any algorithm the performance depends critically on balancing the load between the parallel processors. There are workload-related and design-related factors affecting the load balancing. We will first discuss workload issues and then describe various approaches to design for good load-balance.

Before the discussion of load balancing strategies, we will define terms used throughout the rest of the discussion. *Tasks* are the basic units of work that can be assigned to a processor, ie. objects, primitives, scanlines, pixels etc. *Granularity* quantifies the minimum number of tasks that are assigned to a processor, ie. 10 scanlines per processor or 128x128 pixel regions. *Coherence* describes the similarity between neighboring elements like consecutive frames or neighboring scanlines. Coherence is exploited frequently in incremental calculations, ie. during scan conversion. Parallelization may destroy coherence, if neighboring elements are distributed to different processors. *Load balance* describes how well tasks are distributed across different processor with respect to keeping all processors busy for all (or most) of the time. Surprisingly, there is no commonly agreed upon definition of load balance in the literature. Here, we define load balance based on the time between when the first and when the last work task finish.

$$LB = 1 - \frac{T - T_f}{T} \tag{1}$$

where T is the total processing time and T_f is the time when the fastest processor finishes.

3.1 Workload Characterization

Several properties of the model are important for analyzing performance and load-balancing of a given parallel rendering architecture. Clipping and object tesselation affect load-balancing during geometry processing, while spatial object distribution and primitive size mostly affect the balance amongst parallel rasterizers.

Clipping. Objects clipped by the screen boundaries incur more work than objects that are trivially accepted or rejected. It is difficult to predict whether an object will be clipped and load-imbalances can result as a consequence of one processor receiving a disproportionate number of objects requiring clipping. There are techniques that can reduce the number of objects that require clipping by enabling rasterizers to deal with objects outside of the view frustum [70, 21]. This reduces the adverse affects of clipping on load-balancing to negligible amounts.

Tesselation. Some rendering APIs use higher order primitives, like NURBS, that are tesselated by the rendering subsystem. The degree of tesselation, ie. the number of triangles per object, determines the amount of data expansion occurring during the rendering process. The degree of tesselation is often view-dependent and hence hard to predict a priori. The variable degree of tesselation leads to load imbalances as one processor's objects may expand into more primitives than objects handled by another processor. Tesselation also affects how many objects need to be considered during the sorting step. In sort-first architectures, primitives are sorted before the tesselation, thus saving communication bandwidth compared to sort-middle architectures.

Primitive distribution. In systems using image-space partitioning, the spatial distribution of objects across the screen decides how many objects must be processed by each processor. Usually, objects are not distributed uniformly, ie. more objects may be located in the center of the screen than along the periphery. This creates potential imbalances in the amount of work assigned to each processor. Below we will discuss different approaches to deal with this problem.

Primitive size. The performance of most rasterization algorithms increases for smaller primitives. (Simply put: It takes less time to generate fewer pixels.) The mix of large and small primitives therefore determines the workload for the rasterizer. Several experiments have shown (see ie. [12]) that many scenes contain a large number of small objects and a few large objects. The primitive size also affects the overlap factor, ie. the number of screen regions affected by an object. The overlap factor affects the performance of image-space partitioning schemes like sort-first and sort-middle algorithms.

3.2 Designing for Load-Balancing

Several design techniques are used to compensate for load-imbalances incurred by different workloads. They can be distinguished as *static*, *dynamic* and *adaptive*.

Static load balancing uses a fixed assignment of tasks to processors. Although, a low (ie. no) overhead is incurred for determining this assignment, load imbalances can occur if the duration of tasks is variable. Dynamic load balancing techniques determine the on the fly which processor will receive the next task. Adaptive load balancing determines an assignment of tasks to processors based on estimated cost for each task, thereby trying to assign equal workload to each processor.

We will now look at several concrete techniques to load balance graphics tasks in multi-processor systems.

On-demand assignment is a dynamic method that relies on the fact that there are many more tasks (objects or pixels) than there are processors. New work is assigned to the first available, idle processor. Except during initialization and for the last few tasks, every processor will be busy all the time. The maximum load imbalance is bounded by the difference in processing time between the smallest (shortest processing time) and largest (longest processing time) task. The ratio of the number of tasks and the number of processors is called the *granularity ratio*. Selecting the granularity ratio requires a compromise between good load balancing (high granularity ratio) and overhead for instance due to large overlap factor (low granularity ratio). The optimal granularity ratio depends on the model, typical values range from about 4 to 32.

An example for dynamic load-balancing through the use of on-demand assignment of tasks is the Pixel-planes 5 system [22]. In Pixel-planes 5, the tasks are 80 128x128 pixel regions that are assigned to the next available rasterizer module. The dynamic distribution of tasks also allows for easy upgrade of the system with more processors.

Care must be taken when applying this technique to geometry processing: Some graphics APIs (like OpenGL) require that operations are performed in the exact order in which they were specified, ie. objects are not allowed to "pass each other" on their way through the pipeline. MIMD geometry engines using on-demand assignment of objects could violate that assumption and must therefore take special steps to ensure temporal ordering, ie. by labeling objects with time stamps.

Interleaving is a static technique which is frequently used in rasterizers to decrease the sensitivity to uneven spatial object distributions. In general, the screen is subdivided into regions, ie. pixels, scanlines, sets of scanlines, sets of pixel columns, or rectangular blocks. The shape and the size of these regions determines the overlap factor. For a given region size, square regions minimize the overlap factor [27]. Among n processor, each processor is responsible for every n-th screen region. The value n is known as the interleave factor. Since clustering of objects usually occurs in larger screen regions and since every object typically covers several pixels, this technique will eliminate most load-imbalances stemming from non-uniform distribution of objects. Interleaving makes it harder to exploit spatial coherence as neighboring pixels (or scanlines) are assigned to different processors. Therefore, the interleave factor, ie. the distance between pixels/scanlines assigned to the same processor, must be chosen carefully. Several groups have explored various aspects of interleaving for parallel rasterization, ie. [38].

Adaptive scheduling tries to achieve balanced loading of all processors by assigning different number of tasks depending on task size. For geometry processing this might mean to assign fewer objects to processors that are receiving objects that will be tesselated very finely. In image-space schemes this means that processors are assigned smaller pixel sets in regions with many objects, thus equalizing the number of objects assigned to each processor.

Adaptive scheduling can be performed either dynamically or statically. Dynamic adaptation is achieved by monitoring the load-balance and if necessary splitting tasks to off-load busy processors. Such a scheme is described in [96]: Screen regions are initially assigned statically to processors. If the system becomes unbalanced, idle processors grab a share of the tasks of the busy processors.

Statically adaptive schemes attempt to statically assign rendering tasks such that the resulting work is distributed evenly amongst all processors. Such schemes are either predictive or reactive. Predictive schemes estimate the actual workload for the current frame based on certain model properties. Reactive schemes exploit inter-frame coherence and determine the partioning of the next frame based on the workload for the current frame, ie. [20, 74].

Numerous rendering algorithms using adaptive load-balancing have been described. Most these methods operate in two steps: First, the workload is estimated by counting primitives per screen regions. Then, either the screen is subdivided to create regions with approximately equal workload or different number of fixed-sized regions (tasks) are assigned to the processors.

One of the earliest such methods was described by Roble [74]. The number of primitives in each screen region are counted. Then, low-load regions are combined to form regions with a higher workload. Regions with high workload are split in half. Since the algorithm does not provide any control over the location of splits, it has the potential of low effectiveness in load-balancing.

Whelan [94] published a similar method that determines the workload by inspecting the location of primitive centroids. High-workload regions are split using a median-cut algorithm, thereby improving the load-balancing behavior over Roble's algorithm. The median-cut approach incurs a sizeable overhead for sorting primitives. The use of centroids instead of actual primitives introduces errors because the actual size of the primitives is not taken into account.

Whitman [95] measures workload by counting the number of primitives overlapping a region. The screen is then subdivided using a quad-tree to create a set of regions with equal workload. The principal problem with the algorithm is that work may be overestimated due to double-counting of primitives.

Mueller [63] improves over the shortcomings of Whelan's method. The Mesh-based Adaptive Hierarchical Decomposition (MAHD) is based on a regular, fine mesh overlaid over the screen. For each mesh cell, primitives are counting in inverse proportion to their size. This approach is experimentally justified and avoids the double-counting problems of Whitman's method. The mesh cells are then aggregated into larger clusters by using a summed-area table for the workload distribution across the screen. The summed-area table is more efficient than the media-cut algorithm in Whelan's method.

Later, Whitman [96] describes an dynamically adaptive scheme. Initially, the screen is subdivided regularly to a predetermined granularity ratio (here: 2). During the actual processing run, processors that complete their tasks early, "steal" work from busier processors by splitting their work region. In order to avoid instability and/or extra overhead, processors steal from the processor with most work left. Also, splitting only occurs if the remaining work exceeds a set threshold. Otherwise, the busy processor finishes his work uninterrupted.

Finally, Ellsworth [20] describes a reactive adaptive load-balancing method. The method is based on a fixed grid overlaid over the screen. Between the frames of an animation, the algorithm counts the number of primitives overlapping each grid cell and uses this count to estimate the workload per cell. Then, cells are assigned to processors for the upcoming frame. The assignment is a

multiple-bin-packing algorithm: Regions are first sorted by descending polygon counts; the regions are assigned in this order to the processor with the lightest workload.

Frame-parallel rendering is a straight-forward method to use parallel processors for rendering. Each processor works independently on one frame of an animation sequence. If there is little variation between consecutive frames, frames can be assigned statically to processors as all processor tend complete their respective frame(s) in approximately the same time. If processing time varies between frames, it is also possible to assign frames dynamically (on-demand assignment). In either case, the processors are working on independent frames and no communication between processors is required after the initial distribution of the model. Unfortunately, this approach is only viable for rendering of animation sequence. It is not suitable for interactive rendering as it typically introduces large latencies between the time a frame is issued by the application and when it appears on the screen.

The *application programming interface (API)* impacts how efficiently the strategies outlined above can be implemented. Immediate-mode APIs like OpenGL or Direct3D do not have access to the entire model and hence do not allow global optimizations. Retained-mode APIs like Phigs, Performer, OpenGL Optimizer, Java3D and Fahrenheit maintain an internal representation of the entire model which supports partitioning of the model for load-balancing.

3.3 Data Distribution and Scheduling

In distributed memory architectures, ie. clusters of workstations or message-passing computers, object data must be sent explicitly to the processors. For small data sets, one can simply send the full data set to every processor and each processor is then instructed which objects to use. This approach fails however for large models either because there is not enough storage to replicate the model at every processor and/or the time to transfer the model is prohibitive due to the bandwidth limitations of the network.

Therefore, most implementations replicate only small data structures like graphics state, ie. current transformation matrices, light source data, etc., and distribute the storage for large data structures, primarily the object descriptions and the frame buffer.

For system using static assignment of rendering tasks object data have to be distributed only during the initialization phase of the algorithm. This makes it easy to partition the algorithm into separate phases that can be scheduled consecutively.

For dynamic schemes data must be distributed during the entire process. Therefore processors cannot continuously work rendering objects but must instead divide their available cycles between rendering and communicating with other processors. Such an implementation is described in [15]: The system implements a sort-middle architecture where each processor works concurrently on geometry processing and rasterization, ie. producing and consuming polygons. The advantage is that only a small amount of memory must be allocated for polygons to be transferred between processors. Determining the balance between polygon transformation (generation) and polygon rasterization (consuming) is not obvious. However, [15] states that the overall system performance is fairly insensitive to that choice.

3.4 Summary

Parallel rendering of polygonal datasets faces several challenges most importantly load-balancing. Polygon rendering proceeds in two main steps: geometry processing and rasterization. Both steps have unique computational and communication requirements.

For geometry processing load balancing is usually achieved using on-demand assignment of objects to idle processors. For rasterization, interleaving of pixels or scanlines mostly eliminates load-balancing problems at the expense of less inter-pixel or inter-scanline coherence for each processor. Adaptive load-balancing schemes estimate or measure the workload and divide the screen into regions that will create approximately equal workload.

4 Parallel Volume Rendering

Volume rendering [41] is a powerful computer graphics technique for the visualization of large quantities of 3D data. It is specially well suited for three dimensional scalar [44, 19, 90, 75] and vector fields [13, 54]. Fundamentally, it works by mapping quantities in the dataset (such as color, transparency) to properties of a cloud-like material. Images are generated by modeling the interaction of light with the cloudy materials [100, 56, 55]. Because of the type of data being rendered and the complexity of the lighting models, the accuracy of the volume representation and of the calculation of the volume rendering integrals [6, 40, 39] are of major concern and have received considerable interest from researchers in the field.

A popular alternative method to (direct) volume rendering is isosurface extraction, where given a certain value of interest $\lambda \in \mathcal{R}$, and some scalar function $f : \mathcal{R}^3 \rightarrow \mathcal{R}$, a polygonal representation for the implicit surface $f(x, y, z) = \lambda$ is generated. There are several methods to generate isosurfaces [47, 57, 34, 67], the most popular being the marching cubes method [47]. Isosurfaces have a clear advantage over volume rendering when it comes to interactivity. Once the models have been polygonized (and simplified [80] – marching cubes usually generate lots of redundant triangles), hardware supported graphics workstation can be used to speed up the rendering. Isosurfaces have several disadvantages, such as lack of fine detail and flexibility during rendering (specially for handling multiple transparent surfaces), and its binary decision process where surfaces are either inside or outside a given voxel tends to create artifacts in the data (there is also an *ambiguity* problem, that has been addressed by later papers like [67]).

4.1 Volumetric Data

Volumetric data comes in a variety of formats, the most common being (we are using the taxonomy introduced in [89]) cartesian or regular data. Cartesian data is typically a 3D matrix composed of voxels (a *voxel* can be defined in two different ways, either as

the datum in the intersection of each three coordinate aligned lines, or as the small cube, either definition is correct as long as used consistently), while the regular data has the same representation but can also have a scaling matrix associated with it.

Irregular data comes in a large variety, including curvilinear data, that is data defined in a *warped* regular grid, or in general, one can be given scattered (or unstructured) data, where no explicitly connectivity is defined. In general, scattered data can be composed of tetrahedra, hexahedra, prisms, etc. An important special case is tetrahedral grids. They have several advantages, including easy interpolation, simple representation (specially for connectivity information), and the fact that any other grid can be interpolated to a tetrahedral one (with the possible introduction of Steiner points). Among their disadvantages is the fact that the size of the datasets tend to grow as cells are decomposed into tetrahedra. In the case of curvilinear grids, an accurate decomposition will make the cell complex contain five times as many cells. More details on irregular grids are postponed until Section 4.7.

4.2 Interpolation Issues

In order to generate the cloud-like properties from the volumetric data, one has to make some assumptions about the underlying data. This is necessary because the rendering methods typically assume the ability to compute values as a continuous function, and (for methods that use normal-based shading) at times, even derivatives of such functions anywhere in space. On the other hand, data is given only at discrete locations in space usually with no explicit derivatives. In order to correctly interpolate the data, for the case of regular sampled data, it is generally assumed the original data has been sampled at a high enough frequency (or has been low-pass filtered) to avoid aliasing artifacts [33]. Several interpolation filters can be used, the most common by far is to compute the value of a function $f(x, y, z)$ by trilinearly interpolating the eight closest points. Higher order interpolation methods have also been studied [8, 52], but the computational cost is too high for practical use.

In the case of irregular grids, the interpolation is more complicated. Even finding the cell that contains the sample point is not as simple or efficient as in the regular case [64, 72]. Also, interpolation becomes much more complicated for cells that are not tetrahedra (for tetrahedra a single linear function can be made to *fit* on the four vertices). For curvilinear grids, trilinear interpolation becomes dependent on the underlying coordinate frame and even on the cell orientation [98, 31]. Wilhelms et al. [98] proposes using inverse distance weighted interpolation as a solution to this problem. Another solution would be to use higher order interpolation. In general, it is wise to ask the creator of the dataset for a suitable fitting function.

4.3 Optical Models for Volume Rendering

Volume rendering works by modeling volume as cloud cells composed of semi-transparent material which emits its own light, partially transmits light from other cells and absorbs some incoming light [99, 53, 55]. Because of the importance of a clear understanding of such a model to rendering both, regular and irregular grids, the actual inner workings of one such mechanism is studied here. Our discussion closely follows the one in [99].

We assume each volume cells (differentially) emits light of a certain color $E_\lambda(x, y, z)$, for each color channel λ (red, green and blue), and absorbs some light that comes from behind (we are assuming no multiple scattering of light by particles – our model is the simplest "useful" model – for a more complete treatment see [53]).

Correctly defining opacity for cells of general size is slightly tricky. We define the *differential opacity* at some depth z to be $\Omega(z)$. Computing $T(z)$, the fraction of light transmitted through depth 0 to z (assuming no emission of light inside the material), is simple, we just need to notice that the amount of transmitted light at $z + \Delta z$ is just the amount of light at z minus the attenuation $\Omega(z)$ over a distance of Δz:

$$T(z + \Delta z) = T(z) - \Omega(z)T(z)\Delta z \tag{2}$$

what (after making a division by Δz and taking limits) implies

$$\frac{dT(z + \Delta z)}{dz} = -\Omega(z)T(z) \tag{3}$$

The solution to this linear equation of the first order [11] with boundary condition $T(0) = 1$ is:

$$T(z) = e^{-\int_0^z \Omega(u)du} \tag{4}$$

The accumulated opacity over a ray from front-to-back inside a cell of depth d is $(1 - T(d))$. An important special case is when the cell has constant differential opacity Ω, in this case $T(z) = e^{-\Omega z}$. Before we continue, we can now solve the question of defining *differential opacity* Ω from the *unity* opacity (usually user defined and saved in a transfer function table). A simple formula can express Ω in terms of O:

$$\Omega = \log(\frac{1}{1 - O}) \tag{5}$$

If the model allows for the emission of light inside the material, a similar calculation can be used to calculate the intensity I_λ for each color channel inside a cell. In this case using an initial intensity $I_\lambda(0) = 0$, the final system and solutions are as follows:

$$\frac{dI_\lambda(z)}{dz} = -\Omega(z)I_\lambda(z) + E_\lambda(z) \tag{6}$$

$$I_\lambda(z) = T(z) \int_0^z \frac{E_\lambda(v)}{T(v)} dv \tag{7}$$

Specializing the solution for constant color and opacity cells (as done above) we get the simple solution:

$$I_\lambda(z) = \frac{E}{\Omega}(1 - e^{-\Omega z}) \tag{8}$$

Usually, for computational efficiency, the exponential in the previous equation is approximated by its first terms in the Taylor series. [99, 53, 55] describe in detail analytical solutions under different assumptions about the behavior of the opacity and emitted colors inside the cells, extensions to more complex light behavior and the several tradeoffs of approximating the exponentials with linear functions.

The previous equations show how to calculate the continuous color and opacity intensity, usually this calculation is done once for every cell, and the results from each cell are *composited* in a later step. Compositing operators were first introduced in [71], and are widely used. The most used operator in volume visualization is the **over** operator, its operation is basically to add the brightness of the current cell to the attenuated brightness of the one behind, and *in the case of front-to-back compositing* update the opacities of the cells. The equations for the **over** operator are:

$$C_o = C_a + C_b(1 - O_a) \tag{9}$$

$$O_o = O_a + O_b(1 - O_a) \tag{10}$$

It is important to note, that in these equations the colors are saved pre-multiplied by the opacities (i.e., the actual color is C_o/O_o), this saves one multiplication per compositing operation.

4.4 Ray Casting

A popular method to generate images from volume data is to use *ray casting* [32, 44] which is a simplified variation of the *ray tracing* algorithm for global illumination rendering. Ray casting works by casting (at least) one ray per image pixel into volume space, point sampling the scene with some lighting model (like the one just presented) and compositing the samples as described in the previous section. This method is very flexible and extremely easy to implement. There are several extensions of basic ray casting to include higher order illumination effects, like discrete ray tracing [101], and volumetric ray tracing [88]. Both of these techniques take into account global illumination effects incorporating more accurate approximations of the more general rendering equation [39]. A more recent highly parallized and optimized approach is interactive ray tracing by Parker et al. [69]. Here,

Because of its size, volumetric ray casting (and ray tracing) is very expensive. Several optimizations have been applied to ray tracing [45, 46, 16]. One of the most effective optimizations are the *presence acceleration* techniques, that exploit the fact volumetric data is relatively sparse [45, 46, 16, 103, 102]. Levoy [45] introduced the idea by using an octree [76] to skip over empty space. His idea was further optimized by Danskin and Hanrahan [16] to not only skip over empty space, but also to speed up sampling calculations over uniform regions of the volume. Another important acceleration techniques include *adaptive image sampling* and *early ray termination*. In 1994, Lacroute and Levoy [42] introduced the shear warp volume rendering algorithm – based on previous shear-warp factorizations – which combines base-plane ray casting with bilinear interpolation of samples within the volume slices. While this introduces a number of undersampling problems, it enables a high framerate for volume rendering.

PARC – Hardware-Based Presence Acceleration

Avila, Sobierajski and Kaufman [5, 87] introduced the idea of exploiting the graphics hardware on workstations to speed up volume rendering. First, they introduce PARC (Polygon Assisted Ray Casting) [5], a technique that uses the Z-buffer [28] to find the closest and farthest possibly contributing cells. Later, a revised technique [87] is proposed that (still using the Z-buffer) can produce a better approximation of the set of contributing cells.

Their algorithm consists of first creating a polygonal representation of the set of contributing cells (based on axis aligned quadrilaterals) from a *coarse* volume (see Figure 3). The coarse volume is calculated by grouping neighboring voxels together, creating *supervoxels*. Each supervoxel is then tested for the presence of *interesting* voxels (i.e., voxels that belong to the range of voxels mapped to non-zero intensities and opacities by the transfer functions). All six external faces of supervoxels are then marked based on its possible visibility (the second method seems to need to project all the faces).

In order to perform the actual rendering, in the first method (called *Depth Buffer PARC*), all the visible quadrilaterals are transformed and scan-converted twice. Once for finding the first non-empty front voxel, and again to determine the final integration location. In the second method (called *Color Buffer PARC*), a sweep along the closest major axis is generated by coloring the PARC cubes with power of two numbers (so they do not interfere with each other), what leaves a footprint of the intervals (t_i, t_{i+1}) that can be used to better sample the regions having interesting voxels. This can be quite a savings, given that volumes are quite sparse (most of the time, only 5-10% of a volume contains any lighting and shading information for a given set of transfer functions).

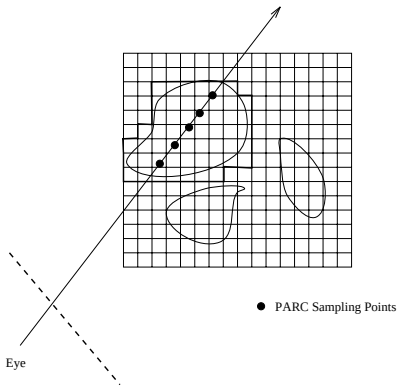


Figure 3: *Polygon Assisted Ray Casting.*

4.5 Splatting or Projection

Ray casting, described in Section 4.4, works from the image space to the object space (volume dataset). Another way of achieving volume rendering is to reconstruct the image from the object space to the image space, by computing for every element in the dataset its contribution to the image. Several such techniques have been developed [19, 93].

Westover's PhD dissertation describes the *Splatting* technique. In splatting, the final image is generated by computing for each voxel in the volume dataset its contribution to the final image. The algorithm works by virtually "throwing" the voxels onto the image plane. In this process every voxel in the object space leaves a *footprint* in the image space that will represent the object. The computation is processed by virtually "peeling" the object space in slices, and by accumulating the result in the image plane.

Formally the process consists of reconstructing the signal that represents the original object, sampling it and computing the image from the resampled signal. This reconstruction is done in steps, one voxel at a time. For each voxel, the algorithm calculates its contribution to the final image, its footprint, and then it accumulates that footprint in the image plane buffer. The computation can take place in back-to-front or front-to-back order. The footprint is in fact the reconstruction kernel and its computation is key to the accuracy of the algorithm. Westover [93] proves that the footprint does not depend on the spatial position of voxel itself (for parallel projections), thus he is able to use a lookup table to approximate the footprint. During computation the algorithm just need to multiply the footprint with the color of the voxel, instead of having to perform a more expensive operation.

Although projection methods have been used for both regular and irregular grids, they are more popular for irregular grids. In this case, projection can be sped up by using the graphics hardware (Z-buffer and texture mapping) [82].

4.6 Parallel Volume Rendering of Regular Grids

Here, we present a high performance parallel volume rendering engine for our PVR system. Our research has introduced two contributions to parallel volume rendering: *content-based load balancing* and *pipelined compositing*. Content-based load balancing (Section 4.6.2) introduces a method to achieve better load balancing in distributed memory MIMD machines. Pipelined compositing (Section 4.6.3) proposes a component dataflow for implementing the *Parallel Ray Casting* pipeline.

The major goal of the research presented is to develop algorithms and code for volume rendering extremely large datasets at reasonable speed with an aim on achieving real-time rendering on the next generation of high-performance parallel hardware. The sizes of volumetric data we are primarily interested are in the approximate range of 512-by-512-by-512 to 2048-by-2048-by-2048 voxels. Our primary hardware focus is on distributed-memory MIMD machines, such as the Intel Paragon and the Thinking Machines CM-5.

A large number of parallel algorithms for volume rendering have been proposed. Schroeder and Salem [79] have proposed a shear-based technique for the CM-2 that could render 128^3 volumes at multiple frames a second, using a low quality filter. The main drawback of their technique is low image quality. Their algorithm had to redistribute and resample the dataset for each view change. Montani et al. [61] developed a distributed memory ray tracer for the nCUBE, that used a hybrid image-based load balancing and context sensitive volume distribution. An interesting feature of their algorithm is the use of clusters to generate higher drawing rates at the expense of data replication. However, their rendering times are well over interactive times. Using a different volume distribution strategy but still a static data distribution, Ma et al. [48] have achieved better frame rates on a CM-5. In their approach the dataset is distributed in a K-d tree fashion and the compositing is done in a tree structure. Others [37, 7, 65] have used similar load balancing schemes using static data distribution, for either image compositing or ray dataflow compositing. Nieh and Levoy [66] have parallelized an efficient volume ray caster [45] and achieved for that time good performance (and good scaling results) performance on a shared memory DASH machine.

4.6.1 Performance Considerations

In analyzing the performance of parallel algorithms, there are many considerations related to the machine limitations, like for instance, communication network latency and throughput [65]. *Latency* can be measured as the time it takes a message to leave

the source processor and be received at the destination end. *Throughput* is the amount of data that can be sent on the connection per unit time. These numbers are particularly important for algorithms in distributed memory architectures. They can change the behavior of a given algorithm enough to make it completely impractical.

Throughput is not a big issue for methods based on volume ray casting that perform static data distribution with ray dataflow as most of the communication is amortized over time [61, 37, 7]. On the other hand, methods that perform compositing at the end of rendering or that have communication scheduled as an implicit synchronization phase have a higher chance of experiencing throughput problems. The reason for this is that communication is scheduled all at the same time, usually exceeding the machines architectural limits. One should try to avoid synchronized phases as much as possible.

Latency is always a major concern, any algorithm that requires communication pays a price for using the network. The start up time for message communication is usually long compared to CPU speeds. For instance, in the iPSC/860 it takes at least 200μs to complete a round trip message between two processors. Latency hiding is an important issue in most algorithms, if an algorithm often blocks waiting for data on other processors to continue its execution, it is very likely this algorithm will perform badly. The classic ways to hide latency is to use pipelining or pre-fetching [36].

Even though latency and throughput are very important issues in the design and implementation of a parallel algorithm, the most important issue by far is *load balancing*. No parallel algorithm can perform well without a good load balancing scheme.

Again, it is extremely important that the algorithm has as few inherently sequential parts as possible if at all. Amdahl's law [36] shows how speed up depends on the parallelism available in your particular algorithm and that *any*, however small, sequential part will eventually limit the speed up of your algorithm.

Given all the constraints above, it is clear that to obtain good load balancing one wants an algorithm that:

- Needs low throughput and spreads communication well over the course of execution.
- Hides the latency, possibly by pipelining the operations and working on more than one image over time.
- Never causes processors to idle and/or wait for others without doing *useful work*.

A subtle point in our requirements is in the last phrase, how do we classify *useful work* ? We define useful work as the number of instructions I_{opt} executed by the best sequential algorithm available to volume render a dataset. Thus, when a given parallel implementation uses a suboptimal algorithm, it ends up using a much larger number of instructions than theoretically necessary as each processor executes more instructions than $\frac{I_{opt}}{P}$ (P denotes the number of processors). Clearly, one needs to compare with the best sequential algorithm as this is the actual speed up the user gets by using the parallel algorithm instead of the sequential one.

The last point on useful work is usually neglected in papers on parallel volume rendering and we believe this is a serious flaw in some previous approaches to the problem. In particular, it is widely known that given a transfer function and some segmentation bounds, the amount of useful information in a volume is only a fraction of its total size. Based on this fact, we can claim that algorithms that use static data distribution based only on spatial considerations are presenting "efficiency" numbers that can be inaccurate, maybe by a large margin.

To avoid the pitfalls of normal static data distribution, we present in the next section a new way to achieve realistic load balancing. Our load balancing scheme, does not scale linearly, but achieves very fast rendering times while minimizing the "work" done by the processors.

4.6.2 Content-Based Load Balancing

This section explains our approach to load balancing, which is able to achieve accurate load balancing even when using presence acceleration optimizations. The original idea of our load balancing technique came from the PARC [5] acceleration technique. We notice that the amount of "work" performed by a presence accelerated ray caster is roughly directly proportional to the number of *full supervoxels* contained in the volume.

We use the number of full supervoxels a given processor is assigned as the measure of how much work is performed by that particular processor. Let P denote the number of processors, and c_i the number of full supervoxels processor i has. In order to achieve a good load balancing (by our metric) we need a scheme that *minimizes* the following function for a partition $X = (c_1, c_2, \ldots)$:

$$f(X) = \max_{i \neq j} |c_i - c_j|, \forall i, j \leq P \tag{11}$$

Equation 11 is very general and applies to any partition of the dataset D into disjoint pieces D_i. In our work we have tried to solve this optimization problem in a very restricted context. We have assumed that each D_i is convex. (We show later that this assumption makes it possible to create a *fixed* depth sorting network for the partial rays independently calculated each disjoint region.) Furthermore, we only work with two very simple subdivisions: slabs and a special case of a BSP-tree.

Before we go any further, it is interesting to study the behavior of our load balancing scheme in the very simple case of a slab subdivision of the volume D. Slabs (see Figure 4) are consecutive slices of the dataset aligned on two major axes. Given a volume D, with s *superslices* and p processors with the restriction that each processor gets contiguous slices, the problem of calculating the "best" load balancing partition for p processors consists of enumerating all the $(s-1)(s-2)\ldots(s-p+1)$ ways of partitioning D, and choosing the one that *minimizes* Equation 11.

The problem of computing the optimal (as defined by our heuristic choice) load balance partition indices can be solved naively as follows. We can compute all the possible partitions of the integer n, where n is the number of slabs, into P numbers, where P is

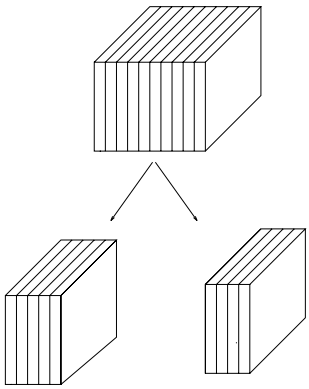


Figure 4: *During slab-based load balancing, each processor gets a range of continuous data set slabs. The number of full supervoxels determines the exact partition ratio.*

the number of processors (it is actually a bit different, as we need to consider addition non-associative). For example, if $n = 5$, and $P = 3$, then $1 + 1 + 3$ represents the solution that gives the first slab to the first processor, the second slab to the second processor and the remaining three slabs to the third processor. Enumerating all possible partitioning to get the optimal one is a feasible solution but can be very computationally expensive for large n and P. We use a slightly different algorithm for the computations that follows, we choose the permutation with the smallest square difference from the average.

In order to show how our approach works in practice, let us work out the example of using our load balancing example to divide the *neghip* dataset (the negative potential of a high-potential iron protein of 66^3 resolution) for four processors. Here we assume the number of superslices to be 16, and the number of supervoxels to be 64 (equivalent to a level 4 PARC decomposition). Using a voxel threshold of 10-200 (out of a range up to 255), we get the following 16 supervoxel count for each slab, out of the 1570 total full supervoxels:

12, 28, 61, 138, 149, 154, 139, 104, 106, 139, 156, 151, 129, 62, 29, 13

A naive approach load balancing scheme would assign regions of equal volume to each processor resulting in the following partition:

12 + 28 + 61 + 138 = 239
149 + 154 + 139 + 104 = 546
106 + 139 + 156 + 151 = 552
129 + 62 + 29 + 13 = 233

Here processors 2 and 3 have twice as much "work" as processors 1 and 4. Using our metric, we get:

12 + 28 + 61 + 138 + 149 = 388
154 + 139 + 104 = 397
106 + 139 + 156 = 401
151 + 129 + 62 + 29 + 13 = 384

One can see that some configurations will yield better load balancing than others but this is a limitation of the particular space subdivision one chooses to implement, the more complex the subdivision one allows, the better load balancing but the harder it is to implement a suitable load balancing scheme and the associated ray caster. Figure 5 plots the examples just described for the naive approach. Figure 6 shows how well our load balancing scheme works for a broader set of processor arrangements.

These shortcomings of slabs let us to an alternative space decomposition structure previously used by Ma et al. [48, 49], the *Binary Space Partition* (BSP) tree, originally introduced by Fuchs et al. [29].

4.6.3 The *Parallel Ray Casting* Rendering Pipeline

Compositing Cluster

The compositing nodes are responsible for regrouping all the sub-rays back together in a consistent manner, in order to keep image correctness. This is only possible because composition is an associative operation, so if we have to sub-ray samples where one ends where the other starts, it is possible to combine their samples into one sub-ray recursively until we have a value that constitutes the full ray contribution to a pixel.

Ma et al. [49] use a different approach to compositing, where instead of having separate compositing nodes, the rendering nodes switch between rendering and compositing. Our method is more efficient because we can use the special structure of the

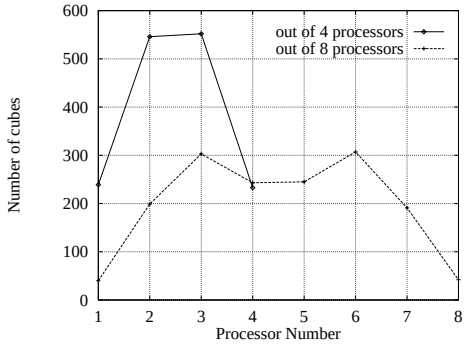


Figure 5: *The graph shows the number of cubes per processor under naive load balancing.*

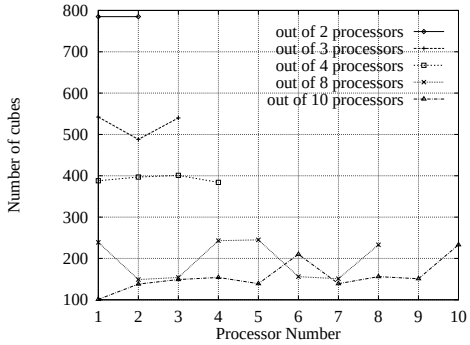


Figure 6: *Load balancing measures for our algorithm. The graph shows the number of cubes the processor receives in our algorithm.*

sub-ray composition to yield a high performance pipeline, where multiple nodes are used to implement the complete pipeline (see Figure 12). Also, the structure of compositing requires synchronized operation (e.g., there is an explicit structure to the composition, that needs to be guaranteed for correctness purposes), and light weight computation, making it much less attractive for parallelization over a large number of processors, specially on machines with slow communication compared to CPU speeds (almost all current machines).

It is easy to see that compositing has a very different structure than rendering. Here, nodes need to synchronize at every step of the computation, making the depth of the compositing tree a hard limit on the speed of the rendering. That is, if one uses 2^m nodes for compositing, and it takes t_c time to composite two images, even without any synchronization or communication factor in, it takes at least mt_c time to get a completely composited image.

Fortunately, most of this latency can be hidden by pipelining the computation. Here, instead of sending one image at a time, we can send images continuously into the compositing cluster, and as long as we send images at a rate lower than one for every

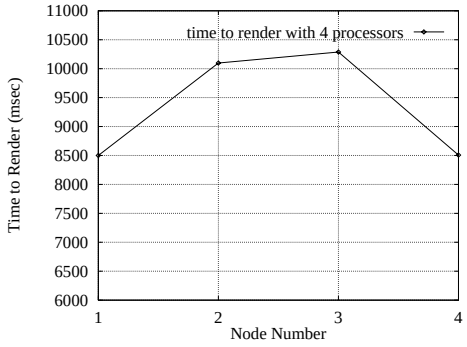


Figure 7: *Naive load balancing on the Paragon. The graph shows the actual rendering times for 4 processors using the naive load balancing.*

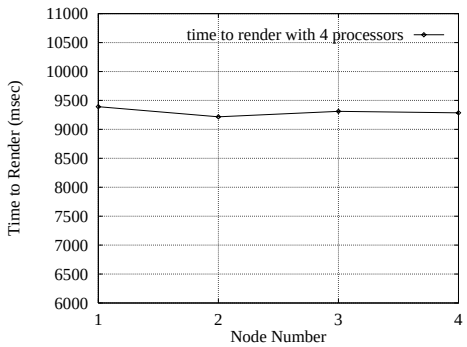


Figure 8: *Our load balancing on the Paragon. The graph shows the actual rendering times for 4 processors using our load balancing.*

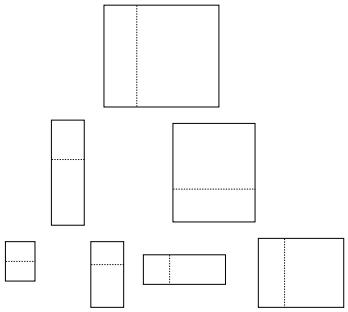


Figure 9: *An example of the partition scheme we used for load balancing. The bottom represents a possible decomposition for 8 nodes. Notice that a cut can be made several times over the same axis to optimize the shape of the decomposition.*

t_c worth of time, the compositing cluster is able to composite those at full speed, and after mt_c times, the latency is fully hidden from the computation. As can be seen for our discussion, this latency hiding process is very sensitive to the rate of images coming in the pipeline. One needs to try to avoid "stalls" as much as possible. Also, one can not pipe more than the overall capacity of the pipeline.

Several implications for real-time rendering can be extracted from this simple model. Even though the latency is hidden from the computation, it is not hidden from the user, at least not totally. The main reason is the overall time that an image takes to be computed. Without network overhead, if an image takes t_r time to be rendered by the rendering cluster, the first image of a sequence takes (at least) time $t_r + mt_c$ to be received by the user. Given that people can notice even very small latencies, our latency budget for real-time volume rendering is extremely low and will definitely have to wait for the next generation of machines to be build. We present a detailed account of the timings later in this chapter.

Going back to the previous discussion, we see that as long as t_r is larger than t_c we don't have anything to worry about with

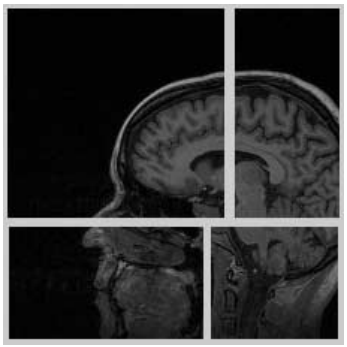


Figure 10: *A cut through the partition accomplished using our load balancing scheme on an MRI head. It is easy to see that if a regular partition scheme were used instead, as the number of processors increase, large number of processors would get just empty voxels to render.*

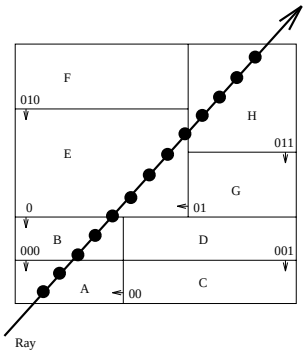


Figure 11: *Data partitioning shown in two dimensions. The dataset is partitioned into 8 pieces (marked A . . . H) in a canonical hierarchical manner by the 7 lines (planes in 3D) represented by binary numbers. Once such a decomposition is performed, it is relatively easy to see how the samples get composited back into a single value.*

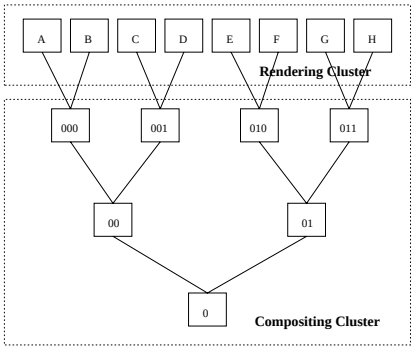


Figure 12: *The internal structure of one compositing cluster, one rendering cluster and their interconnection is shown. In PVR, the communication between the compositing and the rendering clusters is very flexible, with several rendering clusters being able to work together in the same image. This is accomplished by using a set of tokens that are handled by the first level of the compositing tree in order to guarantee consistency. Because of its tree structure, one properly synchronized compositing cluster can work on several images at once, depending on its depth. The compositing cluster shown is relative to the decomposition shown in Figure 11.*

respect to creating a bottleneck in the compositing end. As it turns out, t_r is much larger than t_c, even for relatively small datasets. With this in mind, an interesting question is how to allocate the compositing nodes, with respect to size and topology.

The topology is actually fixed by the corresponding BSP-tree, that is, if the first level of the tree has $n = 2^h$ images (if one image per rendering node, than n would be the number of rendering nodes), than potentially the number of compositing nodes required might be as high as $2^h - 1$. There are several reasons not to use that many compositing nodes. First, it is a waste of processors. Second, the first-image latency grows with the number of processors in the compositing tree. Fortunately, we can lower the number of nodes required in the compositing tree by a process known as virtualization. A general solution to this problem is proposed in Section 4.9.

Types of Parallelism

Due to the fact that each rendering node gets a portion of the dataset, this type of parallelism is called "object-space parallelism". The structure of our rendering pipeline makes it possible to exploit other types of parallelism. For instance, by using more than a single rendering cluster to compute an image, we are making use of "image-space parallelism" (in PVR, it is possible to specify that each cluster compute disjoint scanlines of the same image; see [86] for the issues related to image-space parallelism). The clustering approach coupled with the inherent pipeline parallelism available in the compositing process (because of its recursive structure) gives rise to a third parallelism type, namely "time-space parallelism" or "temporal parallelism". In the latter, we can exploit multiple clusters by concurrently calculating sub-rays for several images at once, that can be sent down the compositing pipeline concurrently. Here, it is important for the correctness of the images, that each composition step be done in lockstep, in order to avoid mixing of images. It should be clear by now that there are several advantages to our separation of nodes into our two types.

4.7 Lazy Sweep Ray Casting Algorithm

Lazy Sweep Ray Casting is a fast algorithm for rendering general irregular grids. It is based on the sweep-plane paradigm, and it is able to accelerate ray casting for rendering irregular grids, including disconnected and non-convex (even with holes) unstructured irregular grids with a rendering cost that decreases as the "disconnectedness" decreases. The algorithm is carefully tailored to exploit spatial coherence even if the image resolution differs substantially from the object space resolution.

Lazy Sweep Ray Casting has several desirable properties, including its generality, (depth-sorting) accuracy, low memory consumption, speed, simplicity of implementation and portability (e.g., no hardware dependencies).

The design of our LSRC method for rendering irregular grids is based on two main goals: (1) the depth ordering of the cells should be correct along the rays corresponding to every pixel; and (2) the algorithm should be as efficient as possible, taking advantage of structure and coherence in the data. With the first goal in mind, we chose to develop a new ray casting algorithm, in order to be able to handle cycles among cells (a case causing difficulties for projection methods). To address the second goal, we use a sweep approach, as did Giertsen [31], in order to exploit both *inter-scanline* and *inter-ray* coherence. Our algorithm has the following advantages over Giertsen's:

(1) It avoids the explicit transformation and sorting phase, thereby avoiding the storage of an extra copy of the vertices;

(2) It makes no requirements or assumptions about the level of connectivity or convexity among cells of the mesh; however, it does take advantage of structure in the mesh, running faster in cases that involve meshes having convex cells and convex components;

(3) It avoids the use of a hash buffer plane, thereby allowing accurate rendering even for meshes whose cells greatly vary in size;

(4) It is able to handle parallel and perspective projection within the same framework (e.g, no explicit transformations).

4.7.1 Performing the Sweep

Our sweep method, like Giertsen's, sweeps space with a sweep-plane that is orthogonal to the viewing plane (the x-y plane), and parallel to the scanlines (i.e., parallel to the x-z plane). See Figure 13.

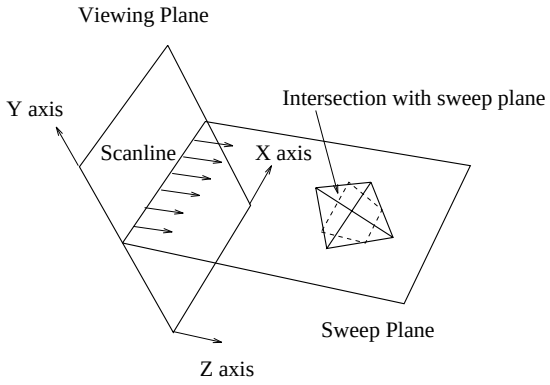


Figure 13: A sweep-plane (perpendicular to the y-axis) used in sweeping 3-space.

Events occur when the sweep-plane hits vertices of the mesh S. But, rather than sorting all of the vertices of S in advance, and placing them into an auxiliary data structure (thereby at least doubling the storage requirements), we maintain an event queue (priority queue) of an appropriate subset of the mesh vertices.

A vertex v is *locally extremal* (or simply *extremal*, for short) if all of the edges incident to it lie in the (closed) halfspace above or below it (in y-coordinate). A simple (linear-time) pass through the data readily identifies the extremal vertices.

We initialize the event queue with the extremal vertices, prioritized according to the magnitude of their inner product (dot product) with the vector representing the y-axis ("up") in the viewing coordinate system (i.e., according to their y-coordinates). We do *not* explicitly transform coordinates. Furthermore, at any given instant, the event queue only stores the set of extremal vertices not yet swept over, plus the vertices that are the upper endpoints of the edges currently intersected by the sweep-plane. In practice, the event queue is relatively small, usually accounting for a very small percentage of the total data size. As the sweep takes place, new vertices (non-extremal ones) will be inserted into and deleted from the event queue each time the sweep-plane hits a vertex of S.

The sweep algorithm proceeds in the usual way, processing events as they occur, as determined by the event queue and by the scanlines. We pop the event queue, obtaining the next vertex, v, to be hit, and we check whether or not the sweep-plane encounters v before it reaches the y-coordinate of the next scanline. If it does hit v first, we perform the appropriate insertions/deletions on the event queue; these are easily determined by checking the signs of the dot products of edge vectors out of v with the vector representing the y-axis. Otherwise, the sweep-plane has encountered a scanline. And at this point, we stop the sweep and drop

into a two-dimensional ray casting procedure (also based on a sweep), as described below. The algorithm terminates once the last scanline is encountered.

We remark here that, instead of doing a sort (in y) of all vertices of S at once, the algorithm is able to take advantage of the partial order information that is encoded in the mesh data structure. (In particular, if each edge is oriented in the $+y$ direction, the resulting directed graph is acyclic, defining a partial ordering of the vertices.) Further, by doing the sorting "on the fly", using the event queue, our algorithm can be run in a "lock step" mode that avoids having to sort and sweep over highly complex subdomains of the mesh. This is especially useful, as we see in the next section, if the slices that correspond to our actual scanlines are relatively simple, or the image resolution (pixel size) is large in comparison with some of the features of the dataset. (Such cases arise, for example, in some applications of scientific visualization on highly disparate datasets.)

4.7.2 Processing a Scanline

When the sweep-plane encounters a scanline, the current sweep status data structure gives us a "slice" through the mesh in which we must solve a two-dimensional ray casting problem. (See Figure 14.) Let $\mathcal{S}$ denote the polygonal (planar) subdivision at the current scanline (i.e., $\mathcal{S}$ is the subdivision obtained by intersecting the sweep-plane with the mesh S.) In time linear in the size of $\mathcal{S}$, we can recover the subdivision $\mathcal{S}$ (both its geometry and its topology), just by stepping through the sweep status structure, and utilizing the local topology of the cells in the slice. In our implementation, $\mathcal{S}$ is actually not constructed explicitly, but only given implicitly by the sweep status data structure, and then *locally* reconstructed as needed during the two-dimensional sweep (described below).

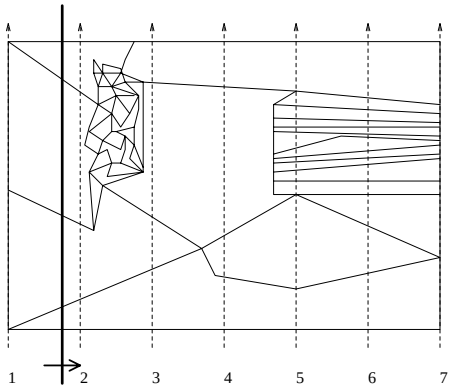


Figure 14: Illustration of a sweep in one slice.

The two-dimensional problem is also solved using a sweep algorithm — now we sweep the plane with a sweep-line parallel to the z axis. Events now correspond to vertices of the planar subdivision $\mathcal{S}$. At the time that we construct $\mathcal{S}$, we could identify those vertices in the slice that are locally extremal in $\mathcal{S}$ (i.e., those vertices that have edges only leftward in x or rightward incident on them.), and insert them in the initial event queue. (The actual implementation just sorts along the x-axis, since the extra memory overhead is negligible in 2D.) The *sweep-line status* is an ordered list of the edges of $\mathcal{S}$ crossed by the sweep-line. The sweep-line status is initially empty. Then, as we pass the sweep-line over $\mathcal{S}$, we update the sweep-line status and the event queue at each event when the sweep-line hits an extremal vertex, making insertions and deletions in the standard way. This is analogous to the Bentley-Ottmann sweep that is used for computing line segment intersections in the plane [72]. We also stop the sweep at each of the x-coordinates that correspond to the rays that we are casting (i.e., at the pixel coordinates along the current scanline), and output to the rendering model the sorted ordering (depth ordering) given by the current sweep-line status. We have noticed that the choice of data structure used to maintain the sweep-line status can have a dramatic impact on the performance of the algorithm.

See Silva and Mitchell [85] for details.

4.8 Parallel Rendering of Irregular Grids

Here, we present a distributed-memory MIMD machine parallelization of the LSRC method. Our parallelization is a distributed-memory parallelization, and each rendering node gets a only portion of the dataset, not the complete dataset.

The need for the parallelization of rendering algorithms for irregular-grid rendering is obvious, given the fact that irregular grids are extremely large (as compared to regular grids), and their rendering is much less efficient. The largest irregular grids currently being rendered are just breaking the 1,000,000 cell barrier, what would be equivalent to a 100-by-100-by-100 regular grid, if only data sample points are taken into account. On the other hand, such a grid requires more than 50MB of memory, when its regular counterpart only needs 1MB. Actually, regular grids of this size can be rendered by inexpensive workstations in real-time (i.e., using the Shear-Warp technique), while the irregular grids of this size would be almost out of reach just a year ago.

Actually, the sizes of irregular grids of interest of computational scientists are larger than one million cells, possibly starting at two times that range. (This is subjective data, obtained by talking to researchers at Sandia National Labs during the summer of 1996). Given that it takes us about 150 seconds to render a 500,000 cell complex, and assuming linear behavior (what is not

completely correct) it would takes us over 10 minutes to generate images of a 2,000,000 cell complex. What is not an unreasonable amount of time, given that Ma [50] needed over 40 minutes to render a dataset over 10 times smaller.

But our goal is to develop a method that is both faster and scalable to larger and larger dataset. The main reason for this trust is not really current dataset, but those upcoming ones, specially from the new breed of supercomputers, such as the ASCI TeraFlop machine installed at Sandia National Labs. The ASCI machine has orders of magnitude more memory than the current Intel Paragon installed there, even more *usable* memory (i.e., not taking OS and network overhead into account). This will enable the generation of extremely large grids, possibly in ranges of 10,000,000-100,000,000 cells or larger.

Part of this increase in dataset sizes can be offset by better algorithms, specially by further improvements in our rendering code by complete implementation of our optimization ideas. But our experience with irregular grids, seems to show that only more computing power can really offset the increase in dataset size.

The other main reason for the use of parallel machines comes from the pure size of the datasets. The largest workstations available to us have 1GB–3GB of memory, what is very short of the 300GB–512GB of memory in the ASCI Tflop machine. Several reasons indicate the visualization should be performed locally: the fact that very few workstations with more than a few gigabytes of memory are available; moving 300GB of data in and out at ethernet, or even ATM OC-3 speeds is clearly infeasible; disk transfer rates, even for reasonably large (and expensive) disk arrays are just too slow for this kind of data.

As all of the reasons pointed above for the use of the parallel machines that generated the dataset is not enough, we also need to note that these simulations do not generate a single static volume, but in general, time dependent data is being generated and the time steps can not, in general, be efficiently accessed (for obvious reasons).

With all of this in mind, we present our algorithm for rendering irregular grid data, in place, on distributed-memory MIMD machines.

4.8.1 Previous Work

There has been very little work on rendering irregular grid data on distributed memory architectures. Overall parallel work on rendering irregular grids has received relatively little attention. This might be due to the fact that rendering irregular grids is so much harder than regular grids, that few people ever get to the point of being able to research parallel methods for irregular grids.

Uselton has parallelized his original ray tracing work (presented in [91]) in a shared memory multiprocessor SGI, and reported that the implementation scales linearly up to 8 processors. Challinger [10] reports on a parallel algorithm for irregular grids, implemented on a shared-memory BBN-2000 Butterfly. Giertsen [30] has also parallelized his sweep algorithm on a collection of IBM RS/6000, using a master/slave scheme and total data replication in the nodes.

The most interesting work, by our perspective, is Ma [50], where a parallelization technique very similar to the one presented here is proposed. It is unfortunate that he used a sequential ray casting technique that is shown to be at least two orders of magnitude slower than the one we use. Because of this, he did not find any interesting bottlenecks of the parallelization technique.

His technique works by breaking up the original grid into multiple, disjoint cell complexes using Chaco [35], a graph-based decomposition tool developed at Sandia National Labs. Chaco-based decompositions have several interesting and important properties for parallelization of computational methods. It is unclear, the extra overhead of using Chaco has actually any influence on the rendering speed of the parallelization. Here, as in our parallel regular grid method presented in Section 4.6, we divide the nodes into two classes: rendering and compositing nodes. The rendering nodes, compute each ray of an image, creating a set of stencils (the rays may not be completely connected). After each ray is computed, they are sent to the compositing nodes for further sorting and the final accumulation. Each compositing node is assigned a set of rays to be composited. He reports that because the rendering takes so long, the compositing phase is negligible and he has not work any further on optimizing it.

In a later paper, Ma and Crockett [51] proposed a technique for rendering on irregular grids which sub-partions the data grid into sub regions. After the parallel rendering of the each sub region – which includes the projection of the cells of the region and the local compositing within that region –, the regions are composed into the final image.

4.8.2 Parallel LSRC

Overall the our algorithm is very similar to Ma's. Continuing in the tradition of our regular grid work and the framework of our PVR system, we divide the nodes into two relevant groups, rendering and compositing nodes. Our differences between our work and Ma's are actually in the details of the rendering and compositing.

Dataset Decomposition

In order to subdivide the dataset among the nodes, we use a hierarchical decomposition method, with a similar flavor to our load balancing scheme for regular grids. Starting with the bounding box of the complete cell complex, we start making cuts in this box, taking two things into account: the aspect ratio of the cuts, and the number of vertices. At every step, we cut along the largest axis in such a way as to break the number of vertices in half, in each stage of the cutting. because cells might belong to more a single of these convex space decomposition "boxes", we assign the cell to the box that has most of it (e.g., in the number of vertices, with ties broken in some arbitrary, but consistent way).

The obvious now, is just to assign each processor to a each box. This is a way to minimize the total rendering time of the complete irregular grid. Unfortunately, it is not clear that this is the right thing, given that one might want to create a rough picture of the grid fast, then wait for more complex rendering. In the future we expect to be able to create a scattered decomposition, that will have better properties in creating approximate renderings of the grids.

With the decomposition method just proposed, each processor should have roughly the same number of primitives, each of which, approximately confined to a rectangular grid of almost bounded aspect ratio (because of the largest-axis cutting).

Rendering

The rendering performed at each node is just a variation of our sequential technique presented in Section 4.7. This is just a single significant difference, instead of generating an image, every node generates a *stencil* data-structure. Of course, all nodes work concurrently on generating stencil scan-lines.

The stencil representation of a scan-line is just a linked-list of color and depth of cells, who have been lazily composited. That is, if two stencils shared an end point (e.g., $(\vec{a}, \vec{b})$ and $(\vec{b}, \vec{c})$), they are composited into a single stencil $(\vec{a}, \vec{c})$, representing the whole region. In the end of a scan-line rendering computation, each node potentially has a collection of stencils. Because of the process of decomposing the dataset among the nodes, it is expected the stencil fragmentation is low. This is necessary in order to enable fast communication for compositing.

Compositing

One solution for compositing would just to copy Ma's technique, where nodes are responsible for certain scan-lines. This way the rendering nodes could just send its collection of stencils for further sorting in the compositing nodes. In our case, we try to achieve better performance by creating a tree of compositing nodes (such as the one we use for the regular case). Every compositing node is responsible for a certain region of space (i.e., one of the original box decompositions proposed above), that belongs to a global BSP-tree.

It is the responsibility of the rendering nodes to respect the BSP-tree boundaries and send the data to the correct compositing nodes, possibly breaking stencils that are span across boundaries.

Once the data of each scan-line is received in the compositing nodes, the final depth sorting can be efficiently performed by merging the stencils into a complete image. An efficient pipeline scheme can be implemented on a scan-line by scan-line basis, with similar good properties as the one implemented image-by-image for the regular grid case.

4.9 General BSP-tree Compositing

A simple way of parallelizing rendering algorithms is to do it at the object-space level: *i.e.*, divide the task of rendering different objects among different rendering processors, and then compose the full images together. A large class of rendering algorithms (although not all), in particular scan-line algorithms, can be parallelized using this strategy. Such parallel rendering architectures, where renderers operate independently until the visibility stage, are called *sort-last* (SL) architectures [60]. A fundamental advantage of SL architecture is the overall simplicity, since it is possible to parallelize a large class of existing rendering algorithms without major modifications. Also, such architectures are less prone to load imbalance, and can be made linearly scalable by using more renderers [58, 59]. One shortcoming of SL architectures is that very high bandwidth might be necessary, since a large number of pixels have to be communicated between the rendering and compositing processors. Despite the potential high bandwidth requirements, sort-last has been one of the most used, and successful parallelization strategies for both volume rendering and polygon rendering, as shown by the several works published in the area [14, 97, 43, 49].

Here we present a general purpose, optimal compositing machinery that can be used as a black box for efficiently parallelizing a large class of sort-last rendering algorithms. We consider sort-last rendering pipelines that are based on separating the rendering processors from the compositing processors, similar to what was proposed previously by Molnar [58]. The techniques described in this paper optimize overall performance and scalability without sacrificing generality or the ease of adaptability to different renderers. Following Molnar, we propose to use a scan-line approach to image composition, and to execute the operations in a pipeline as to achieve the highest possible frame rate. In fact, our framework inherits most of the salient advantages of Molnar's technique. The two fundamental differences between our pipeline and Molnar's are:

(1) instead a fixed network of Z-buffer compositors, our approach uses a user-programmable BSP-tree based composition tree;

(2) we use general purpose processors and networks, instead of Molnar's special purpose Z-comparators arranged in a tree.

In our approach, hidden-surface elimination is not performed by Z-buffer alone, but instead by executing a BSP-tree model. This way, we are able to offer extra flexibility, and instead of only providing parallelization of simple depth-buffer scan-line algorithms, we are able to provide a general framework that adds support for true transparency, and general depth-sort scan-line algorithms. In trying to extend the results of Molnar to general purposes parallel machines, we must deal with a processor allocation problem. The basic problem is how to minimize the amount of processing power devoted to the compositing back-end and still provide performance guarantees (*i.e.*, frame rate guarantees) for the user. We propose a solution to this problem in the paper.

In our framework the user defines a BSP-tree, in which the leaves correspond to renderers (the renderers perform user-defined rendering functions). Also, the user defines a data structure for each pixel, and a compositing function, that will be applied to each pixel by the internal nodes of the BSP-tree previously defined. Given a pool of processors to be used for the execution of the compositing tree, and a minimum required frame rate, our processor allocation algorithm partitions the compositing operations among processors. The partition is chosen so as to minimize the number of processors without violating the frame-rate needs. During rendering, the user just needs to provide a viewpoint (actually, for optimum performance, a sequence of viewpoints, since our algorithm exploits pipelining). Upon *execution* of the compositing tree, messages are sent to the renderers specifying where to

send their images, so no prior knowledge of the actual compositing order is necessary on the (user) rendering nodes side. For each viewpoint provided, a *complete* image will be generated, and stored at the processor that was allocated the root of the compositing tree. The system is fully pipelined, and if no stalls are generated by the renderers, our system guarantees a frame rate at which the user can collect the full images from the root processor.

4.9.1 Optimal Partitioning of the Compositing Tree

We can view the BSP tree as an expression tree, with compositing being the only operation. In our model of compositing clusters, evaluation of the compositing expression is mapped on to a *tree* of *compositing processes* such that each process evaluates exactly one sub-expression. See Figure 15 for an illustration of such a mapping. The actual ordering of compositing under a BSP-tree depends not only on the position of the nodes, but also on the viewing direction. So, during the execution phase, a specific ordering has to be obeyed. Fortunately, given any partition of the tree, each subtree can still be executed independently. Intuitively, correctness is achieved by having the nodes "fire up" in a on-demand fashion.

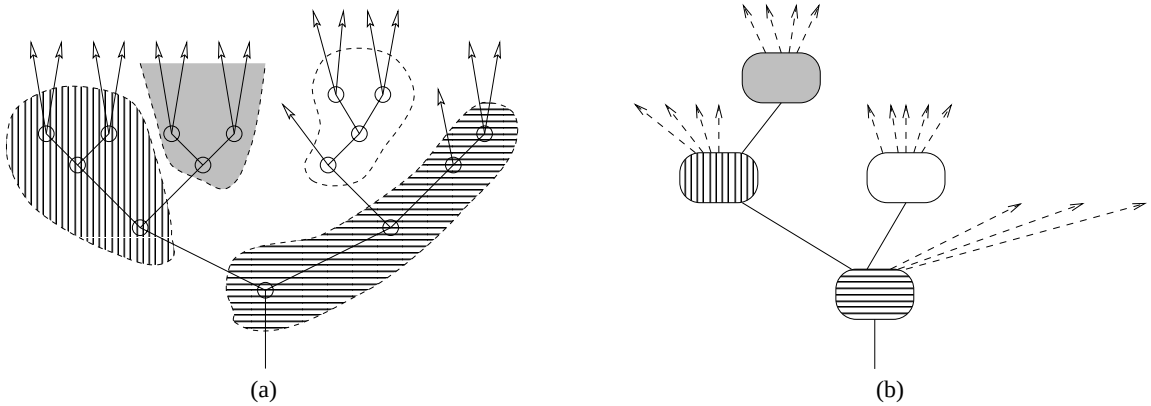


Figure 15: (a) A BSP tree, showing a grouping of compositing operations and (b) the corresponding tree of compositing processes. Each compositing process can be mapped to a different physical node in the parallel machine.

Such a decompositing is based on a model of the cost of the subtrees. For details on this, and the partitioning algorithm, shown in Figure 16, see Ramakrishnan and Silva [73].

Practical Considerations

Algorithm *partition* provides a simple way of given a BSP-tree, and a performance requirement, given in terms of the frame rate, how to divide up the tree in such a way as to optimize the use of processors. Several issues, including machine architecture bottlenecks, such as synchronization, interconnection bandwidth, mapping the actual execution to a specific architecture (*e.g.*, a mesh-connected MIMD machine) were left out of the previous discussion. We now describe how Algorithm *partition* can be readily adapted to account for some of the above issues in practice.

Compositing Granularity: Note that there is nothing in the model that requires that full images be composited and transfered one at a time. Actually, one should take into consideration when determining the unit size of work, and communication, hardware constraints such as memory limitations, and bandwidth requirements. So, for instance, instead of messages being a full image, it might be better to send a pre-defined number of scan-lines. Notice that in order for images of arbitrary large size to be able to be computed, the rendering algorithm must also be able to generate the images in scan-line order.

Communication Bandwidth: Of course, in order to achieve the desired frame rate, enough bandwidth for distributing the images during composition is strictly necessary. Given p processors, each performing k compositing operations, the overall aggregate bandwidth required is proportional to $p(k+2)$. It should be clear that as k_{max} increases, the actual bandwidth requirement actually decreases (both for the case of a SL-full, as well as a SL-sparse architecture) since as k_{max} increases the number of processors required decreases. This decrease in bandwidth is due to the fact that compositing computation are performed locally, inside each composite processor, instead of being sent over the network. If one processor performs exactly k_{max} compositing operations, it needs $k_{max} + 2$ units of bandwidth, as opposed to $3k_{max}$ when using one processor per compositing operation— a bandwidth savings of almost a factor of three!

Another interesting consideration related to bandwidth is the fact that our messages tend to be large, implying that our method operates on the best range of the message size versus communication bandwidth curve. For instance, for messages smaller than 100 bytes the Intel Paragon running SUNMOS achieve less than 1 MB/sec bandwidth, while for large messages (i.e., 1MB or larger), it is able to achieve over 160MB/sec. (This is very close to 175MB/sec, which is the peak hardware network performance of the machine.) As will be seen in Section 4.9.2, our tree execution method is able to completely hide the communication latency, while still using large messages for its communication.

```
Algorithm partition(u)
    /* The algorithm marks the beginning of partitions in
       the subtree of G rooted at u. If more vertices,
       can be added to the root partition, the algorithm
       returns the size of the root partition.
       Otherwise, the algorithm returns 0. */
1.  if (arity(u) = 2) then    /* u is a binary vertex */
2.      w1 := partition(left_child(u));
3.      w2 := partition(right_child(u));
4.      w := w1 + w2 + 1;
5.      if (w > K) then
6.          if (w1 ≤ w2) then
7.              Mark right_child(u) as start of new partition
8.              w := w1 + 1;
9.          else
10.             Mark left_child(u) as start of new partition
11.             w := w2 + 1;
12. else if (arity(u) = 1) then    /* u is a unary vertex */
13.     w := partition(child(u)) + 1;
14. else    /* u is a leaf */
15.     w := 1;
16. if (w = K) then
17.     Mark u as a start of new partition
18.     return(0);
19. else
20.     return(w);
```

Figure 16: Algorithm *partition*

Latency and Subtree Topology: As will be seen in Section 4.9.2, the whole process is pipelined, with a request-based paradigm. This greatly reduces the overhead of any possible synchronization. Actually, given enough compositing processors, the overall time is only dependent on the performance of the rendering processors. Also, note that the actual *shape* of the subtree that a given processor gets is irrelevant, since the execution of the tree is completely pipelined.

Architectural Topology Mapping: We do not provide any mechanism for optimizing the mapping from our tree topology to the actual processors in a given architecture. With recent advancements in network technology, it is much less likely that the use of particular communication patterns improve the performance of parallel algorithms substantially. In new architectures, the point-to-point bandwidth in access of 100–400 MB/sec are not uncommon[1], while in the old days of the Intel Delta, it was merely on the order of 20 MB/sec. Also, network switches, with complex routing schemes, are less likely to make neighbor communication necessary. (Actually, the current trend is not to try to exploit such patterns since new fault-handling and adaptive routers usually make such tricks useless.)

Limitations of Analytical Cost Model: Even though we can support both SL-full and SL-sparse architecture, our model does not make any distinction of the work that a given compositing processor is performing based on the depth of its compositing nodes. This is one of the limitations of our analytical formulation. However, the experimental results indicate that this limitation does not seem have any impact on the use of our partitioning technique in practice. Actually, frame-to-frame differences might diminish the concrete advantage of techniques that try to optimize for this fact.

4.9.2 Optimal Evaluation

In the previous section, we described techniques to partition the set of compositing operations and allocate one processor to each partition, such that the various costs of the compositing pipeline can be minimized. We now describe efficient techniques for performing the compositing operations within each processor.

Space-Optimal Sequential Evaluation of Compositing Trees

Storage is the most critical resource for evaluating a compositing tree. We need 4MB of memory to store an image of size 512×512, assuming 4-bytes each for RGB and α values per pixel. Naive evaluation of a compositing tree with N nodes may require intermediate storage for up to N images.

We now describe techniques, adapted from register allocation techniques used in programming language compilation, to minimize the total intermediate storage required. Figure 17a shows a compositing tree for compositing images I_1 through I_6. We can

[1] These numbers are already outdated by now.

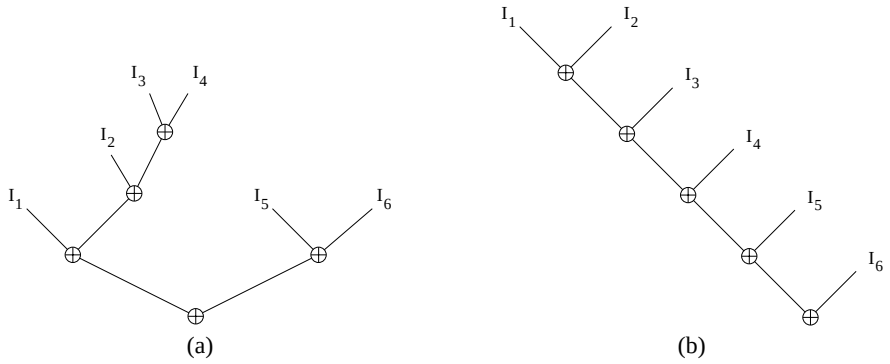


Figure 17: (a) A compositing tree and (b) its corresponding associative tree.

consider the tree as representing the expression

$$(I_1 \oplus (I_2 \oplus (I_3 \oplus I_4))) \oplus (I_5 \oplus I_6) \tag{12}$$

where $\oplus$ is the compositing operator. Since images I_1 through I_6 are obtained from remote processors, we need to copy these images locally into intermediate buffers before applying the compositing operator. The problem now is to sequence these operations and reuse intermediate buffers such that the total number of buffers needed for evaluating the tree is minimized.

We encounter a very similar problem in a compiler, while generating code for expressions. Consider a machine instruction (such as integer addition) that operates only on pairs of registers. Before this operation can be performed on operands stored in the main memory, the operands must be loaded into registers. We now describe how techniques to generate optimal code for expressions can be adapted to minimize intermediate storage requirements of a compositing process. The number of registers needed to evaluate an expression tree can be minimized, using a simple tree traversal algorithm [2, pages 561–562]. Using this algorithm, the compositing tree in Figure 17a can be evaluated using 3 buffers. In general, $O(\log N)$ buffers are needed to evaluate a compositing tree of size N. However, by exploiting the algebraic properties of the operations, we can further reduce the number of buffers needed— to $O(1)$. Since $\oplus$ is associative, evaluating expression (12) is equivalent to evaluating the expression:

$$((((I_1 \oplus I_2) \oplus I_3) \oplus I_4) \oplus I_5) \oplus I_6 \tag{13}$$

The above expression is represented by the compositing tree in Figure 17b, called an *associative tree* [81]. The associative tree can be evaluated using only 2 buffers.

Again, for full details, we refer the reader to the full paper [73].

4.9.3 Implementation

In this section, we sketch the implementation of our compositing pipeline. We implemented our compositing back-end in the PVR system [84]. PVR is a high-performance volume rendering system, and it is freely available for research purposes. Our main reason for choosing PVR was that it already supported the notion of separate rendering and compositing clusters, as explained in [83, Chapter 3]. The basic operation is very simple. Initially, before image computation begins, all compositing nodes receive a BSP-tree defining the compositing operations based on the object space partitioning chosen by the user. Each compositing node, in parallel, computes its portion of the compositing tree, and generates a view-independent data structure for its part. Image calculation starts when all nodes receive a sequence of viewpoints.

The rendering nodes, simply run the following simple loop:

```
For each (viewpoint v)
  ComputeImage(v);
  p = WaitForToken();
  SendImage(p);
```

Notice that the rendering nodes do not need any explicit knowledge of parallelism; in fact, each node does not even need to know, *a priori*, where its computed image is to be sent. Basically, the object space partitioning and the BSP-tree takes care of all the details of parallelization.

The operation of the compositing nodes is a bit more complicated. First, (for each view) each compositing processor computes (in parallel, using its portion of the compositing tree) an array with indices of the compositing operations assigned to it as a sequence of processor numbers from which it needs to fetch and compose images. The actual execution is basically an implementation of the pre-fetching scheme proposed here, with each `read_request` being turned into a `PVR_MSG_TOKEN` message, where the value of the token carries its processor id. So, the basic operation of the compositing node is:

```
For each (viewpoint v)
  CompositeImages(v);
  p = WaitForToken();
  SendImage(p);
```

Notice that there is no explicit synchronization point in the algorithm. All the communication happens bottom-up, with requests being sent as early as possible (in PVR, tokens are sent asynchronously, and in most cases, the rendering nodes do not wait for the tokens), and speed is determined by the slowest processor in the overall execution, effectively pipelining the computation. Also, one can use as many (or as few) nodes one wants for the compositing tree. That is, the user can determine the rendering performance for a given configuration, and based on the time to composite two images it is straightforward simple to scale our compositing back-end for his particular application.

Acknowledgments

Numerous individuals have contributed to all parts of the material presented here. In particular, we like to thank Bengt-Olaf Schneider, who originally contributed earlier versions of the sections on personal workstations and parallel polygon graphics.

References

[1] G. Abram and H. Fuchs. Vlsi architectures for computer graphics. In G. Enderle, editor, *Advances in Computer Graphics I*, pages 6–21, Berlin, Heidelberg, New York, Tokyo, 1986. Springer-Verlag.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison Wesley, 1988.

[3] K. Akeley. Realityengine graphics. In *Computer Graphics (Proc. Siggraph)*, pages 109–116, August 1993.

[4] K. Akeley and T. Jermoluk. High-performance polygon rendering. *Computer Graphics (Proc. Siggraph)*, 22(4):239–246, August 1988.

[5] R. Avila, L. Sobierajski, and A. Kaufman. Towards a comprehensive volume visualization system. In *IEEE Visualization*, pages 13–20. IEEE CS Press, 1992.

[6] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Computer Graphics (SIGGRAPH '82 Proceedings)*, pages 21–29, July 1982.

[7] E. Camahort and I. Chakravarty. Integrating volume data analysis and rendering on distributed memory architectures. In *IEEE/ACM Parallel Rendering Symposium*, pages 89–96. ACM Press, October 1993.

[8] Ingrid Carlbom. Optimal filter design for volume reconstruction and visualization. In *IEEE Visualization*, pages 54–61, 1993.

[9] L. Carpenter. The a-buffer, an antialiased hidden surface method. *Computer Graphics (Proc. Siggraph)*, 18(3):125–138, 1985.

[10] J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *IEEE/ACM Parallel Rendering Symposium*, pages 81–88, 1993.

[11] Earl Coddington. *An Introduction to Ordinary Differential Equations*. Prentice-Hall, 1961.

[12] M. Cox and P. Hanrahan. Depth complexity in object-parallel graphics architectures. In *Proc. 7th Eurographics Workshop on Graphics Hardware*, pages 204–222, Cambridge (UK), 1992.

[13] Roger Crawfis and Nelson Max. Direct volume visualization of three-dimensional vector fields. *1992 Workshop on Volume Visualization*, pages 55–60, 1992.

[14] T. W. Crockett. Parallel rendering. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 34, Supp. 19, A., pages 335–371. Marcel Dekker, 1996. Also available as ICASE Report No. 95-31 (NASA CR-195080), 1996.

[15] T.W. Crockett and T. Orloff. A parallel rendering algorithm for mimd architectures. Technical Report ICASE-Report 91-3, Institute for Computer Science and Engineering, NASA Langley Research Center, 1991.

[16] John Danskin and Pat Hanrahan. Fast algorithms for volume ray tracing. *1992 Workshop on Volume Visualization*, pages 91–98, 1992.

[17] M. Deering and S.R. Nelson. Leo: A system for cost effective 3d shaded graphics. In *Computer Graphics (Proc. Siggraph)*, pages 101–108, August 1993.

[18] S. Demetrescu. High-speed image rasterization using scan line access memories. In H. Fuchs, editor, *Proc. Chapel Hill Conference on VLSI*, pages 221–243, 1985.

[19] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 65–74, August 1988.

[20] D. Ellsworth. A new algorithm for interactive graphics on multicomputers. *IEEE Computer Graphics & Applications*, pages 33–40, July 1994.

[21] A. Barkans et al. Guardband clipping method and apparatus for 3d graphics display system. U.S. Patent 4,888,712. Issued Dec 19, 1989.

[22] H. Fuchs et al. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics (Proc. Siggraph)*, 23(3):79–88, July 1989.

[23] I. Sutherland et al. A characterization of ten hidden surface algorithms. *ACM Computing Surveys*, 6(1):1–55, March 1974.

[24] J. Eyles et al. Pixelflow: The realization. In *Proc. of Eurographic/ACM SIGGRAPH Workshop on Graphics Hardware*, pages 57–68, New York, 1997. ACM Press.

[25] J. Foley et al. *Computer Graphics: Principles and Practice*. Addison Wesley, 2nd edition, 1990.

[26] M. Deering et al. The triangle processor and normal vector shader: A vlsi system for high-performance graphics. *Computer Graphics (Proc. Siggraph)*, 12(2):21–30, August 1988.

[27] S. Molnar et al. A sorting classification of parallel rendering. *IEEE Computer Graphics & Applications*, pages 23–32, July 1994.

[28] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990. Overview of research to date.

[29] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, pages 124–133, July 1980.

[30] C. Giertsen and J. Petersen. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, 13(6):16–23, 1993.

[31] Christopher Giertsen. Volume visualization of sparse irregular meshes. *IEEE Computer Graphics and Applications*, 12(2):40–48, March 1992.

[32] A. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, 1989.

[33] Andrew Glassner. *Principles of Digital Image Synthesis (2 Vols)*. Morgan Kaufmann Publishers, Inc. ISBN 1-55860-276-3, San Francisco, CA, 1995.

[34] Heinrich Müller and Michael Stark. Adaptive generation of surfaces in volume data. *The Visual Computer*, 9(4):182–199, January 1993.

[35] B. Hendrickson and R. Leland. The chaco user's guide (version 1.0). Tech. Rep. SAND93-2339, Sandia National Laboratories, Albuquerque, N.M., 1993.

[36] J. Hennesy and D. Paterson. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann, 1990.

[37] W. Hsu. Segmented ray casting for data parallel volume rendering. In *IEEE/ACM Parallel Rendering Symposium*, pages 7–14. ACM Press, October 1993.

[38] M. Hu and J. Foley. Parallel processing approaches to hidden-surface removal in image space. *Computers & Graphics*, 9(3):303–317, 1985.

[39] James T. Kajiya. The rendering equation. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, August 1986.

[40] James T. Kajiya and Brian P. Von Herzen. Ray tracing volume densities. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 165–174, July 1984.

[41] Arie E. Kaufman. *Volume Visualization*. IEEE Computer Society Press, ISBN 908186-9020-8, Los Alamitos, CA, 1990.

[42] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear–warp factorization of the viewing transformation. In Andrew Glassner, editor, *Proceedings of SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 451–458. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[43] T. Lee, C. Raghavendra, and J. Nicholas. Image composition methods for sort-last polygon rendering on 2d mesh architectures. In *IEEE/ACM Parallel Rendering Symposium*, pages 55–62, 1995.

[44] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[45] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[46] Marc Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, February 1990.

[47] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

[48] K. Ma, J. Painter, C. Hansen, and M. Krogh. A data distributed parallel algorithm for ray-traced volume rendering. In *IEEE/ACM Parallel Rendering Symposium*, pages 15–22. ACM Press, October 1993.

[49] K. Ma, J. Painter, C. Hansen, and M. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

[50] Kwan-Liu Ma. Parallel volume rendering for unstructured-grid data on distributed memory machines. In *IEEE/ACM Parallel Rendering Symposium*, pages 23–30, 1995.

[51] Kwan-Liu Ma and Thomas Crockett. Parallel volume rendering for unstructured-grid data on distributed memory machines. In *IEEE/ACM Parallel Rendering Symposium*, pages 95–104, 1997.

[52] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *IEEE Visualization*, pages 100–107, 1994.

[53] Nelson Max. Optical models for direct volume rendering. *IEEE Transations on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[54] Nelson Max, Roger Crawfis, and Barry Becker. New techniques in 3D scalar and vector field visualization. In *First Pacific Conference on Computer Graphics and Applications*. Korean Information Science Society, Korean Computer Graphics Society, August 1993.

[55] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 27–33, November 1990.

[56] Nelson L. Max. Efficient light propagation for multiple anisotropic volume scattering. In *Fifth Eurographics Workshop on Rendering*, pages 87–104, Darmstadt, Germany, June 1994.

[57] James V. Miller, David E. Breen, William E. Lorensen, Robert M. O'Bara, and Michael J. Wozny. Geometrically deformed models: A method for extracting closed geometric models from volume data. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 217–226, July 1991.

[58] S. Molnar. Combining Z-buffer engines for higher-speed rendering. In *Advances in Computer Graphics Hardware III*, pages 171–182, 1988.

[59] S. Molnar. *Image Composition Architectures for Real-Time Image Generation*. Ph.D. thesis, University of North Carolina, Chappel Hill, 1991.

[60] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification for parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[61] C. Montani, R. Perego, and R. Scopigno. Parallel volume visualization on a hypercube architecture. In *1992 Workshop on Volume Visualization Proceedings*, pages 9–16. ACM Press, October 1992.

[62] J. Montrym. Infinite reality: A real-time graphics system. In *Computer Graphics (Proc. Siggraph)*, pages 293–302, August 1997.

[63] C. Mueller. The sort-first rendering architecture for high-performance graphics. In *Proc. 1995 Symposium on Interactive 3D Graphics*, pages 75–84, New York, 1995. ACM Press.

[64] Henry Neeman. A decomposition algorithm for visualizing irregular grids. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 49–56, November 1990.

[65] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *IEEE/ACM Parallel Rendering Symposium*, pages 97–104. ACM Press, October 1993.

[66] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory mimd architectures. In *1992 Workshop on Volume Visualization Proceedings*, pages 17–24. ACM Press, October 1992.

[67] Gregory M. Nielson and Bernd Hamann. The asymptotic decider: Removing the ambiguity in marching cubes. In *IEEE Visualization*, pages 83–91, 1991.

[68] M. Olano and T. Greer. Triangle scan conversion using 2d homogeneous coordinates. In *Proc. of Eurographic/ACM SIGGRAPH Workshop on Graphics Hardware*, pages 89–96, New York, 1997. ACM Press.

[69] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization*, pages 233–238, 1998.

[70] J. Pineda. A parallel algorithm for polygon rasterization. *Computer Graphics (Proc. Siggraph)*, 22(4):17–20, August 1988.

[71] Thomas Porter and Tom Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

[72] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[73] C.R. Ramakrishnan and C.T. Silva. Optimal processor allocation for sort-last compositing under bsp-tree ordering, submitted for publication, 1997.

[74] D. Roble. A load balanced parallel scanline z-buffer algorithm for the ipsc hypercube. In *Proc. Pixim '88*, pages 177–192, Paris (France), October 1988.

[75] Paolo Sabella. A rendering algorithm for visualizing 3D scalar fields. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 51–58, August 1988.

[76] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts, 1990.

[77] B.-O. Schneider. A processor for an object-oriented renderin system. *Computer Graphics Forum*, 7:301–310, 1988.

[78] B.-O. Schneider and J. van Welzen. Efficient polygon clipping for a simd graphics pipeline. *IEEE Transactions on Visualization and Computer Graphics*, 4(3), July 1998. To appear.

[79] P. Schroeder and J. Salem. Fast rotation of volume data on data parallel architectures. In *IEEE Visualization*, pages 50–57. IEEE CS Press, 1991.

[80] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.

[81] R. Sethi and J. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4), 1970.

[82] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 63–70, November 1990.

[83] C. Silva. *Parallel Volume Rendering of Irregular Grids*. Ph.D. thesis, State University of New York at Stony Brook, 1996.

[84] C. Silva, A. Kaufman, and C. Pavlakos. PVR: High-Performance Volume Rendering. In *IEEE Computational Science and Engineering*, 1996.

[85] C.T. Silva and J.S.B. Mitchell. The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transations on Visualization and Computer Graphics*, 3(2), 1997.

[86] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, 1994.

[87] L. Sobierajski and R. Avila. A hardware acceleration method for volume ray tracing. In *IEEE Visualization*. IEEE CS Press, 1995.

[88] Lisa Sobierajski and Arie Kaufman. Volumetric ray tracing. In Arie Kaufman and Wolfgang Krueger, editors, *1994 Symposium on Volume Visualization*, pages 11–18. ACM SIGGRAPH, October 1994. ISBN 0-89791-741-3.

[89] Don Speray and Steve Kennon. Volume probes: Interactive data exploration on arbitrary grids. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 5–12, November 1990.

[90] Craig Upson and Michael Keeler. VBUFFER: Visible volume rendering. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 59–64, August 1988.

[91] Sam Uselton. Volume rendering for computational fluid dynamics: Initial results. Tech Report RNR-91-026, Nasa Ames Research Center, 1991.

[92] R. Weinberg. Parallel processing image synthesis with anti-aliasing. *Computer Graphics (Proc. Siggraph)*, 15(3):55–62, August 1981.

[93] Lee Westover. Footprint evaluation for volume rendering. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 367–376, August 1990.

[94] D. Whelan. A rectangular area filling display system architecture. *Computer Graphics (Proc. Siggraph)*, 16(3):147–153, July 1982.

[95] S. Whitman. *Multiprocessor Methods for Computer Graphics Rendering*. Jones and Bartlett, Boston, London, 1992.

[96] S. Whitman. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics & Applications*, pages 41–48, July 1994.

[97] S. R. Whitman. A survey of parallel algorithms for graphics and visualization. In International Workshop on High-Performance Computing for Computer Graphics and Visualization, Swansea, United Kingdom, 1995.

[98] Jane Wilhelms and Judy Challinger. Direct volume rendering of curvilinear volumes. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 41–47, November 1990.

[99] Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 275–284, July 1991.

[100] Peter L. Williams and Nelson Max. A volume density optical model. *1992 Workshop on Volume Visualization*, pages 61–68, 1992.

[101] R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, pages 19–28, 1992.

[102] K. Zuiderveld. *Visualization of Multimodality Medical Volume Data using Object-Oriented Methods*. PhD thesis, University of Utrecht, The Netherlands, 1995.

[103] K. Zuiderveld, A. Koning, and M. Viergever. Acceleration of ray-casting using 3D distance transforms. In *Proc. of Visualization in Biomedical Computing*, pages 324–335. SPIE, 1992.

Building and Driving High-Resolution Displays

James T. Klosowski
IBM T. J. Watson Research Center

Outline

- Motivation
- High-Resolution Displays
- Tiled Display Walls
- LCDs
- Chromium

SC 2002
Tutorial M9

Motivation

- **Who needs high-resolution anyway?**
  - Whoever wants to visualize large data
    - Accelerated Strategic Computing Initiative (ASCI)
  - Whoever wants to collaborate effectively
    - Industrial design, petroleum exploration, entertainment

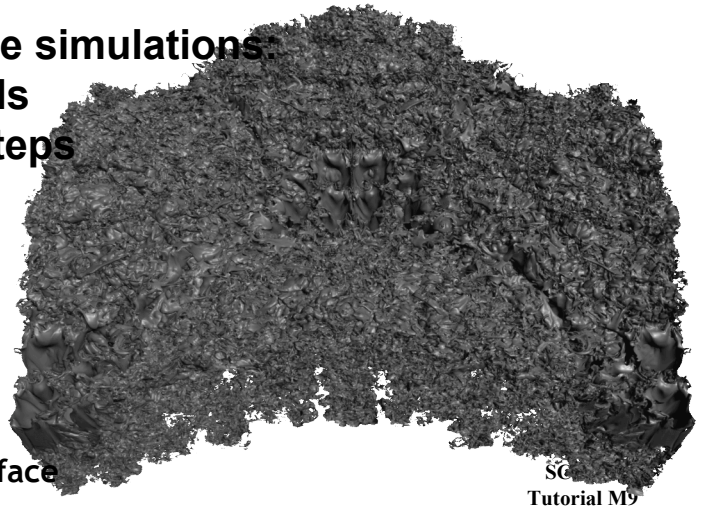
Motivation

- **ASCI Visualization Needs**
  - High fidelity simulations required to achieve the needed confidence in simulation results

	Today's High-end Technology	2004 Needs
Surface Rendering	~2.5 Million polygons per second per graphics pipe	20 Billion polygons per second (aggregate)
Pixel Fill Rate	~.1 Gpixel per raster manager	200 Gpixel (aggregate)
Display Resolution	16 Mpixel (4K x 4K)	64 Mpixel (8K x 8K)

Applications: Scientific Visualization

- **ASCI terascale simulations:**
  8 billion voxels
  27,000 time steps

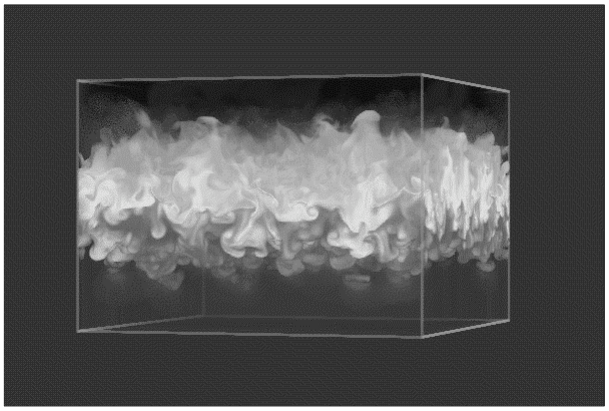


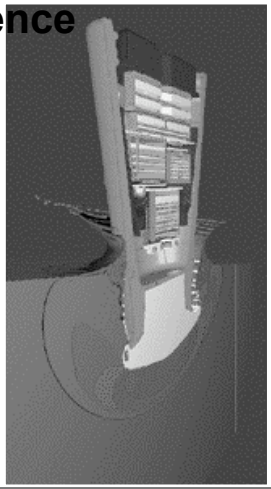
469M Triangle Isosurface

SC
Tutorial M9

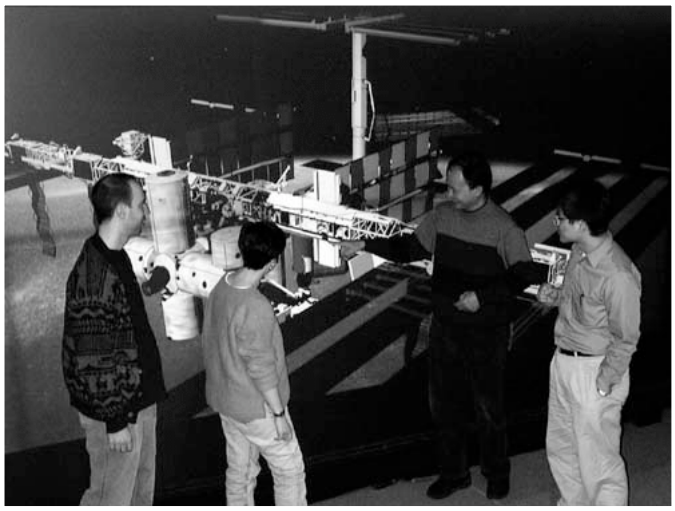
Applications: Simulation Analysis

- **Crashes, Fires, Explosions, Turbulence**



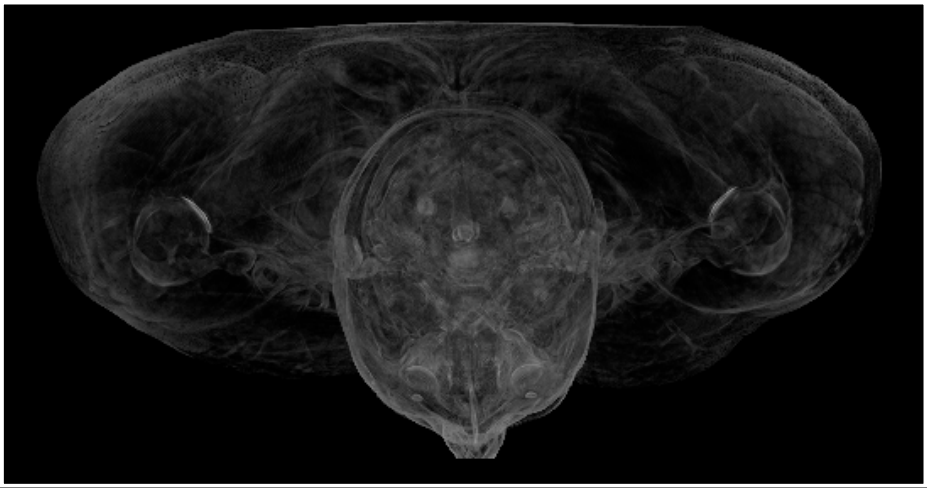


Applications: Collaboration

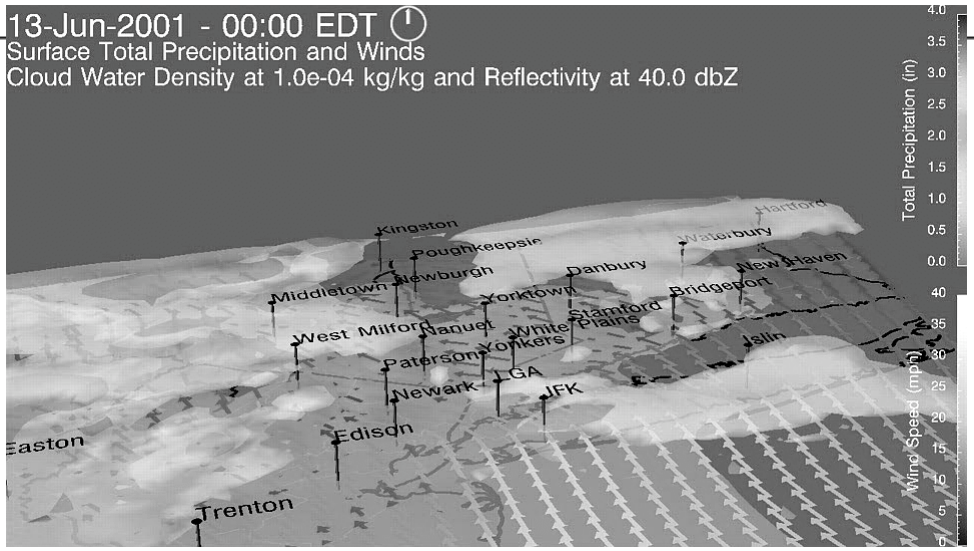


SC 2002
Tutorial M9

Applications: Medical Imaging



Applications: Weather Forecasting



Other Applications

- Flight Simulators
- Planetariums
- Manufacturing
- Automotive Design
- Oil & Gas Exploration
- Entertainment

Outline

- **Motivation**
- **High-Resolution Displays**
- **Tiled Display Walls**
- **LCDs**
- **Chromium**

High-Resolution Displays

- **Multi-Projector Displays**
  - Tiled, Cylindrical, Rectilinear, Domes
    - Our focus will be on tiled display walls
- **Liquid Crystal Displays (LCDs)**
  - IBM T221 (9.2 million pixels)

Tiled Display Walls

- **Projectors**
  - CRT, LCD, DLP, laser
  - Characteristics to consider:
    resolution, brightness, contrast, color gamut, gamma, data interfaces, image quality, calibration, flatness of illumination
  - Front versus rear projection

Tiled Display Walls

- **Screens**
  - Characteristics to consider:
    image performance (brightness, resolution, angle of view, contrast ratio), weight, fragility, portability
  - Color balance and dynamic range should match projectors

Tiled Display Walls

- **Pros**
  - Great for data visualization & collaboration
    - Resolution, human scale
- **Cons**
  - Seamless display complications
  - Large footprint

LCDs

- **Characteristics to consider:**
  - Resolution, contrast, pixels per inch, vertical refresh, viewable image area

LCDs

- **Pros**
  - Great for data visualization
    - Resolution, seamless display
  - Small footprint
- **Cons**
  - Poor for large group collaborations

Outline

- **Motivation**
- **High-Resolution Displays**
- **Tiled Display Walls**
  - Driving tiled displays
  - Seamless displays: challenges
  - Examples of existing systems
- **LCDs**
- **Chromium**

Tiled Display Walls: Manufacturers

- **Projectors**
  - Barco, Boxlight, Christie, Compaq, Dell, Epson, Fujitsu, Hitachi, Mitsubishi, NEC, Optoma, Proxima, Sharp, Sony, and many, many others
- **Screens**
  - Da-Lite, Draper, JenMar, Stewart Filmscreen

Driving Tiled Display Walls

- **Hardware**
  - Custom/High-End machine
    - SGI
  - Clusters of workstations
    - Commodity PCs

SGI

- **Pros**
  - Distributed shared-memory architecture
  - InfiniteReality accelerators (HW edge blending)
  - Simple display synchronization
  - Simplier development of applications
- **Cons**
  - Expensive, large footprint
  - Limited number of projectors (max 16 IR pipes)

Cluster of Workstations

- **Pros**
  - Price/performance
  - Tracks commodity parts very well (gfx, networks, CPUs)
  - Potential scalability
- **Cons**
  - More complicated application development
  - I/O limitations within and between nodes
  - Graphics capabilities driven by gamer's needs

Driving Tiled Display Walls

- **Parallel Rendering**
  - Aggregation of rendering output
- **Load Balancing**
- **Synchronization**

Parallel Rendering: Sort-First

- Geometric primitives distributed among nodes
- 2D screen partitioning into disjoint tiles
- Prior to transformation step, primitives redistributed to appropriate regions
- Pros:
  frame-to-frame coherence; any rendering algorithm
- Cons:
  load balancing; border primitives

Parallel Rendering: Sort-Middle

- Arbitrary assignment of geometric primitives to nodes
- Portion of total display assigned to each rasterizer
- Geometry processor transforms and lights primitives
- Screen-space primitives sorted and sent to rasterizers
- Most popular until very recently
- Cons:
  load balance on rasterizers, special hardware

Parallel Rendering: Sort-Last

- Arbitrary assignment of geometric primitives to nodes
- Geometry rendered
- Pixels (color & depth) sent to compositing nodes
- Pros:
  data size scalability
- Cons:
  high bandwidth; transparency & anti-aliasing

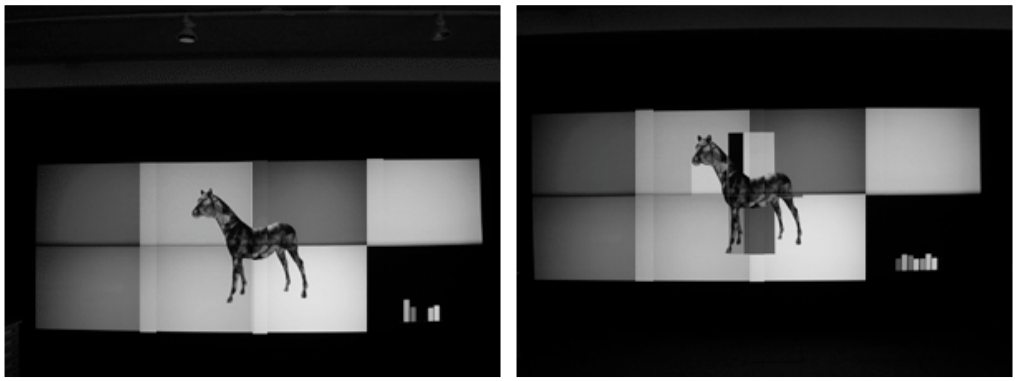
Clusters of Workstations

- **Sort-first and sort-last most appropriate**
- **Sort-middle not appropriate:**
  - Geometry processing and rasterization are tightly coupled on commodity graphics accelerators: only slow access to intermediate rendering results

Parallel Rendering: Samanta 1999

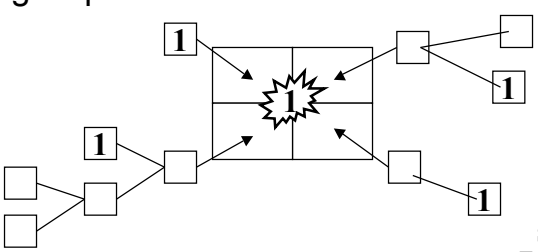
- **Load balancing multi-projector systems**
  - Sort-first rendering
  - Partition image into virtual tiles for each frame (k-d tree)
    - virtual tiles are different from physical (projector) tiles
    - readback pixels from framebuffer and transfer them to PC driving the corresponding physical tile's projector
  - VMMC: low latency, high bandwidth, & fast synchronization for framebuffer swap

Parallel Rendering: Samanta 1999



Parallel Rendering: Moreland 2001

- **Extremely large data sets**
  - Sort-last rendering on 12Mpel tiled display wall
  - Each nodes renders images for each tile in the wall
  - 120 million triangles per second



Aggregation of Rendering

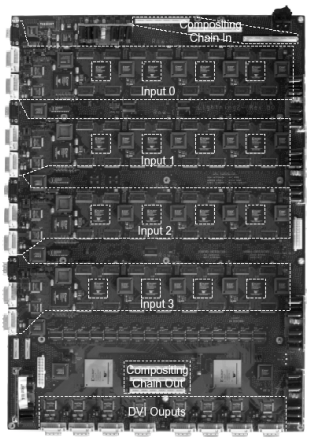
- **Sort-first**
  - 1 screen tile per node: each node drives 1 projector
  - 2+ tiles per node: pixels must be transferred b/w nodes
- **Sort-last**
  - Readback framebuffer (RGB, Z, alpha, etc.)
  - Transmit contents to compositing engine
  - Composite contents & display results

Sort-Last Aggregation

- **Compositing done on nodes**
  - imagery exchanged over cluster interconnect (binary swap)
- **Complications**
  - framebuffer readback performance is generally slow
  - bandwidth and latency
  - processor overhead

Sort-Last Aggregation

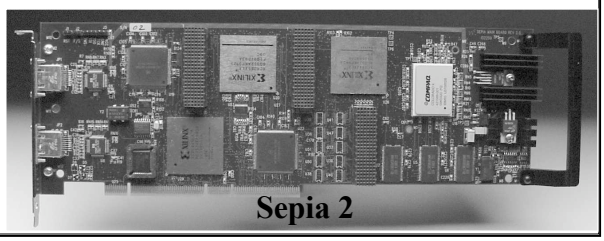
- **Compositing done on DVI attached HW**
  - Stanford's Lightning-2
    U. Texas' Metabuffer
  - Avoids readback bottleneck
- **Complications**
  - Synchronization issues
  - DVI limitations
  - Limited compositing functions



Lightning-2

Sort-Last Aggregation

- **Compositing done on network attached HW**
  - Compaq's Sepia, IBM's Scalable Graphics Engine
  - Offload compositing operation from cluster nodes
  - General pixel formats, programmable composition functions
- **Complications**
  - Framebuffer readback
  - Bus contention

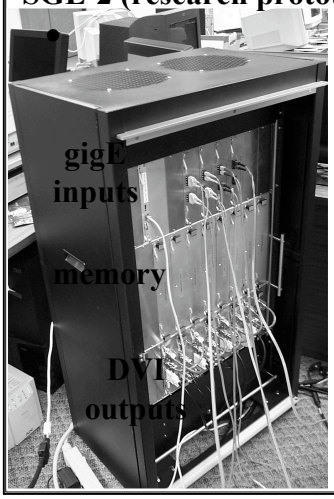

Sepia 2

IBM Scalable Graphics Engine

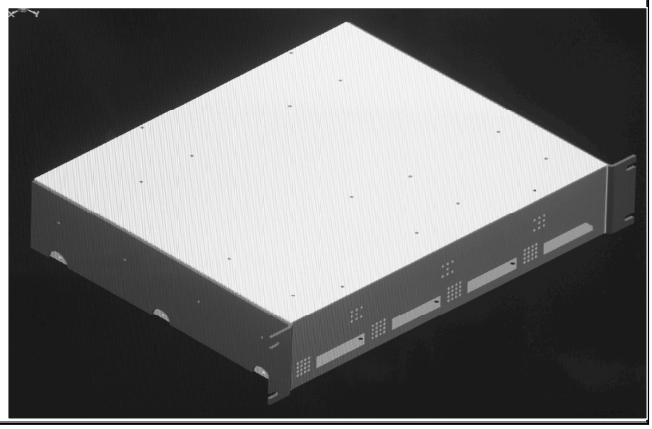
- **Network attached framebuffer**
  - 16 million RGBA pixels double buffered
  - up to 16 GigE inputs, 8 synchronized DVI outputs
  - up to 720 Mpel/s display refresh bandwidth
- **Not a graphics adapter!**

SGE: Today and Tomorrow

SGE-2 (research prototype)



SGE-3 (artist's drawing)



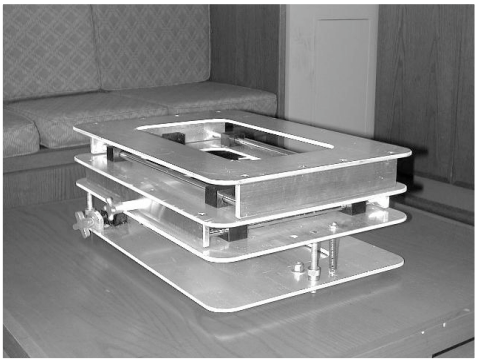
Challenges (to seamless display)

- **Projector Alignment**
- **Color calibration**
- **Blending**

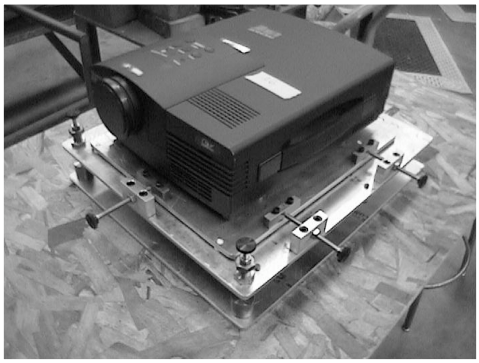
Projector Alignment

- **Very difficult problem (active research area)**
  - Many degrees of freedom
  - Distortions
  - Manual alignment is time consuming and tedious
- **Hardware and software solutions**

Mechanical Projector Positioners



Princeton-Intel



Argonne National Lab

Positioners not as effective for commodity projectors (due to distortions)

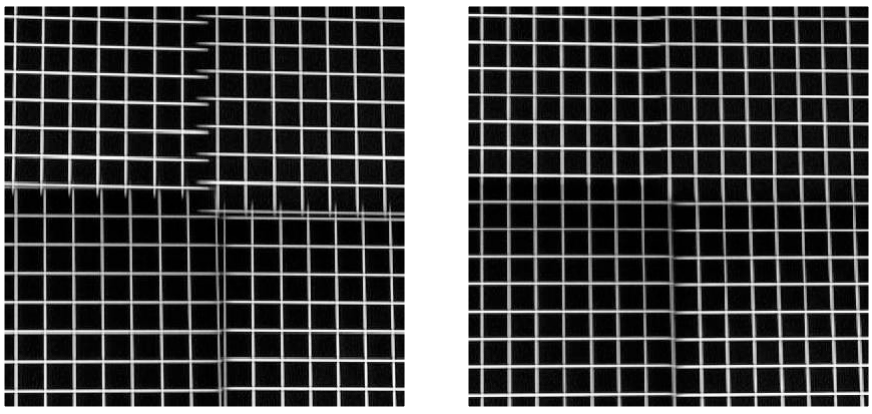
Projector Alignment

- **Software controlled alignment**
  - MIT, UNC: automatically distort images sent to projectors by using a calibrated camera
  - UNC: Office of the Future -- freeform projector placement with alignment fixed completely in software
  - Princeton: uncalibrated camera

Projector Alignment: Chen 2000

- **Camera measures point correspondences & line matches between projectors**
  - Simulated annealing used to minimize global error and solve for the projection matrices
  - Image-warping process resamples the images to counter the effects of physical misalignment

Projector Alignment: Chen 2000



Projector Alignment: Yang 2001

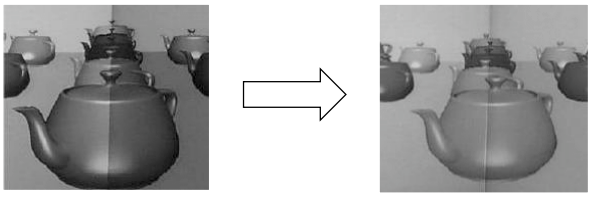
- **PixelFlex: multi-projector system**
  - Reconfigurable: can change pixel density, size, and shape of single logical display
  - Calibration (image warping and blending) runs in minutes
  - Front-projection system
    - ceiling-mounted computer aligned projectors
    - wide field-of-view camera for closed-loop calibration (must see entire display surface)

Projector Alignment: New Research

- **Chen et al, IEEE Visualization 2002**
  - Scalable Alignment of Large-Format Multi-Projector Displays Using Camera Homography Trees
- **Raskar, et al., ACCV 2002**
  - A Low-Cost Projector Mosaic with Fast Registration

Color Calibration

- **Variation between projectors (even same model)**
- **Variation within a projector's FOV**



Color Calibration

- **Manual adjustment of projectors possible for small numbers**
- **Computer assisted adjustment**
  - Color analyzer used to adjust each projector's color through the serial port
- **Software adjustment of colors**

Color Calibration: Majumder 2000

- **Goal: For the same input, the colors perceived at every pixel of every projector are equivalent**
- **Approach: color mapping**
  - for each projector, map input color to a new input color to produce perceptually identical pixels

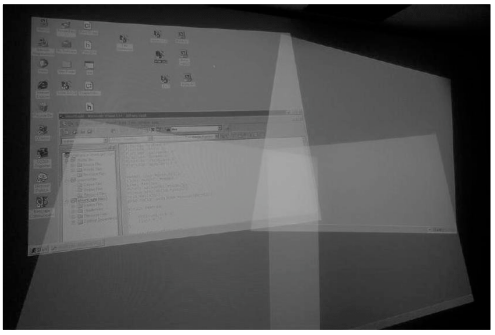
Color Calibration: Majumder 2000

- **For 24-bit color, number of inputs is large: 2^{24}**
  - Plus the variation within a projector's FOV
- **Reduce number of color maps needed**
  - ignore variation within a projector
  - one color map per channel per projector

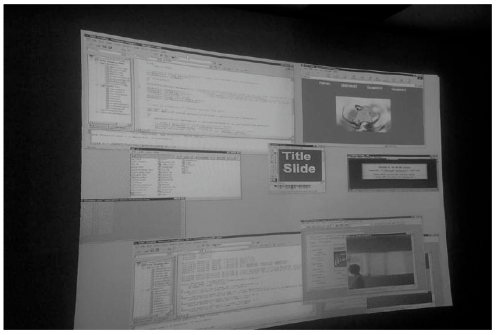
Color Calibration: Majumder 2000

- **Color map algorithm:**
  - non-linearity correction
  - luminance matching
  - chrominance matching
- **Use the per channel color LUT available in image generators**

Color Calibration: Majumder 2002



uncalibrated



calibrated

Blending

- **Overlapping tiles and tapering brightness of images can give smooth transitions**
  - Soft averaging of errors (misalignment, zoom, distortion)
- **LCD/DLP projectors leak light for black pixels**
  black + black = gray

Blending: Solutions

- **Avoid it**
  - Reduces pixel counts & creates a “blurred” region that some people notice more than the edges themselves
  - High-end projectors (better color matching & flat illumination) and very careful alignment
- **Software**
  - use roll-off function (linear ramp, cosine curve) to smooth the intensity transition between projectors

Blending: Solutions

- **Hardware**
  - Modulate video signals (Panoram, Trimension)
    - Only available in high-end CRT projectors
  - Aperture modulation (Princeton)
    - Obscure projected image in overlap region with opaque object (no change to input images needed)

Blending: Aperture Modulation

Existing Systems

- Sandia National Laboratories
- L.L.N.L.: PowerWall
- Princeton: Scalable Display Wall
- Stanford: Interactive Mural
- Many others

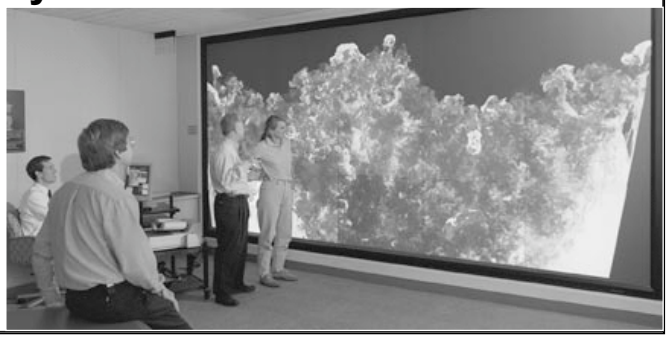
Sandia National Laboratories

- 20 Mpel wall (10'x 3')
- 64 PC cluster
- 4x4 projector array

- Next step: 60 Mpel

Lawrence Livermore National Lab

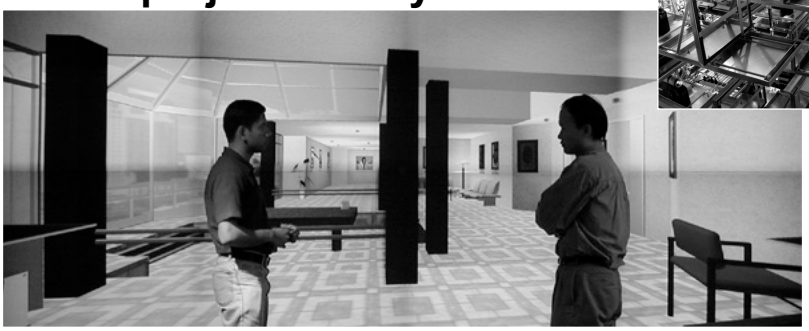
- 20 Mpel PowerWall (16'x8')
- SGI Onyx2 (16 IR pipes)
- 5x3 projector array



- 3x2 PowerWall
  SGI Onyx2 or
  32 PC cluster

Princeton's Scalable Display Wall

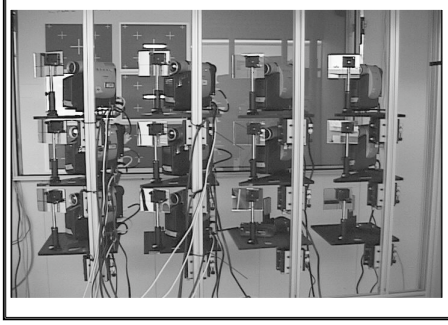
- 18 Mpel (18'x8')
- 24 PC cluster
- 6x4 projector array



SC 2002
Tutorial M9

Stanford's Interactive Mural

- 9 Mpel (6'x4')
- SGI Onyx2 or PC cluster
- 3x4 projector array



Many others...

- Argonne National Laboratory: ActiveMural
- AT&T: InfoWall
- MIT: DataWall
- Univ. of Minnesota: PowerWall
- Univ. of Texas at Austin

Outline

- **Motivation**
- **High-Resolution Displays**
- **Tiled Display Walls**
- **LCDs**
  - Available displays
  - How to drive them
- **Chromium**

Displays

- **IBM T221**
  - 3840 x 2400 pixels > 9.2 Mpel
  - 204 pixels per inch
  - 22.2 inches viewable image size
- **Viewsonic VP2290b**
  - Same panel as T221

IBM T221

- Vertical refresh: 41 Hz
- Contrast: 400:1
- Pixel pitch: 0.1245 mm
- Viewing angle: 170 degrees (horiz & vert)
- 21.5” W x 17.2” H x 7.7” D
- 26.4 pounds, 135 watts

Quake3 on the T221

High-Resolution LCD Market

- **Engineering**
- **Finance**
- **Publishing**
- **Earth Resource Management**
- **Digital Content**
- **Medicine**

Driving the T221

- **2D**
  - Matrox 200MMS (32-bit PCI, 32 MB, 4 DVI)
    - 4 stripes 960x2400, or 4 tiles 1920x1200
- **3D**
  - nVidia Quadro4 900 XGL (AGP 4x, 128 MB, 2 DVI)
    - full screen, or 2 stripes 1920x2400
  - ATI Fire GL4 (AGP 4x, 128 MB, 2 DVI)
    - 2 stripes 1920x2400

Driving the T221

- **Many other cards also capable**
  - 3D Labs, IBM, ATI, nVidia, etc.
- **Drivers**
  - Windows: widely available
  - Linux: somewhat available

Driving the T221

- **PC cluster**
  - A single graphics card has limited resources
  - By combining computational power of several PCs, the throughput to the T221 can be even higher
    - Direct connection to T221
      - – synchronization becomes an issue
    - Compositing hardware
      - – SGE, Lightning-2, Sepia, MetaBuffer

IBM Deep View Visualization System

- **8 Linux nodes**
  - Real-time rendering
  - Media streaming
- **SGE**
- **T221**

Outline

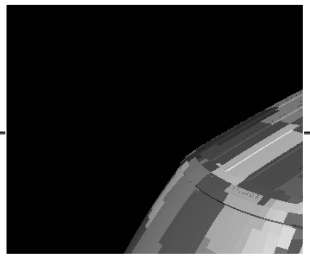
- **Motivation**
- **High-Resolution Displays**
- **Tiled Display Walls**
- **LCDs**
- **Chromium**

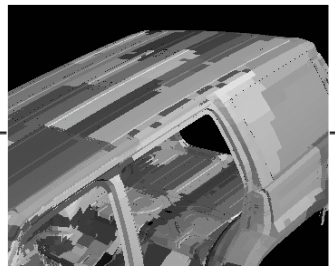
Chromium: Cluster Rendering System

- **Open Source Software (Sourceforge, Free-BSD)**
- **Development initiated at Stanford**
- **DOE/ASCI VIEWS funding**
- **Follow-up project to WireGL**
- **Humphreys et al. SIGGRAPH 2002**

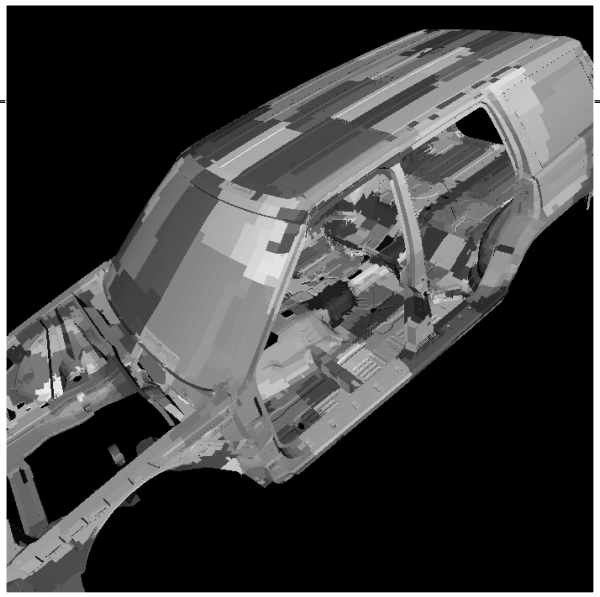
WireGL: Humphreys 2001

- **Allows an OpenGL application to render in parallel on a cluster of workstations**
  - Supports existing applications without modifications
  - Transparent support for tiled displays (sort-first)
  - Parallel interface for (input) scalability
    - requires OpenGL parallel extensions (Igehy, et al. 1998)





SC 2002
Tutorial M9



SC 2002
Tutorial M9

WireGL's limitations

- **Too restrictive**
  - sort-first is hard to load balance
  - screen-space parallelism is limited
  - dependent on spatial locality
- **Resource utilization**
  - geometry moved over network each frame
  - server's graphics hardware remains underutilized

Chromium (Cr)

- **Cr approaches cluster rendering as processing streams of OpenGL commands**
  - Forms a DAG of transformation nodes (computers)
  - Edges in DAG are OpenGL communications
  - Each node can generate, absorb, or modify a stream of OpenGL commands
  - WireGL operations: tile-sort (on client) & render (on server)

Chromium

- **Cr servers are similar to those in WireGL**
  - Difference lies in the decoding library: Cr allows arbitrary processing, not just render and display
- **Cr can do everything that WireGL did**

Chromium: SPUs

- **Stream Processing Units (SPUs)**
  - Shared libraries
  - Generic, interchangeable
  - Each node loads a chain of SPUs
- **Commonly, SPU will intercept a few OpenGL calls, pass all others to downstream SPU**

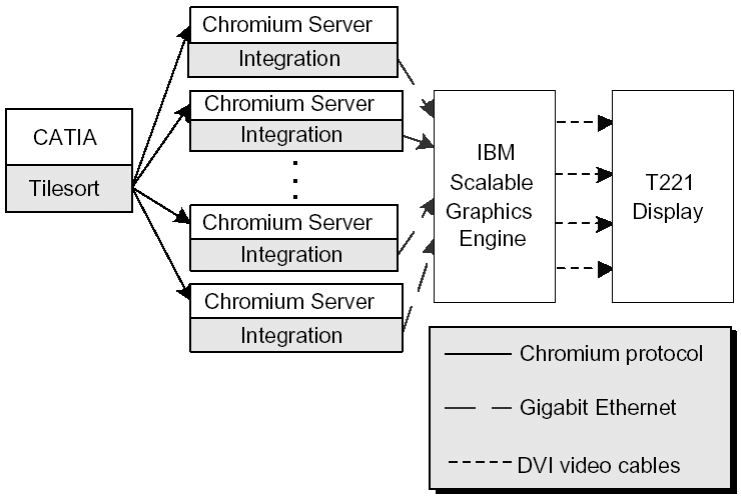
Chromium

- Libraries: network, packing, unpacking, & state tracking
- Sort-first, sort-last out of the box
- Mechanisms provide greater flexibility than algorithms
- Windows, Linux, AIX, Irix, Solaris, HPUX, +

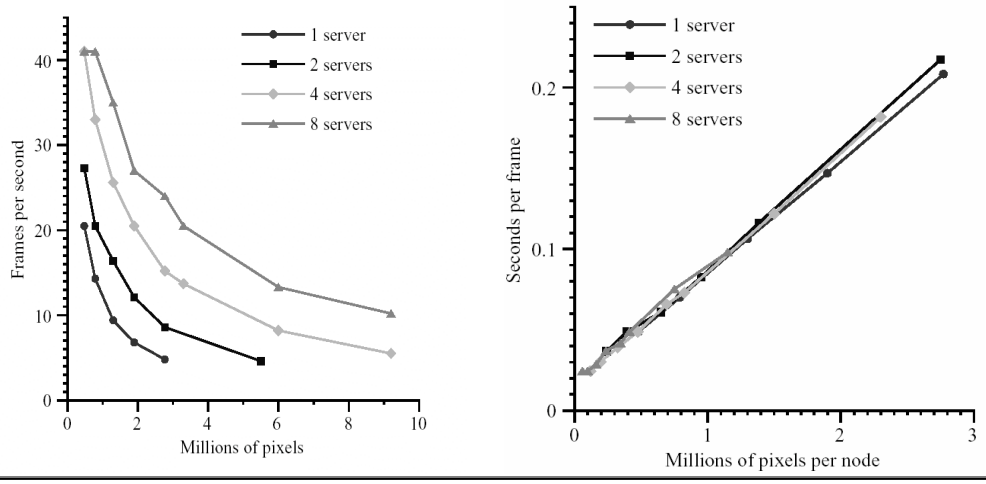
Chromium on Deep View

- Application Integration
- AIX client & Linux servers

Chromium with Application Integration



Chromium on Deep View: Scalability



Summary

- High-resolution display systems are an efficient, cost-effective means of visualizing large data
- Tiled display walls offer high resolution on a human scale for visualization and collaboration
- LCDs offer seamless, high resolution for beautiful visualizations

Good References

- SuperComputing 2002 Course #M9: High-Performance Visualization of Large and Complex Scientific Datasets (Building and Driving High-Resolution Displays)
  http://www.research.ibm.com/people/j/jklosow/sc2002.html
  jklosow@us.ibm.com
- ACM SIGGRAPH Course #37: Commodity-based Scalable Visualization
  Pavlakos, et al., ACM SIGGRAPH 2001
- ACM SIGGRAPH Course #55: Projector-based Graphics
  Raskar and Lantz, ACM SIGGRAPH 2002

Selected Bibliography (categorized)

Projector Alignment:
4, 6, 8, 11, 12, 19, 24, 25, 26, 29, 31, 32

Color Calibration:
11, 12, 21, 32

Blending:
11, 12, 19, 20, 24, 25, 26, 29, 31

Parallel Rendering & Compositing:
2, 7, 9, 10, 13, 14, 15, 16, 17, 18, 19, 22, 23, 27

Projector-based Systems:
1, 3, 5, 7, 10, 11, 12, 16, 19, 24, 28, 30, 32

Selected Bibliography

1. Working in the Office of "Real Soon Now"
Bishop and Welch, IEEE CG&A 20(4), 2000
2. The MetaBuffer: A Scalable Multiresolution Multidisplay 3D Graphics System using Commodity Rendering Engines, Blanke et al., CS Dept, U Texas (Austin), TR2000-16
3. Large Displays in Automotive Design
Buxton, et al., IEEE CG&A 20(4), 2000
4. Scalable Alignment of Large-Format Multi-Projector Displays Using Camera Homography Trees, Chen, et al., IEEE Vis 2002
5. Experiences with Scalability of Display Walls
Chen, et al., Immersive Projection Technology Workshop, 2002
6. Calibrating Scalable Multi-Projector Displays Using Camera Homography Trees
Chen, et al., Computer Vision and Pattern Recognition (sketch) 2001
7. Data Distribution Strategies for High Resolution Displays
Chen, et al., Computer Graphics 25(5) 2001
8. Automatic Alignment of High-Resolution Multi-Projector Displays Using an Uncalibrated Camera, Chen, et al., IEEE Vis 2000

Selected Bibliography

9. Out-of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays
   Correa, et al., Eurographics Workshop on Parallel Graphics and Visualization 2002
10. Design Considerations for a Multi-Projector Display Rendering Cluster
    Gotz, UNC Technical Report 10-025, 2001
11. Introduction to Building Projection-Based Tiled Display Systems
    Hereld, et al., IEEE CG&A 20(4), 2000
12. Developing Tiled Projection Display Systems
    Herald, et al., Immersive Projection Technology Workshop, 2000
13. Chromium: A Stream Processing Framework for Interactive Rendering on Clusters
    Humphreys, et al., ACM SIGGRAPH 2002
14. WireGL: A Scalable Graphics System for Clusters
    Humphreys, et al., ACM SIGGRAPH 2001
15. Distributed Rendering for Scalable Displays
    Humphreys, et al., SuperComputing 2000
16. A Distributed Graphics System for Large Tiled Displays
    Humphreys and Hanrahan, IEEE Vis 1999

Selected Bibliography

17. The Design of a Parallel Graphics Interface
    Igehy, et al., ACM SIGGRAPH 1998
18. Deep View: High-Resolution Reality
    Klosowski, et al., IEEE CG&A 22(3), 2002
19. Building and Using a Scalable Display Wall System
    Li, et al., IEEE CG&A 20(4), 2000
20. Optical Blending for Multi-Projector Display Wall System
    Li and Chen, Lasers and Electro-Optics Society 1999 Annual Meeting
21. Achieving Color Uniformity Across Multi-Projector Displays
    Majumder, et al., IEEE Vis 2000
22. Sepia: Scalable 3D Compositing using PCI Pamette
    Moll, et al., IEEE Field Programmable Custom Computing Machines 1999
23. Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays
    Moreland, et al., IEEE PVG 2001
24. A Low-Cost Projector Mosaic with Fast Registration
    Raskar, et al., Asian Conference on Computer Vision 2002

Selected Bibliography

25. Multi-Projector Displays Using Camera-Based Registration
    Raskar, et al., IEEE Vis 1999
26. Seamless Projection Overlaps using Image Warping and Intensity Blending
    Raskar, et al., 4th Conference on Virtual Systems and Multimedia, 1998
27. Load Balancing for Multi-Projector Rendering Systems
    Samanta, et al., SIGGRAPH/Eurographics Workshop on Graphics Hardware 1999
28. High-Resolution Multiprojector Display Walls
    Schikore, et al., IEEE CG&A 20(4), 2000
29. Scalable Self-Calibrating Technology for Seamless Large-Scale Displays
    Surati, Ph.D. Thesis, MIT, 1999
30. Visualization Research with Large Displays
    Wei, et al., IEEE CG&A 20(4), 2000
31. Projected Imagery in Your "Office of the Future"
    Welch, et al., IEEE CG&A 20(4), 2000
32. PixelFlex: A Reconfigurable Multi-Projector Display System
    Yang, et al., IEEE Vis 2001

Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters

Greg Humphreys* Mike Houston* Ren Ng* Randall Frank† Sean Ahern† Peter D. Kirchner‡
James T. Klosowski‡

*Stanford University †Lawrence Livermore National Laboratory ‡IBM T.J. Watson Research Center

Abstract

We describe Chromium, a system for manipulating streams of graphics API commands on clusters of workstations. Chromium's stream filters can be arranged to create sort-first and sort-last parallel graphics architectures that, in many cases, support the same applications while using only commodity graphics accelerators. In addition, these stream filters can be extended programmatically, allowing the user to customize the stream transformations performed by nodes in a cluster. Because our stream processing mechanism is completely general, any cluster-parallel rendering algorithm can be either implemented on top of or embedded in Chromium. In this paper, we give examples of real-world applications that use Chromium to achieve good scalability on clusters of workstations, and describe other potential uses of this stream processing technology. By completely abstracting the underlying graphics architecture, network topology, and API command processing semantics, we allow a variety of applications to run in different environments.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; I.3.4 [Computer Graphics]: Graphics Utilities—Software support, Virtual device interfaces; C.2.2 [Computer-Communication Networks]: Network Protocols—Applications; C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/Server, Distributed Applications

Keywords: Scalable Rendering, Cluster Rendering, Parallel Rendering, Tiled Displays, Remote Graphics, Virtual Graphics, Stream Processing

1 Introduction

The performance of consumer graphics hardware is increasing at such a fast pace that a large class of applications can no longer utilize the full computational potential of the graphics processor. This is largely due to the slow serial interface between the host and the graphics subsystem. Recently, clusters of workstations have emerged as a viable option to alleviate this bottleneck. However, cluster rendering systems have largely been focused on providing specific algorithms, rather than a general mechanism for enabling interactive graphics on clusters. The goal of our work is to allow applications to utilize more easily the aggregate rendering power of a collection of commodity graphics accelerators housed in a cluster of workstations, without imposing a specific scalability algorithm that may not meet an application's needs.

To achieve this goal, we have designed and built a system that provides a generic mechanism for manipulating streams of graphics API commands. This system, called Chromium, can be used as the underlying mechanism for any cluster-graphics algorithm by having the algorithm use OpenGL to move geometry and imagery across a network as required. In addition, existing OpenGL applications can use a cluster with very few modifications, because Chromium provides an industry-standard graphics API that virtualizes the disjoint rendering resources present in a cluster. In some cases, the application does not even need to be recompiled. Compatibility with existing applications may accelerate the adoption of rendering clusters and high resolution displays, encouraging the development of new applications that exploit resolution and parallelism.

Chromium's stream processors are implemented as modules that can be interchanged and combined in an almost completely arbitrary way. By modifying the configuration of these stream processors, we have built sort-first and sort-last parallel graphics architectures that can, in many cases, support the same applications without recompilation. Unlike previous work, our approach does not necessarily require that any geometry be moved across a network (although this may be desirable for load-balancing reasons). Instead, applications can issue commands directly to locally housed graphics hardware, thereby achieving the node's full advertised rendering performance. Because our focus is on clusters of commodity components, we consider only architectures that do not require communication between stages in the pipeline that are not normally exposed to an application. For example, a sort-middle architecture, which requires communication between the geometry and rasterization stages, is not a good match for our system.

Chromium's stream processors can be extended programmatically. This added flexibility allows Chromium users to solve more general problems than just scalability, such as integration with an existing user interface, stylized drawing, or application debugging. This extensibility is one of Chromium's key strengths. Because we simply provide a programmable filter mechanism for graphics API calls, Chromium can implement many different underlying algorithms. This model can be thought of as an extension of Voorhies's virtual graphics pipeline [33], which insulates applications from the details of the underlying implementations of a common API.

2 Background and Related Work

2.1 Cluster Graphics

Clusters have long been used for parallelizing traditionally non-interactive graphics tasks such as ray-tracing, radiosity [5, 25], and volume rendering [6]. Other cluster-parallel rendering efforts have largely concentrated on exploiting inter-frame parallelism rather than trying to make each individual frame run faster [20]. We are interested in enabling fast, interactive rendering on clusters, so these techniques tend to be at most loosely applicable to our domain.

In the last few years, there has been growing interest in using clusters for interactive rendering tasks. Initially, the goal of these

systems was to drive large tiled displays. Humphreys and Hanrahan described an early system designed for 3D graphics [9]. Although the system described in that paper ran on an SGI InfiniteReality, it was later ported to a cluster of workstations. At first, their cluster-based system, called WireGL, only allowed a single serial application to drive a tiled display over a network [7]. WireGL used traditional sort-first parallel rendering techniques to achieve scalable display size with minimal impact on the application's performance. The main drawback of this system was its poor utilization of the graphics resources available in a cluster. Because it only focused on display resolution, applications would rarely run faster on a cluster than they would locally.

Other approaches focused on scalable rendering rates. Samanta et al. described a cost-based model for load-balancing rendering tasks among nodes in a cluster, eventually redistributing the resulting non-overlapping pixel-tiles to drive a tiled display [29, 31]. They then extended this technique to allow for tile overlap, creating a hybrid sort-first and sort-last algorithm that could effectively drive a single display [30]. All of these algorithms required the full replication of the scene database on each node in the cluster, so further work was done to only require partial replication, trading off memory usage for efficiency [28]. Although these papers present an excellent study of differing data-management strategies in a clustered environment, they all provide *algorithms* rather than *mechanisms*. Applying these techniques to a big-data visualization problem would require significant reworking of existing software.

A different approach to dataset scalability was taken by Humphreys et al. when they integrated a parallel interface into WireGL [8]. By posing as the system's OpenGL driver, WireGL intercepts OpenGL commands made by an application (or multiple applications), and generates multiple new command sequences, each represented in a compact wire protocol. Each sequence is then transmitted over a network to a different server. Those servers manage image tiles, and execute the commands encoded in the streams on behalf of the client. Finally, the resulting framebuffer tiles are extracted and transmitted to a compositing server for display. Ordering between streams resulting from a parallel application is controlled using the parallel immediate mode graphics extensions proposed by Igehy et al [10]. WireGL can use either software-based image reassembly or custom hardware such as Lightning-2 [32] to reassemble the resulting image tiles and form the final output. This approach to cluster rendering allows existing applications to be parallelized easily, since it is built upon a popular, industry-standard API. However, by imposing a sort-first architecture on the resulting application, it can be difficult to load-balance the graphics work. Load-balancing is usually attempted by using smaller tiles, but this will tend to cause primitives to overlap more tiles, resulting in additional load on the network and reduced scalability. More fundamentally, WireGL requires that all geometry be moved over a network every frame, but today's networks are not fast enough to keep remote graphics cards busy.

2.2 Stream Processing

Continual growth in typical dataset size and network bandwidth has made stream-based analysis a hot topic for many different disciplines, such as telephone record analysis [4], multimedia, rendering of remotely stored 3D models [27], database queries [2], and theoretical computer science [16]. In these domains, streams are an appropriate computational primitive because large amounts of data arrive continuously, and it is impractical or unnecessary to retain the entire data set. In the broadest sense, a stream is a potentially infinite ordered sequence of records. Applications designed to operate on streams only access the elements of the sequence in order, although it is possible to buffer a portion of a stream for more global analysis. Any stream processing algorithm must operate on this potentially infinite input set using only finite resources.

Many of the traditional techniques used to solve problems in computer graphics can be thought of as stream processing algorithms. Immediate-mode rendering is a classic example. In this graphics model, an unbounded sequence of primitives is sent one at a time through a narrow API. The graphics system processes each primitive in turn, using only a finite framebuffer (and possibly texture memory) to store any necessary intermediate results. Because such a graphics system does not have memory of past primitives, its computational expressiveness is limited[1]. Owens et al. implemented an OpenGL-based polygon renderer on Imagine, a programmable stream processor [17]. Using Imagine, they achieved performance that is competitive with custom hardware while enabling greater programmability at each stage in the pipeline.

Mohr and Gleicher demonstrated that a variety of stylized drawing techniques could be applied to an unmodified OpenGL application by only analyzing and modifying the stream of commands [14]. They intercept the application's API commands by posing as the system's OpenGL driver, in exactly the same way Chromium obtains its command source. Although some of their techniques require potentially unbounded memory, some similar effects can be achieved using Chromium and multiple nodes in a cluster.

3 System Architecture

The overall design of Chromium was influenced by Stanford's WireGL system [8]. Although the sort-first architecture implemented by WireGL is fairly restrictive, one critical aspect of the design led directly to Chromium: The wire protocol used to move image tiles from the servers to the compositor is the same as the networked-OpenGL protocol used to move geometry from the clients to the servers. In effect, WireGL's servers themselves become clients of a second parallel rendering application, which uses imagery as its fundamental drawing primitive. This means that the compositing node is not special; in fact, it is just another instance of the same network server executing OpenGL commands and resolving ordering constraints on behalf of some parallel client.

If we consider a sequence of OpenGL commands to be a stream, WireGL provides three main stream "filters". First, it can sort a serial stream into tiles. Next, it can dispatch a stream to a local implementation of OpenGL. Finally, WireGL can read back a framebuffer and generate a new stream of image-drawing commands. In WireGL, these stream transformations can only be realized at specific nodes in the cluster (e.g., an application's stream can only be sorted). To arrive at Chromium's design, we realized that it would be useful to perform other transformations on API streams, and it would also be necessary to arrange cluster nodes in a more generic topology than WireGL's many-to-many-to-few arrangement.

3.1 Cluster Nodes

Chromium users begin by deciding which nodes in their cluster will be involved in a given parallel rendering run, and what communication will be necessary. This is specified to a centralized configuration system as a directed acyclic graph. Nodes in this graph represent computers in a cluster, while edges represent network traffic. Each node is actually divided into two parts: a *transformation* portion and a *serialization* portion.

[1] Because most graphics API's have some mechanism to force data to flow back towards the host (i.e., `glReadPixels`), graphics hardware is actually not a purely feed-forward stream processor. This fact has been exploited to perform more general computation using graphics hardware [18, 22], and extensions to the graphics pipeline have been proposed to further generalize its computational expressiveness [12].

The transformation portion of a node takes a single stream of OpenGL commands as input, and produces zero or more streams of OpenGL commands as output. The mapping from input to output is completely arbitrary. The output streams (if any) are sent over a network to another node in the cluster to be serialized and transformed again. Stream transformations are described in greater detail in section 3.2.

The serialization portion of a node consumes one or more independent OpenGL streams, each with its own associated graphics context, and produces a single OpenGL stream as output. This task is analogous to the scheduler in a multitasking operating system; the serializer chooses a stream to "execute", and copies that stream to its output until the stream becomes "blocked". It then selects another input stream, performs a context switch, and continues copying. Streams block and unblock via extensions to the OpenGL API that provide barriers and semaphores, as proposed by Igehy et al [10]. These synchronization primitives do not block the issuing process, but rather encode ordering constraints that will be enforced by the serializer. Because the serializer may have to switch between contexts very frequently, we use a hierarchical OpenGL state tracker similar to the one described by Buck et al [3]. This state representation allows for the efficient computation of the difference between two graphics contexts, allowing for fine-grained sharing of rendering resources.

A node's serializer can be implemented in one of two ways. Graph nodes that have one or more incoming edges are realized by Chromium's network server, and are referred to as *server nodes*. Servers manage multiple incoming network connections, interpreting messages on those connections as packed representations of OpenGL streams.

On the other hand, nodes that have no incoming edges must generate their (already serial) OpenGL streams programmatically. These nodes are called *client nodes*. Clients obtain their streams from standalone applications that use the OpenGL API. Chromium's application launcher causes these programs to load our OpenGL shared library on startup. Chromium's OpenGL library injects the application's commands into the node's stream transformer, so the application does not have to be modified to initialize or load Chromium. If there is only one client in the graph, it will typically be an unmodified off-the-shelf OpenGL application. For graphs with multiple clients, the applications will have to specify the ordering constraints on their respective streams.

3.2 OpenGL Stream Processing

Stream transformations are performed by OpenGL "Stream Processing Units", or SPUs. SPUs are implemented as dynamically loadable libraries that provide the OpenGL interface, so each node's serializer will load the required libraries at run time and build an OpenGL dispatch table. SPUs are normally designed as generically as possible so they can be used anywhere in a graph.

A simple example configuration is shown in figure 1. The client loads the `tilesort` SPU, which incorporates all of the sort-first stream processing logic from WireGL. The servers use the `render` SPU, which dispatches the incoming streams directly to their local graphics accelerators. This configuration has the effect of running the unmodified client application on a tiled display using sort-first stream processing, giving identical semantics and similar performance to the tiled display system described by Humphreys et al [7]. Notice that in figure 1, the graph edges originate from the `tilesort` SPU, not the application itself. This convention is used because the SPU in fact manages its own network resources, originates connections to servers, and generates traffic.

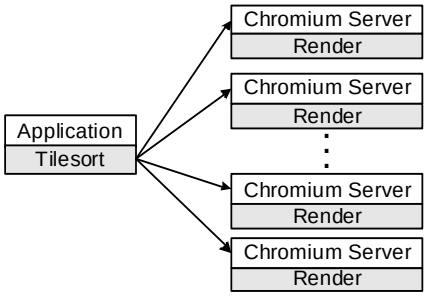


Figure 1: A simple Chromium configuration. In this example, a serial application is made to run on a tiled display using a sort-first stream processor called `tilesort`.

3.3 SPU Chains

A node's stream transformation need not be performed by only a single SPU; serializers can load a linear chain of SPUs at run time. During initialization, each SPU receives an OpenGL dispatch table for the next SPU in its local chain, meaning simple SPUs can be chained together to achieve more complex results. Using this feature, a SPU might intercept and modify (or discard) calls to one particular OpenGL function and pass the rest untouched to its downstream SPU. This allows a SPU, for example, to adjust the graphics state slightly to achieve a different rendering style.

One example of such a SPU is a "wireframe style" filter. This SPU issues a `glPolygonMode` call to its downstream SPU at startup to set the drawing mode to wireframe. It then passes all OpenGL calls directly through except `glPolygonMode`, which it discards, preventing the application from resetting the drawing mode. Note that Chromium does not require a stream to be rendered on a different node from where it originated; it is straightforward for the client to load the `render` SPU as part of its chain. In this way, an application's drawing style can be modified while it runs directly on the node's graphics hardware, without any network traffic.

SPU chains are always initialized in back-to-front order, starting with the final SPU in the chain. At initialization, a SPU must return a list of all the functions that it implements. A SPU that wants to pass a function call through to the SPU immediately downstream can return the downstream SPU's function pointer as its own. Because there is no indirection in this model, passing OpenGL calls through multiple SPUs does not incur any performance overhead. Such function pointer copying is common in Chromium; as long as SPUs copy and change OpenGL function tables using only our provided API's, they can change their own exported interface on the fly and automatically propagate those changes throughout the node.

3.4 SPU Inheritance

A SPU need not export a complete OpenGL interface. Instead, SPUs benefit from a single-inheritance model in which any functions not implemented by a SPU can be obtained from a "parent", or "super" SPU. The SPU most commonly inherited from is the `passthrough` SPU, which passes all of its calls to the next SPU in its node's chain. The wireframe drawing SPU mentioned in the previous section would likely be implemented this way—it would implement only `glPolygonMode`, and rely on the `passthrough` SPU to handle all other OpenGL functions. At initialization, each SPU is given a dispatch table for its parent. When the wireframe SPU wishes to set the drawing mode to wireframe during initialization, it calls the `passthrough` SPU's implementation of `glPolygonMode`.

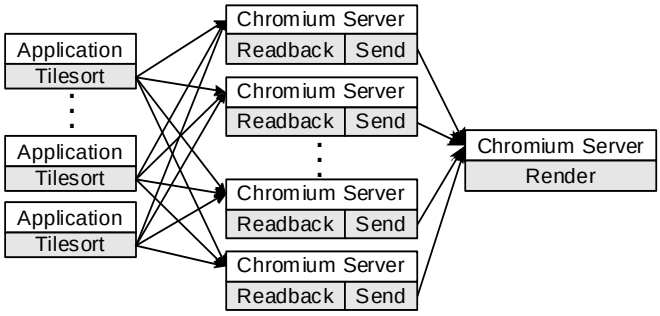


Figure 2: Chromium configured as a complete WireGL replacement. A parallel application drives a tiled display using the sort-first logic in the `tilesort` SPU. Imagery is then read back from the servers managing those tiles and sent to a final compositing server for display.

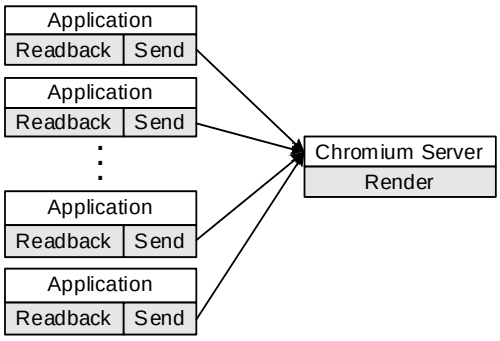


Figure 3: Another possible Chromium configuration. In this example, nodes in a parallel application render their portion of the scene directly to their local hardware. The color and depth buffers are then read back and transmitted to a final compositing server, where they are combined to produce the final image.

3.5 Provided Tools and SPUs

Chromium provides four libraries that encapsulate frequently performed stream operations. The first is a stream packing library. This library takes a sequence of commands and produces a serialized encoding of the commands and their arguments. Although this library is normally used to prepare commands for network transmission, it can also be used to buffer a group of commands for later analysis, as described in section 4.3. We use a very similar encoding method to the one described by Buck et al [3]. It incurs almost no wasted space, retains natural argument alignment, and allows a group of command "opcodes" and their arguments to be sent with a single call to the networking library.

Second, we provide a stream unpacking library. This library decodes an already serialized representation of a sequence of commands and dispatches those commands to a given SPU. This library is primarily used by Chromium's network server to handle incoming network traffic, but it can also be used by SPUs that need to locally buffer a portion of a stream in order to perform more global analysis or make multiple passes over that portion.

The third is a point-to-point connection-based networking abstraction. This library abstracts the details of the underlying transport mechanism; we have implemented this API on top of TCP/IP and Myrinet. In addition, the library can be used by SPUs and applications to communicate with each other along channels other than those implied by the configuration graph described in section 3.1. This out-of-band communication allows complex compositing SPUs to be built, such as the one described in section 4.1.

Finally, Chromium includes a complete OpenGL state tracker. In addition to maintaining the entire OpenGL state, this library can efficiently compute the difference between two graphics contexts, generating a call to a given SPU for every discrepancy found. This efficient context differencing operation is due to a hierarchical representation described by Buck et al [3].

In addition to these support libraries, Chromium provides a number of SPUs that can be used as is or extended to realize the desired stream transformation. There are too many SPUs to list here; a complete list can be found in the Chromium documentation at `http://chromium.sourceforge.net`.

3.6 Realizing Parallel Rendering Architectures

We now present two examples of parallel rendering architectures that can be realized using Chromium. As described by Molnar et al., parallel rendering architectures can be classified according to the point in the graphics pipeline at which data is "sorted" from an object-parallel distribution to an image-parallel distribution [15].

The first configuration, shown in figure 2, shows a sort-first graphics architecture that functions identically to WireGL. As in figure 1, we use the `tilesort` SPU to sort the streams into tiles. Each intermediate server serializes its incoming streams and passes the result to the `readback` SPU. The `readback` SPU inherits from the `render` SPU using the mechanism described in section 3.4, so the streams are rendered on the locally housed graphics hardware. However, the `readback` SPU provides its own implementation of `SwapBuffers`, so at the end of the frame it extracts the framebuffer and uses `glDrawPixels` to pass the pixel data to another SPU. In the figure, each pixel array is passed to a `send` SPU, which transmits the data to a final server for tile reassembly. Each `readback` SPU is configured at startup to know where its tiles should end up in the final display; these coordinates are passed to the `send` SPU using `glRasterPos`. The `readback` SPU also uses Igehy's parallel graphics synchronization extensions [10] to ensure that the tiles all arrive at their destination before the final rendering server displays its results. This final tile reassembly step could also be performed using custom hardware such as Lightning-2 [32].

A dramatically different architecture is shown in figure 3. In this figure, the `readback` SPU is loaded directly by the applications. Recall that the `readback` SPU dispatches all of the OpenGL API directly to the underlying graphics hardware, so the application running in this configuration benefits from the full performance of local 3D acceleration. In this case, the `readback` SPU is configured to extract both the color and depth buffers, sending them both to a final compositing server along with the appropriate OpenGL commands to perform a depth composite. In contrast to WireGL, this is a sort-last architecture. In practice, having many full framebuffers arriving at a single display server would be a severe bottleneck, so this architecture is rarely used. In addition, when doing depth compositing in Chromium, it can be beneficial to write a special SPU to perform the composite in software, because compositing depth images in OpenGL requires using the stencil buffer in a way that is quite slow on many architectures. A more advanced (and practical) Chromium-based sort-last architecture is presented in section 4.1.

Because Chromium provides a virtual graphics pipeline with a parallel interface, the application in figure 3 could be run unmodified on the architecture in figure 2 simply by specifying a different configuration DAG. The architectures may provide different semantics (e.g., the sort-last architecture cannot guarantee ordering constraints), but the application need not be aware of the change.

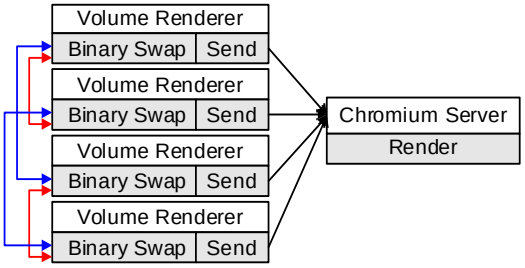


Figure 4: Configuration used for a four-node version of our cluster-parallel volume rendering system. Each client renders its local portion of the volume using local graphics hardware. Next, the volumes are composited using the `binaryswap` SPU. The SPUs use out-of-band communication to exchange partial framebuffers until each SPU contains one quarter of the final image. These partial images are then sent to a single server for display.

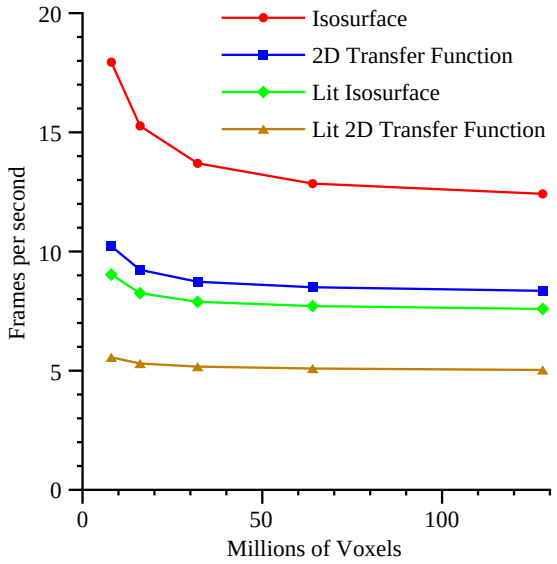


Figure 5: Performance of our volume renderer as larger volumes are used. In this graph, each node renders a 256×256×128 subvolume to a 1024×256 window. The data points correspond to a cluster of 1, 2, 4, 8, and 16 nodes. At 16 nodes, we are rendering two copies of the full 256×256×1024 dataset.

4 Results

In this section, we present three different Chromium usage scenarios: a parallel volume renderer used to interactively explore a large volumetric dataset, the reintegration of an application's graphics stream into its original user interface on a high-resolution display device, and a stream transformation to achieve a non-photorealistic drawing style.

4.1 Parallel Volume Rendering

Our volume rendering application uses 3D textures to store volumes and renders them with view-aligned slicing polygons, composited from back to front. Using Stanford's Real-Time Shading Language [22], we can implement different classification and shading schemes using the latest programmable graphics hardware, such as NVIDIA's GeForce3. Small shaders can easily exhaust these cards' resources; for example, a shader that implements a simple 2D transfer function and a specular shading model requires two 3D texture lookups, one 2D texture lookup (dependent on one of the 3D lookups), and all eight register combiners.

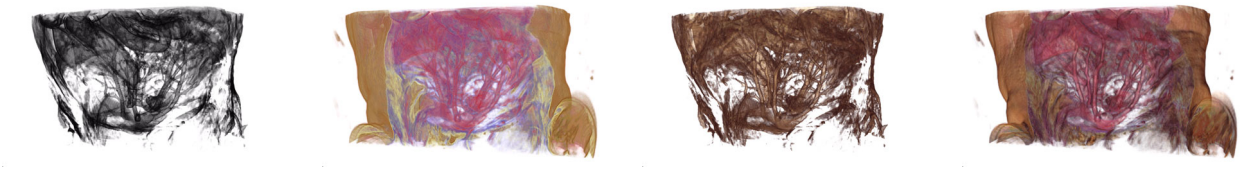
Because we store our volumes as textures, the maximum size of the volume that can be rendered is limited by the amount of available texture memory. In practice, on a single GeForce3 with 64 MB of texture memory, the largest volume that can be rendered with the shader described above is 256×256×128. In addition, the speed of volume rendering with 3D textures is limited by the fill rate of our graphics accelerator. While the theoretical fill rate of the GeForce3 is 800 Mpix/sec, complex fragment processing greatly decreases the attainable performance. Depending on the complexity of the shader being used, we achieve between 42 and 190 Mpix/sec, or roughly 5% to 24% of the GeForce3's theoretical peak fill rate.

Both of these limitations can be mitigated by parallelizing the rendering across a cluster. We first divide the volume among the nodes in our cluster. Each node renders its subvolume on locally housed graphics hardware using the `binaryswap` SPU, which composites the resulting framebuffers using the "binary swap" technique described by Ma et al [11]. In this technique, rendering nodes are first grouped into pairs. Each node sends one half of its image to its counterpart, and receives the other half of its counterpart's image. This communication uses Chromium's connection-based networking abstraction, described in section 3.5. The SPUs then composite the image they received with their local framebuffer. This newly composited sub-region of the image is then split in half, a different pairing is chosen, and the process repeats. If there are n nodes in our cluster, after $\log(n)$ steps each node will have completely composited $\frac{1}{n}$ of the total image. Because we are compositing transparent images using Porter and Duff's "over" operator [21], the sequence of pairings is chosen carefully so that blending is performed in the correct order with respect to the viewpoint.

Our scalability experiments were conducted on a cluster of sixteen nodes, each running RedHat Linux 7.2. The nodes contain an 800 MHz Pentium III Xeon, a GeForce3 with 64 MB of video memory, 256 MB of main memory, and a Myrinet network with a maximum bandwidth of approximately 100 MB/sec. The dataset is a 256×256×1024 magnetic resonance scan of a mouse. All of our renderings are performed in a window of size 1024×256, ensuring that each voxel is sampled exactly once. Table 1 shows the four shaders we used to vary the achievable per-node performance.

Figure 4 shows the Chromium communication graph for a cluster of four nodes. Note that a minimum of eight nodes is required to render the full mouse volume, because each node in our cluster has only 64 MB of texture memory. Figure 5 shows the performance of our volume renderer as the size of the volume is scaled. In this experiment, we rendered a portion of the mouse dataset on each node in our cluster. The initial drop in performance is due to the additional framebuffer reads required, but because the binary swap algorithm keeps all the nodes busy while compositing, the graph flattens out, and we sustain nearly constant performance as the size of the volume is repeatedly doubled. At 16 nodes, we render two copies of the full 256×256×1024 volume at a rate between 643 MVox/sec and 1.59 GVox/sec, depending on the shader used.

If we instead fix the size of the volume and parallelize the rendering, we quickly become limited by our pixel readback and network performance. When rendering a single 256×256×128 volume split across multiple nodes, the rendering rate rapidly becomes negligible. When creating a 1024×256 image, our volume renderer's performance converges to approximately 22 frames per second. Because the parallel image compositing and final transmission for display happen sequentially, we can analyze this performance as follows: With 16 nodes, each node eventually extracts and sends $\frac{15}{16}$



Isosurface | 2D Transfer Function | Lit Isosurface | Lit 2D Transfer Function

Shader	3D textures	2D (dependent) textures	Register Combiners	Single-Node Fill Rate (Mpix/sec)
Isosurface	1	0	6	190
2D Transfer Function	1	1	4	98
Lit Isosurface	2	0	8	78
Lit 2D Transfer Function	2	1	8	42

Table 1: Shaders used in our volume rendering experiments. The lit 2D transfer function shader exhausts the resources of a GeForce3. Mouse dataset courtesy of the Duke Center for In Vivo Microscopy.

of its framebuffer, requiring four bytes per pixel. The final transmission sends only $\frac{1}{16}$ of a framebuffer at three bytes per pixel, but because all of these framebuffer portions arrive at the same node, we must consider the aggregate incoming bandwidth at that node, which is a full framebuffer at three bytes per pixel. This adds up to 1.69 MB/frame, or 37.1 MB/sec. This measurement is close to our measured RGBA readback performance of the GeForce3, which is clearly the limiting factor for the `binaryswap` SPU, since our network can sustain 100 MB/sec. Future improvements in pixel readback rate and network bandwidth would result in higher framerates, as would an alpha-compositing mode for a post-scanout compositing system such as Lightning-2.

4.2 Integration With an Existing User Interface

Normally, when Chromium intercepts an application's graphics commands, that application's graphics window will be blank, with the rendering appearing in one or more separate windows, potentially distributed across multiple remote computers. Because the interface is now separated from the visualization, this can interfere with the productive use of some applications. To address this problem, we have implemented the `integration` SPU to reincorporate remotely rendered tiles into the application's user interface. This way, users can apply a standard user interface to a parallel client.

This manipulation can also be useful for serial applications. Even though the net effect is a null transformation on the application's stream, it can aid in driving high resolution displays. For our experiments, we use the IBM T221, a 3840×2400 LCD. Few graphics cards can drive this display directly, and those that can do not have sufficient scanout bandwidth to do so at a high refresh rate. The T221 can be driven by up to four separate synchronized digital video inputs, so we can achieve higher bandwidth to the display using a cluster and special hardware such as Lightning-2 [32], or a network-attached parallel framebuffer such as IBM's Scalable Graphics Engine (SGE) [19]. The SGE supports up to 16 one-gigabit ethernet inputs, can double buffer up to 16 million pixels, and can drive up to eight displays. In our tests, we used the SGE to supply four synchronized DVI outputs that collectively drive the T221 at its highest resolution. An X-Windows server for the SGE provides a standard user interface for this configuration.

The `integration` SPU is conceptually similar to the `readback` SPU in that it inherits almost all of its functionality from the `render` SPU. To extract the color information from the framebuffer, the `integration` SPU implements its own `SwapBuffers` handler, which uses the SGE to display those pixels on the T221. The configuration graph used to conduct this experiment is shown in figure 8. The application's graphics stream is sorted into tiles managed by multiple Chromium servers, each of which dispatches its tile's stream to the `integration` SPU. The integration SPU places the resulting pixels into X regions by *tunneling*, meaning that the pixels are transferred to the SGE's framebuffer without the involvement of the X server that manages the display. Because the SGE supports multiple simultaneous writes to the framebuffer, this technique does not unnecessarily serialize tile placement. Note that the number of tiles sent to the SGE is independent of the number of the SGE's outputs, so we use an 8-node cluster to drive the four outputs at interactive rates.

The `integration` SPU must also properly handle changes to the size of the application's rendering area. When an application window is resized, it will typically call `glViewport` to reset its drawing area. Accordingly, the `integration` SPU overrides the `render` SPU's implementation of `glViewport` to detect these changes, and adjusts the size of the render tiles if necessary. Because the `tilesort` SPU sorts based on a logical decomposition of the screen, it does not need to be notified of this change[2].

Although the `integration` SPU enables functionality that is not otherwise possible, it is still important that it not impede interactivity. For our performance experiments, we used a cluster of eight nodes running RedHat Linux 7.1, each with two 866MHz Pentium III Xeon CPUs, 1GB of RDRAM, NVIDIA Quadro graphics, and both gigabit ethernet and Myrinet 2000 networking. One of our cluster nodes runs the SGE's X-windows server in addition to the Chromium server. We successfully tested applications ranging from trivial (simple demos from the GLUT library) to a medium-complexity scientific visualization application (OpenDX) to a closed-source, high-complexity CAD package (CATIA).

The graph shown in figure 6 shows the average frame rate as we scale the display resolution of the T221 from 800×600 to 3840×2400. Four curves are shown, corresponding to a cluster of 1, 2, 4, and 8 nodes. Because we want to measure only the performance impact of the `integration` SPU, we rendered only small amounts of geometry (approximately 5000 vertices per frame) using the GLUT `atlantis` demo. This demo runs at a much greater rate than the refresh rate of the display, so its effect on performance is minimal compared to the expense of extracting and transmitting tiles.

The maximum frame rate achieved using 4 or 8 nodes is 41 Hz, which is exactly the vertical refresh rate of the T221. Because

[2]Our example application uses only geometric primitives. In order for pixel-based primitives to be rendered correctly, the `tilesort` SPU would need to be notified when the window size changes. Alternately, the tilesort SPU could be configured to broadcast all `glDrawPixels` calls.

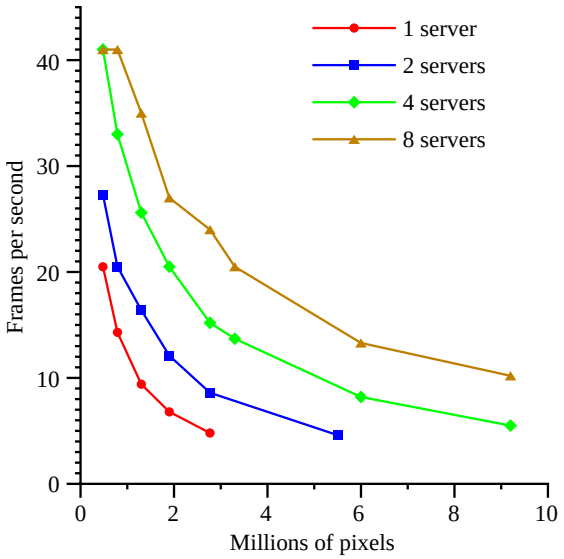


Figure 6: Performance of the GLUT "atlantis" demo using the `integration` SPU to drive the T221 display at different resolutions. Each curve shows the relationship between performance and resolution for a given number of rendering servers. For smaller windows, the SPU becomes limited by the vertical refresh rate of the display (41 Hz). As the resolution approaches 3840×2400 (9.2 million pixels), a small 8-server configuration still achieves interactive refresh rates.

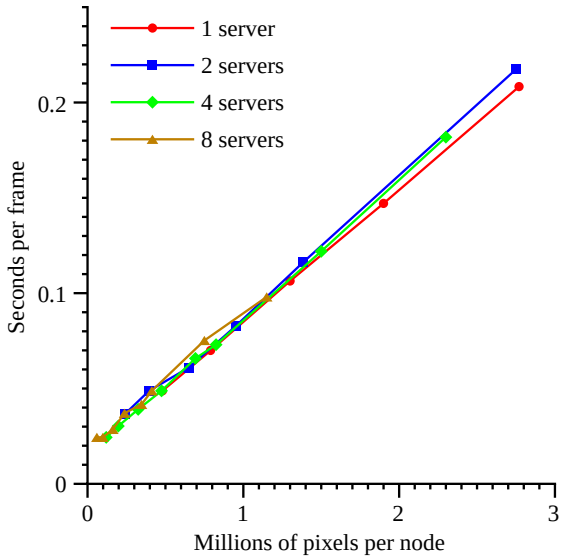


Figure 7: We have replotted the data from figure 6 to show *seconds per frame* versus pixels *per node*, to show per-node throughput. The coincidence of the four curves shows that there is insignificant overhead to doubling the number of rendering nodes, so linear speedup is achieved until the monitor refresh rate becomes the limiting performance factor.

the SGE requires hardware synchronization to the refresh rate, no higher frame rate can be achieved. For a given fixed resolution, the `integration` SPU achieves the expected performance increase as more rendering nodes are used, because this application is completely limited by the speed at which we can redistribute pixels. Figure 7 shows this phenomenon more clearly. In this graph, the same data are plotted showing seconds per frame rather than frames per second. In addition, the data have been normalized by the number of nodes used, so the quantity being measured is the pixel throughput per node. The coincidence of the four curves shows that there is no penalty associated with adding rendering nodes, so linear speedup is achieved until the display's refresh rate becomes the limiting factor. The rate at which each node can read back pixels and send them to the SGE is given by the slope of the line, which is approximately 12 MPix/second/node, or 48 MB/second/node. Extrapolating to a very small image size, the system overhead is approximately 15 milliseconds, which indicates that the maximum system response rate of the `integration` SPU is approximately 70 Hz (in the absence of monitor refresh rate limitations).

The measurements presented here give a worst-case scenario for the `integration` SPU, in which it is responsible for almost 100% of the overhead in the system. We are able to demonstrate frame rates exceeding 40Hz using only 8 nodes, and achieve an interactive 10 Hz even with each node supplying over one million pixels per frame. In addition, if measured independently, pixel readback rate and the SGE transfer rate can both provide bandwidths exceeding 23 Mpix/sec, nearly twice what they achieve when measured together. This leads us to believe that the system I/O bus or memory subsystem is under-performing when these two tasks are being performed simultaneously, an effect that will likely be eliminated with the introduction of new I/O subsystems designed specifically for high-end servers. This is a similar contention effect to that observed by Humphreys et al. when evaluating WireGL on a cluster of SMP nodes [8].

4.3 Stylized Drawing

For a long time, research on non-photorealistic, or "stylized", rendering focused on non-interactive, batch-mode techniques. In recent years, however, there has been considerable interest in real-time stylized rendering. Early interactive NPR systems required *a priori* knowledge of the model and its connectivity [13, 26]. More recently, Raskar has shown that non-trivial NPR styles can be achieved with no model analysis using either standard graphics pipeline tricks [24] or slight extensions to modern programmable graphics hardware [23].

We have developed a simple stylized rendering filter that creates a flat-shaded hidden-line drawing style. Our approach is similar to that taken by Mohr and Gleicher [14], although we show a technique that requires only finite storage. Hidden line drawing in OpenGL is a straightforward multi-pass technique, accomplished by first rasterizing all polygons to the depth buffer, and then re-rasterizing the polygon edges. The polygon depth values are offset using `glPolygonOffset` to reduce aliasing artifacts [1].

Achieving this effect in Chromium can be accomplished with a single SPU. The `hiddenline` SPU packs each graphics command into a buffer as if they were being prepared for network transport. This has the effect of recording the entire frame into local memory. Instead of actually sending them to a server, we instead decode the commands twice at the end of each frame, once as polygons and once as lines, achieving our desired style. The code required to achieve this transformation is shown in figure 9, and the visual results are shown in figure 10. The performance impact of this SPU is shown in figure 11.

There are three interesting notes regarding the actual implementation of a `hiddenline` SPU. First, the application may generate state queries that need to be satisfied immediately and not recorded. In order to do this, the entire graphics state is maintained using our state tracking library, and any function that might affect the state is passed to the state tracker before being packed. This behavior is frequently overly cautious; most state queries are attempts to deter-

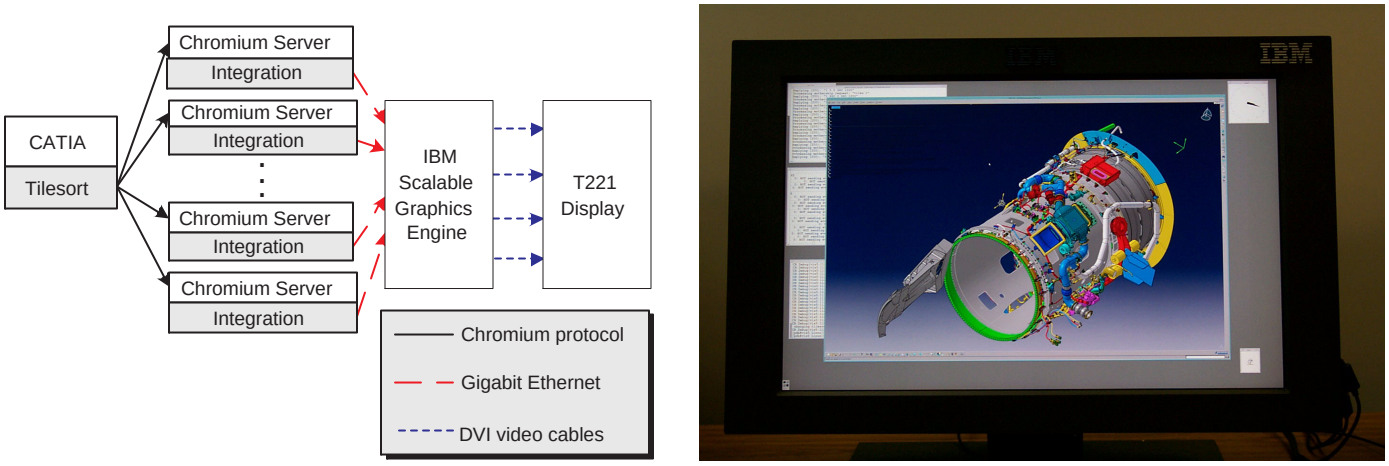


Figure 8: Configuration used to drive IBM's 3840×2400 T221 display using Chromium. The commercial CAD package CATIA is used to create a tiled rendering of a jet engine nacelle (model courtesy of Goodrich Aerostructures). The tiles are then re-integrated into the application's original user interface, allowing CATIA to be used as designed, despite the distribution of its graphics workload on a cluster. Due to the capacity and range of gigabit ethernet, all of the computational and 3D graphics hardware can be remote from the eventual display.

mine some fundamental limit of the graphics system (such as the maximum size of a texture), rather than querying state that was set by the application itself. Robust implementations of style filters like the `hiddenline` SPU would likely benefit from the ability to disable full state tracking.

Second, the SPU does not play back the exact calls made in the frame. Because we want to draw all polygons in the same color (and similarly for lines), the application must be prevented from enabling texturing, changing the current color, turning on lighting, changing the polygon draw style, enabling blending, changing the line width, disabling the depth test, or disabling writes to the depth buffer. To accomplish this, a new OpenGL dispatch table is built, containing mostly functions from the SPU immediately following the `hiddenline` SPU in its chain, but with our own versions of `glEnable`, `glDisable`, `glDepthMask`, `glPolygonMode`, `glLineWidth`, and all the `glColor` variants, which enforce these rules. Applications which rely on complex uses of these functions may not function properly using this SPU.

Finally, some care must be taken to properly handle vertex arrays. Because the semantics of vertex arrays allow for the data buffer to be changed (or discarded) after it is referenced, we cannot store vertex array calls verbatim and expect them to decode properly later in the frame. Instead, we transform uses of vertex arrays back into sequences of separate OpenGL calls. Although this could be done by the `hiddenline` SPU itself, we have found this transformation to be useful in other situations, so we have implemented the vertex array filtering in a separate `vertexarray` SPU. This SPU appears immediately before the `hiddenline` SPU in figure 10.

It should be noted that the `hiddenline` SPU as presented requires potentially infinite storage, since it buffers the entire frame, and therefore cannot be considered a true stream processor. There are two possible solutions to this problem. One is to perform primitive assembly in the `hiddenline` SPU, drawing each stylized primitive separately. This technique does satisfy our resource constraints (extremely large polygonal primitives can be split into smaller ones), but would result in a significant performance penalty for applications with a high frame rate, due to the overhead of software primitive assembly as well as the frequent state changes.

A better solution to this problem is to use multiple cluster nodes, as shown in figure 12. Rather than buffering the entire frame, we send the entire stream verbatim to two servers, one rendering the incoming stream as depth-offset polygons, the other as lines. Instead of writing two new SPUs for each of these rendering styles, we would inject the appropriate OpenGL calls into the streams before transmission. We then use the `readback` and `send` SPUs to combine the two renderings using a depth-compositing network, as described in section 3.6. Note that we could more economically use our resources by rendering depth-offset polygons locally and forwarding the stream to a single line-rendering node (or vice versa), thereby requiring only three nodes instead of four, although this would require a more complex implementation.

```
void hiddenline_SwapBuffers( void )
{
  /* Draw filled polygons */
  super.Clear( color and depth );
  super.PolygonOffset( 1.5f, 0.000001f );
  super.PolygonMode( GL_FRONT_AND_BACK, GL_FILL );
  super.Color3f( poly_r, poly_g, poly_b );
  PlaybackFrame( modified_child_dispatch );

  /* Draw outlined polygons */
  super.PolygonMode( GL_FRONT_AND_BACK, GL_LINE );
  super.Color3f( line_r, line_g, line_b );
  PlaybackFrame( modified_child_dispatch );

  super.SwapBuffers();
}
```

Figure 9: End-of-frame logic for a simple hidden-line style SPU. The entire frame is played back twice, once as depth-offset filled polygons, and once as lines. We modify the downstream SPU's dispatch table to discard calls that would affect our drawing style, such as texture enabling and color changes.

5 Discussion and Future Work

In their seminal paper on virtual graphics, Voorhies, Kirk and Lathrop note that providing a level of abstraction between an applica-

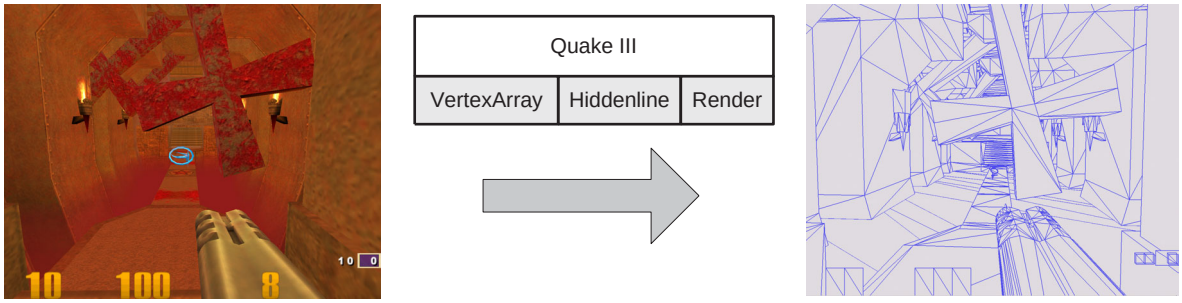


Figure 10: Drawing style enabled by the `hiddenline` SPU. After uses of vertex arrays are filtered out, the SPU records the entire frame, and plays it back twice to achieve a hidden-line effect. No high-level knowledge of the model is required.

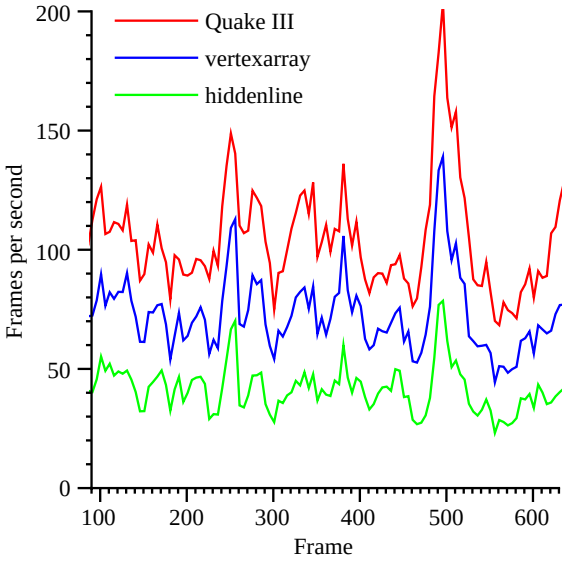


Figure 11: Performance of Quake III running a prerecorded demo. The first 90 frames are devoted to an introductory splash screen and are not shown here. The red curve shows the performance achieved by the application alone. The blue curve shows the same demo using just the `vertexarray` SPU, and the green curve gives the performance of the demo rendering with a hidden-line style. Despite more than a 2:1 reduction in speed, the demo still runs at approximately 40-50 frames per second.

tion and the graphics hardware "allows for cleaner software design, higher performance, and effective concurrent use of the display" [33]. We believe that the power and implications of these observations have not yet been fully explored. Chromium provides a compelling mechanism with which to further investigate the potential of virtual graphics. Because Chromium provides a complete graphics API (many of the key SPUs such as `tilesort`, `send`, and `render` pass almost all of the OpenGL conformance tests), it is no longer necessary to write custom applications to test new ideas in graphics API processing. Also, the barrier to entry is quite low; for example, the `hiddenline` SPU described in section 4.3 adds only approximately 250 lines of code to Chromium's SPU template.

In the future, we would like to see Chromium applied to new application domains, especially new ideas in scalable interactive graphics on clusters. Of particular interest is the problem of managing enormous time-varying datasets, both volumetric and polygonal. Today's time-varying volumetric datasets can easily exceed 30 terabytes in size. We intend to build a new parallel rendering application designed specifically for interactively visualizing these datasets on a cluster, using Chromium as the underlying transport, rendering, and compositing mechanism.

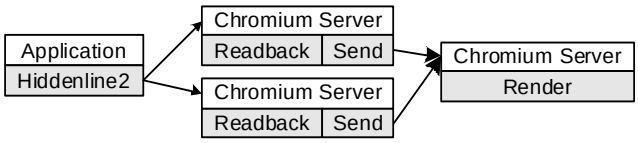


Figure 12: A different usage model for achieving a hidden-line drawing style. In this example, the filled polygon stream and the wireframe stream are sent to two different rendering servers and the resulting images are depth composited. This way, no single SPU needs to buffer the entire frame, and the system requires only finite resources.

We are particularly interested in building infrastructure to support flexible remote graphics. We believe that a clean separation between a scalable graphics resource and the eventual display has the potential to change the way we use graphics every day. We are actively pursuing a new direction to make scalable cluster-based graphics appear as a remote, shared service akin to a network mounted filesystem.

Most of all, we hope that Chromium will be adopted as a common low-level mechanism for enabling new graphics algorithms, particularly for clusters. If this happens, research results in cluster graphics can more easily be applied to existing problems outside the original researcher's lab.

6 Conclusions

We have described Chromium, a flexible framework for manipulating streams of graphics API commands on clusters of workstations. Chromium's stream processors can be configured to provide a sort-first parallel rendering architecture with a parallel interface, or a sort-last architecture capable of handling most of the same applications. Chromium's flexibility makes it an ideal launching point for new research in parallel rendering systems, particularly those that target clusters of commodity hardware. In addition, it is likely that Chromium's stream-processing model can be applied to other problems in visualization and computer illustration.

Acknowledgments

The authors would like to thank Brian Paul and Alan Hourihaine for their tireless efforts to make Chromium more robust. Allan Johnson, Gary Cofer, Sally Gewalt, and Laurence Hedlund from the Duke Center for In Vivo Microscopy (an NIH/NCRR National Resource) provided the dataset for our volume renderer. Kekoa Proudfoot and Bill Mark provided assistance with the implementation of a volume renderer on top of the Stanford Real Time Shading Language. Finally, we would like to especially thank all the WireGL and Chromium users for their continued support. This work was funded by DOE contract B504665, and was also 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 (UCRL-JC-146802).

References

[1] *Advanced Graphics Progamming Techniques Using OpenGL*. SIGGRAPH 1998 Course Notes.

[2] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, pages 109–120, September 2001.

[3] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 87–95, August 2000.

[4] Corrina Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rodgers, and Frederick Smith. Hancock: A language for extracting signatures from data streams. *Proceedings of 2000 ACM SIGKDD International Conference on Knowledge and Data Mining*, pages 9–17, August 2000.

[5] Thomas Funkhouser. Coarse-grained parallelism for hierarchical radiosity using group iterative methods. *Proceedings of SIGGRAPH 96*, pages 343–352, August 1996.

[6] Christopher Giertsen and Johnny Peterson. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, pages 16–23, November 1993.

[7] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed rendering for scalable displays. *IEEE Supercomputing 2000*, October 2000.

[8] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. WireGL: A scalable graphics system for clusters. *Proceedings of SIGGRAPH 2001*, pages 129–140, August 2001.

[9] Greg Humphreys and Pat Hanrahan. A distributed graphics system for large tiled displays. *IEEE Visualization '99*, pages 215–224, October 1999.

[10] Homan Igehy, Gordon Stoll, and Pat Hanrahan. The design of a parallel graphics interface. *Proceedings of SIGGRAPH 98*, pages 141–150, July 1998.

[11] Kwan-Liu Ma, James Painter, Charles Hansen, and Michael Krogh. Parallel volume rendering using binary-swap image compositing. *IEEE Computer Graphics and Applications*, pages 59–68, July 1994.

[12] William Mark and Kekoa Proudfoot. The F-buffer: A rasterization order FIFO buffer for multi-pass rendering. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57–64, August 2001.

[13] Lee Markosian, Michael Kowalski, Samuel Trychin, Lubomir Bourdev, Daniel Goldstein, and John Hughes. Real-time nonphotorealistic rendering. *Proceedings of SIGGRAPH 1997*, pages 415–420.

[14] Alex Mohr and Michael Gleicher. Non-invasive, interactive, stylized rendering. *ACM Symposium on Interactive 3D Graphics*, pages 175–178, March 2001.

[15] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Algorithms*, pages 23–32, July 1994.

[16] Liadan O'Callaghan, Nina Mishra, Adam Meyerson, Sudipto Guha, and Rajeev Motwani. Streaming-data algorithms for high-quality clustering. To appear in *Proceedings of IEEE International Conference on Data Engineering*, March 2002.

[17] John Owens, William Dally, Ujval Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon rendering on a stream architecture. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 23–32, August 2000.

[18] Mark Peercy, Marc Olano, John Airey, and Jeffrey Ungar. Interactive multi-pass programmable shading. *Proceedings of SIGGRAPH 2000*, pages 425–432, August 2000.

[19] Kenneth Perrine and Donald Jones. Parallel graphics and interactivity with the scaleable graphics engine. *IEEE Supercomputing 2001*, November 2001.

[20] Pixar animation studios. *PhotoRealistic RenderMan Toolkit*. 1998.

[21] Thomas Porter and Tom Duff. Compositing digital images. *Proceedings of SIGGRAPH 84*, pages 253–259, July 1984.

[22] Kekoa Proudfoot, William Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real time procedural shading system for programmable graphics hardware. *Proceedings of SIGGRAPH 2001*, pages 159–170, August 2001.

[23] Ramesh Raskar. Hardware support for non-photorealistic rendering. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 41–46, August 2001.

[24] Ramesh Raskar and Michael Cohen. Image precision silhouette edges. *ACM Symposium on Interactive 3D Graphics*, pages 135–140, April 1999.

[25] Rodney Recker, David George, and Donald Greenberg. Acceleration techniques for progressive refinement radiosity. *ACM Symposium on Interactive 3D Graphics*, pages 59–66, 1990.

[26] Jareck Rossignac and Maarten van Emmerik. Hidden contours on a framebuffer. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, September 1992.

[27] Szymon Rusinkiewicz and Marc Levoy. Streaming QSplat: A viewer for networked visualization of large, dense models. *ACM Symposium on Interactive 3D Graphics*, pages 63–68, 2001.

[28] Rudrajit Samanta, Thomas Funkhouser, and Kai Li. Parallel rendering with k-way replication. *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, October 2001.

[29] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Sort-first parallel rendering with a cluster of PCs. *SIGGRAPH 2000 Technical Sketch*, August 2000.

[30] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 97–108, August 2000.

[31] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 107–116, August 1999.

[32] Gordon Stoll, Matthew Eldridge, Dan Patterson, Art Webb, Steven Berman, Richard Levy, Chris Caywood, Milton Taveira, Stephen Hunt, and Pat Hanrahan. Lightning-2: A high-performance display subsystem for PC clusters. *Proceedings of SIGGRAPH 2001*, pages 141–148, August 2001.

[33] Douglas Voorhies, David Kirk, and Olin Lathrop. Virtual graphics. *Proceedings of SIGGRAPH 88*, pages 247–253, August 1988.

Editors: Lawrence Rosenblum and Michael Macedonia

Deep View: High-Resolution Reality

James T. Klosowski, Peter D. Kirchner, Julia Valuyeva, Greg Abram, Christopher J. Morris, Robert H. Wolfe, and Thomas Jackman

IBM T.J. Watson Research Center

Complex geometric models more accurately represent actual physical objects, giving virtual objects more realism. For example, we can model a virtual ball using a tessellated sphere with many tiny triangles to better approximate the ball's curvature. However, even when using a million triangles in the geometric model of that ball, if we're only rendering those triangles into a window of 250,000 pixels (500 × 500), then at most 25 percent of those triangles could possibly contribute to the final image. By increasing the display window's resolution, more triangles can contribute to the rendered image, thereby adding further detail and making the virtual object appear more realistic.

By today's standards, most displays (and graphics adapters) can support resolutions up to 1600 × 1200 pixels (1.9 million). Although this provides good image quality, additional pixel counts can provide the viewer with more detail and better overall context of what they're seeing. Thanks to recent advances in liquid crystal display technology, IBM has developed its T221 display, which consists of more than 9.2 million pixels (3840 × 2400) in a 22.2-inch (diagonal) viewing image area (more than 200 dpi). With this resolution and clarity, rendered images of virtual objects come to life and become harder to distinguish from the real thing. We can scale video walls, which enlist an array of projectors to generate a display mosaic, to arbitrary resolutions. Driving such high-content displays at interactive rates, however, presents challenges. For example, no single graphics adapter yet has the necessary horsepower and bandwidth to feed a 9.2-million pixel display (at 41 Hz using 24 bits per pixel).

We've addressed many of these challenges by designing the Deep View visualization system (see Figure 1). Deep View consists of a Linux cluster that performs computations to produce 3D geometry, renders the geometry to produce 2D pixels, and then transfers the pixels to be displayed on the T221 display (or video wall). We accelerate the pixel transfer operations using IBM's Scalable Graphics Engine (SGE).

System hardware

We designed Deep View to leverage commodity components wherever possible. Our Linux cluster consists of eight workstations running Red Hat 7.1, each with two 866-MHz processors, 1 Gbyte of RAM, and a mid-range graphics adapter. In addition, our cluster nodes have two commodity interconnects: Myrinet for internode communications and Gigabit Ethernet for transfer of the rendered pixels to the display.

1 Overview of Deep View. The full rack contains eight cluster nodes and two network switches. The half rack to the right is the Scalable Graphics Engine (SGE), connected to the cluster by eight Gigabit Ethernet links. The T221 display sits directly to the left of the SGE.

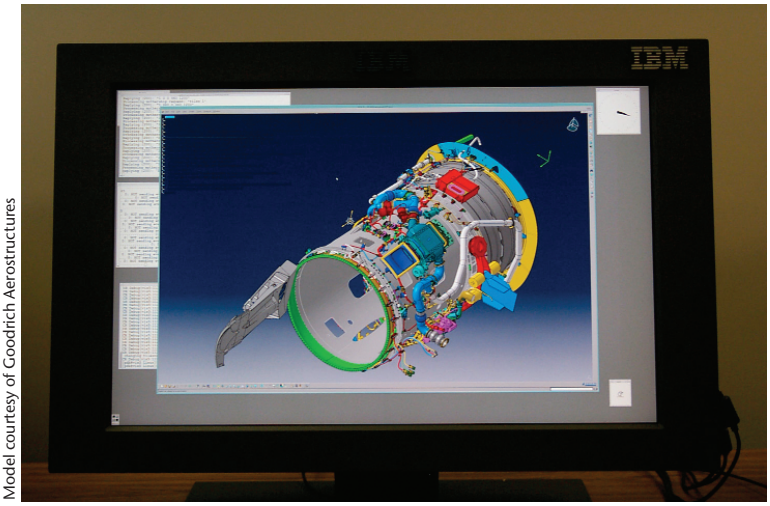
We drive high-resolution displays at interactive frame rates using our cluster and the SGE. The SGE is a network-attached frame buffer capable of double buffering up to 16-million pixels. It routes incoming pixels from multiple sources to the appropriate locations in its frame buffer and then transfers the composited result to the T221 display using digital video interface (DVI) output. In total, the SGE can accept up to 16 input links and can drive as many as eight synchronized DVI outputs. In addition, it can time interleave image pairs from its frame buffer to effect time-division stereo. In the current Deep

0272-1716/02/$17.00 © 2002 IEEE

View configuration, the rendered pixels are sent by each node in the cluster to the SGE over a Gigabit Ethernet link, and we use four of the synchronized DVI outputs to drive the T221 at full resolution.

System software

Another key component of Deep View is the underlying software that leverages the hardware. We've written several in-house applications to visualize the data from various application domains and participated in several open-source software projects. Two of the main pieces of software we're using on Deep View are Chromium and OpenDX-MPI.



Model courtesy of Goodrich Aerostructures

2 Dassault Systems' CAD/CAM design package CATIA running on the T221 display. As engineers design models, in this case a jet-engine nacelle, having the high-resolution display provides detailed information and context that would otherwise be unavailable.

Chromium

Chromium (http://www.sourceforge.net/projects/chromium) is a flexible system that uses modules, called stream processing units (SPUs), to manipulate the stream of OpenGL calls as desired by the user. For example, researchers have implemented sort-first and sort-last parallel rendering algorithms using Chromium's SPUs. Initiated at Stanford University, and now an open-source software project, Chromium's development has been supported by the Lawrence Livermore, Sandia, and Los Alamos National Labs and many other members of the cluster rendering community including IBM Research.

Chromium lets us manipulate streams of OpenGL graphics calls on clusters. More specifically, Chromium intercepts the OpenGL calls by replacing the native OpenGL driver with its own. By doing so, Chromium enables an unmodified application to run on a node (or nodes) in the cluster and have its graphics output automatically rendered in parallel on all the cluster's nodes. This capability allows efficient scaling of the graphics output of an application to drive high-resolution displays such as the T221 or a tiling of standard displays.

One side effect of intercepting graphics calls is that the graphics output is consequently divorced from the application's user interface. Thankfully, due to Chromium's flexibility, we've written an SPU to reintegrate the scaled graphics back into the user interface to display them on the T221 display (see Figure 2). This lets the end user continue using the application exactly as before, with the added benefit of high-resolution visualization.

OpenDX-MPI

Visualizing scientific and engineering data often requires much computation to convert abstract data to meaningful geometry before rendering begins. When we run large simulations on distributed systems, it becomes impractical to off-load the visualization task onto a special-purpose server when doing so requires the serialization and exportation of the raw data to that server. We've developed a distributed-parallel visualization system based on OpenDX, a visualization package developed at IBM Research and released as an open-source package in 1998.

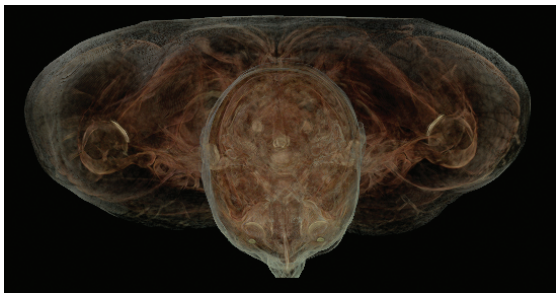
OpenDX-MPI lets us preserve the distribution of data in a parallel simulation through the visualization process, letting us closely couple the visualization of the data to its generation. Thus, the visualization process can take place in situ on the nodes of the distributed system on which the simulation takes place or by using the system's high-performance intercommunications to transfer the data in parallel to dedicated visualization nodes within the same distributed system. The parallel visualization operations that OpenDX-MPI performs result in the distribution of renderable geometry across the visualization nodes. OpenDX-MPI then takes advantage of the distributed rendering capabilities of Deep View to provide high-resolution interactive imagery.

Applications

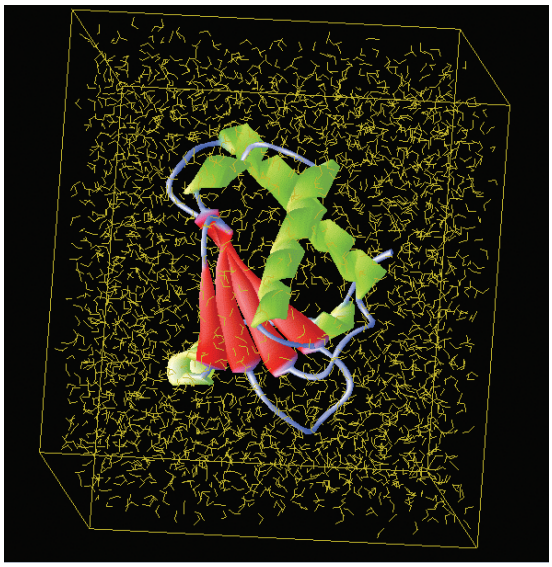
The true test of our worth is how well we support visual-intensive applications. Many application domains would benefit from increased pixel resolutions as a result of high-resolution displays. We've applied our system to applications in the mechanical CAD, medical, molecular modeling, and entertainment fields.

Mechanical CAD

In CAD, designers are constantly challenged to perceive in detail individual components of the model they're designing while observing interrelationships visible only when seeing the entire model. On typical displays, designers must switch back and forth between a whole assembly view and views of individual components, a solution that is tolerable at best. For example, consider the CATIA model in Figure 2. Approximately 1 million triangles are necessary to render the model, which means that on a typical 1-million pixel display, there would be, on average, no more than one pixel available per triangle. To get a detailed display of a portion of the model, so that a triangle spreads over $n \times n$ pixels, a designer must zoom in



3 The Visible Male. We produced this image using parallel volume rendering on Deep View. The high-resolution T221 lets us generate such views without downsampling the original images.



4 A virtual protein (1A6F). Using OpenDX-MPI on Deep View, we can perform the visualization of time steps of a molecular dynamics simulation in parallel and concurrently with the simulation.

5 Using the Chromium software, we can play Quake III Arena at a resolution of 3840 × 2400 pixels.

by a factor of n. In contrast, the T221, being a 9.2-million pixel display, can devote 3×3 pixels to each triangle when showing the entire model, thereby greatly reducing the zoom range required.

An excellent illustration of where this resolution benefits the engineer is when visualizing the model's underlying finite elements. This information, which is often shown using line segments drawn on top of the model's surface, can be lost or clouded when viewed on standard resolution displays. By zooming closely into the model, we could get that information back, but we would then lose the overall context of the model. By using the T221 display, we greatly reduce this problem.

Medical

High resolution is especially beneficial when visualizing detailed medical data. The ability to adequately view and interact with these complex data sets can provide a wealth of information. We illustrate this capability by visualizing high-resolution (3800×2000 pixels) digital images from the National Library of Medicine's Visible Male data set.[1] These images depict volumes of the original data set that we rendered, in parallel, on Deep View. We can then display the resulting images on the T221 display at their intended resolution without loss of detail (see Figure 3).

Interactive and animated visualization also provide opportunities to see internal structures that we wouldn't generally see by solely looking at static images. Using IBM's General Parallel File System (GPFS), we can read each of the images directly from disk in parallel. In addition, GPFS lets us store the entire data set (more than 1,800 images, each 22.8 Mbytes) by striping its 41 Gbytes of data across Deep View's nodes. Using an in-house viewer, we are able to display the images at more than 5 frames per second (fps). When prefetching the images and storing them in the cluster nodes' main memory, we can transfer the pixels through the SGE to the T221 display at 20 fps.

Molecular modeling

We've implemented an extension to OpenDX-MPI that lets it perform real-time concurrent visualization of the progress of distributed-parallel simulations. We've used this extension to provide interactive visualization of the secondary structures of proteins[2] during the course of a run of GROMACS (Groningen Machine for Chemical Solutions), an open-source molecular dynamics simulation package (http://www.gromacs.org), as Figure 4 shows. In this application domain, both the simulation and visualization run concurrently on Deep View, and the high-resolution displays provide ample screen space to visualize the complex molecular structures.

Entertainment

Due to game consoles such as Sony's PlayStation 2, Microsoft's X-Box, and Nintendo's GameCube, computer games are generating more revenue than the entire movie industry. Although high resolution isn't critical to play and enjoy most games, it certainly adds to the user's overall experience. Multiplayer games are especially well suited for high-resolution technology, thereby letting all the players see and experience the same environment.

Using Chromium, we can play Quake III Arena, using only the binary executable we downloaded off the Internet (see Figure 5). Of course, because its developers didn't originally intend resolutions this high when they wrote the game, certain features (such as texture mapping) don't look as good as they could. This will inevitably change as high-resolution displays continue to decrease in price.

Conclusion

From its inception, virtual reality has tested the limits of computing, graphics and displays. Technological progress and commoditization continues to produce remarkable increases in power and decreases in the cost of these elements. Systems such as Deep View aggregate these components with commodity interconnects, creating more powerful systems at modest cost. This progression should mean that over the next few years, the cost and performance of the visual part of VR systems will cease to be limiting factors. At this watershed, we may be too immersed in our work to notice! ■

References

1. V. Spitzer et al., "The Visible Human Male: A Technical Report," *J. Am. Medical Informatics Assoc.*, vol. 3, no. 2, 1996, pp. 118-130.
2. R.E. Gillilan and F. Wood, "Visualization, Virtual Reality, and Animation within the Data Flow Model of Computing," *Computer Graphics*, vol. 29, no. 2, 1995, pp. 55-58.

Readers may contact Thomas Jackman, at the IBM T.J. Watson Research Center, 19 Skyline Dr., Hawthorne, NY 10532, email tjackman@us.ibm.com.

Readers may contact the department editors by email at rosenblu@ait.nrl.navy.mil or Michael_Macedonia@stricom.army.mil.

Early Experiences and Challenges in Building and Using A Scalable Display Wall System

Kai Li, Han Chen, Yuqun Chen, Douglas W. Clark, Perry Cook, Stefanos Damianakis, Georg Essl, Adam Finkelstein, Thomas Funkhouser, Timothy Housel, Allison Klein, Zhiyan Liu, Emil Praun, Rudrajit Samanta, Ben Shedd, Jaswinder Pal Singh, George Tzanetakis, Jiannan Zheng

Department of Computer Science,
Princeton University, Princeton, NJ 08544
http://www.cs.princeton.edu/omnimedia/

Introduction

The Princeton Scalable Display Wall project explores how to build and use a large-format display with multiple commodity components. Our goal is to construct a collaborative space that fully utilizes a large-format display, immersive sound, and natural user interfaces.

Unlike most display wall systems today, which are built with high-end graphics machines and high-end projectors, our prototype system is built with low-cost commodity components: a cluster of PCs, PC graphics accelerators, consumer video and sound equipment, and portable presentation projectors. The advantages of this approach are low cost and technology tracking, as high-volume commodity components typically have better price/performance ratios and improve at faster rates than special-purpose hardware. The challenge is to use commodity components to construct a high-quality collaborative environment that delivers display, rendering, input, and sound performance competitive with, or better than, that delivered by the custom-designed, high-end graphics machine approach.

A schematic representation of our current display wall system is shown in Figure 1. It comprises an 8'×18' rear projection screen with a 4×2 array of Proxima LCD polysilicon projectors, each driven by a 450 Mhz Pentium II PC with an off-the-shelf graphics accelerator. The resolution of the resulting image is 4,096×1,536. The display system is integrated with several components, including: a *sound server*, a PC that uses two 8-channel sound cards to drive 16 speakers placed around the area in front of the wall; an *input cluster*, which uses two 300 Mhz Pentium II PCs to capture video images from an array of video cameras, to gather input from a gyroscope mouse, and to receive audio input from a microphone; a *storage server*, which uses two PCs each with 5 inexpensive EIDE disks; a *local compute cluster* of 4 PCs, which provides high-bandwidth access to compute cycles; a *remote compute cluster* containing 32 PCs; and, a *console PC*, which controls execution of the system.

All the PCs are connected by a 100 Base-T Ethernet network. In addition, the PCs of the display cluster, local compute cluster, and storage server are connected by a Myrinet system area network. We are using the *Virtual Memory-Mapped Communication* (VMMC) mechanism developed in the *Scalable High-performance Really Inexpensive MultiProcessor* (SHRIMP) project.[3] VMMC implements a protected, reliable, user-level communication protocol. Its end-to-end end-to-end latency at the user level is about 13 microseconds and its peak user-level bandwidth is about 100 Mbytes/sec on the Myrinet.[7,4]

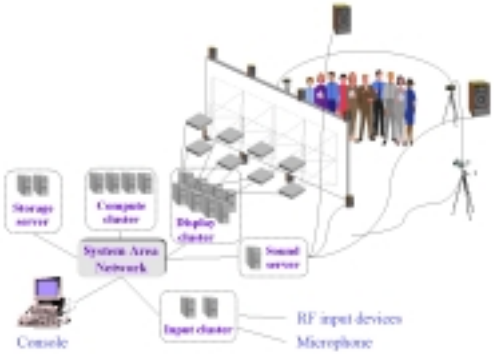


Figure 1: A Scalable, Inexpensive Display Wall System

The foci of our research are usability and scalability. In order to address usability: we must investigate new user interfaces, new content design methodologies, and learn from human perception studies in teaching design courses. In order to achieve scalability, we must carefully address three key system design issues:

- *Coordination among multiple components:* Commodity components are usually designed for individual use rather than as building blocks for a larger, seamless system. To achieve seamless imaging and sound, one must develop methods to coordinate multiple components effectively.
- *Communication performance and requirements:* Immersive and collaborative applications require that multiple components communicate effectively. A scalable system should provide a low-latency, high-bandwidth mechanism to deliver high-performance communication among multiple commodity components. At the same time, software systems and applications must be carefully designed to achieve high quality and performance while minimizing communication requirements.
- *Resource allocation:* Effective resource allocation and partitioning of work among components is critical at both the system and application levels.

In the following sections, we report our early experiences in building and using a display wall system, and we describe our approach to research challenges in several specific research areas, including seamless tiling, parallel rendering, parallel data visualization, parallel MPEG decod-

ing, layered multi-resolution video input, multi-channel immersive sound, user interfaces, application tools, and content-creation.

Seamless Tiling

Image Blending: Although a lot of progress has been made recently in the development of new display technologies such as *Organic Light Emitting Diodes* (OLEDs), the current economical approach to making a large-format, high-resolution display is to use an array of projectors. In this case, an important issue is the coordination of multiple commodity projectors to achieve seamless edge blending and precise alignment.

Seamless edge blending can remove the visible discontinuities between adjacent projectors. Edge blending techniques overlap the edges of projected, tiled images and blend the overlapped pixels to smooth the luminance and chromaticity transition from one image to another. The current state-of-the-art technique is to use specially designed hardware to modulate the video signals that correspond to the overlapped region.[11,18] This electrical edge-blending approach works only with CRT projectors but does not work well with commodity LCD or DLP projectors. This is because these new projectors leak light when projected pixels are black, making them appear dark gray. Overlapped dark gray regions are then lighter gray – brighter than non-overlapped regions. In order to avoid seams we reduce the light projected in the overlapped regions.

Our approach is based on the technique of *aperture modulation*, that is, placing an opaque object in front of a lens (between the projector lens and the screen) to reduce the luminance of the image without distorting the image itself. Thus, by carefully placing an opaque rectangular frame, we can make its shadow penumbra coincide with the inter-projector overlap regions.[8]

Computational Alignment: To make a multi-projector display appear seamless, projected images must have precise alignment with each other in all directions. Aligning projectors manually is a time-consuming task. The traditional alignment method is to use a sophisticated adjustable platform to fine-tune projector position and orientation. This approach requires expensive mechanical devices and extensive human time. In addition, it does not work for commodity projectors whose lenses tend to produce image distortions.

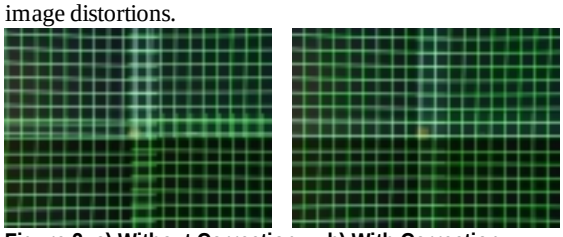


Figure 2: a) Without Correction b) With Correction

To overcome both misalignment and image distortion problems, we use image-processing techniques to "correct" the source image before it is displayed by misaligned projectors. In other words, we pre-warp the image in such a way that the projected images are aligned. We call this approach *computational alignment*. It requires only the coarsest physical alignment of the projectors. Our alignment algorithm currently calculates for each projector a 3×3 projection matrix, with which an image warping process resamples the images to counter the effects of physical misalignment. Figure 2a shows a picture without correction. Figure 2b shows the picture after each projector re-samples the image according to its correct perspective matrix. As a work in progress, we adapt our alignment algorithm to correct some distortions caused by imperfect lenses, e.g. radial distortions.

We obtain precise alignment (or misalignment) information with an off-the-shelf camera that has much lower resolution than our display wall. We zoom the camera to focus on a relatively small region of the display wall, and pan the camera across the wall to get a broader coverage. The camera measures point correspondences and line matches between neighboring projectors. We then use simulated annealing to minimize alignment error globally, and solve for the projection matrices. Our approach differs from the solutions of Rasker, *et al.*,[14] which uses carefully calibrated, fixed-zoomed camera(s) to obtain projector distortion measurements. The cameras in their approach have to see the entire screen or a significant portion of it; and, therefore, cannot easily obtain sub-pixel alignment information.

Parallel rendering

We are investigating parallel rendering algorithms[6] for real-time display of very large, high-resolution images partitioned over multiple projectors. Here we face all three general types of research challenges: coordination of PCs and graphics accelerators to create consistent, real-time images, communication among multiple PCs and their graphics accelerators, and resource allocation to achieve good utilization.

The focus of our efforts is on developing "sort-first" and "sort-last" parallel rendering methods that minimize communication requirements and balance the rendering load across a cluster of PCs.[12] Our general approach is to partition each frame into a number of "virtual tiles." Each rendering machine is then assigned a set of virtual tiles so that the load is as evenly balanced as possible. Since the virtual tiles usually do not correspond to the physical tiles on the wall, rendered pixels must often be read back from the rendering PC's frame buffer and transferred over the network to the projecting PC's frame buffer. We use the VMMC mechanism to achieve low latency and high bandwidth communication for the pixel redistribution phase, as well as to provide fast synchronization of the frame buffer swapping.

The research issues are to develop algorithms that compute the shapes and arrangement of virtual tiles dynamically, sort graphics primitives among virtual tiles in real-time, deliver graphics primitives to multiple PCs in parallel, and redistribute pixels across a network efficiently. To explore this space we have designed and implemented several "sort-first" virtual tiling algorithms. The best of these algorithms uses a KD-tree partition of the screen space followed by an optimization step to ensure the best possible balance of the load.[15] Figure 3 and Figure 4 show the cases with a static screen-space partition without load balancing and a KD-tree partition after load balancing, respectively. The colors indicate which machines render the different parts of the scene. The imbalance in the first case can be observed by looking at the "load bars" on the bottom right of the figure. The load is much better balanced in the KD-

tree case, and as a result the final frame-time is up to four times lower with eight PCs.

Figure 3: Parallel Rendering without Loading Balancing

Figure 4: Parallel Rendering with Load Balancing

Parallel Data Visualization

Increases in computing power have enabled researchers in areas ranging from astrophysics to zoology to amass vast data sets resulting from both observation and simulation. Since the data itself is quite rich, the display wall presents an ideal medium for scientific visualization at high resolution. The magnitude of the data sets motivates the use of parallel computation, a fast network, and separation of computation and rendering across different machines.

The initial focus of our research is to develop parallel algorithms that permit the users to interactively view isosurfaces in volumetric data on the display wall. Our system uses the PCs in the display cluster to perform rendering, the PCs in compute cluster to perform isosurface extraction, and storage servers to hold datasets. We coordinate these three sets of PCs in a pipelined fashion on a per frame basis. Data are sent from the storage servers to the isosurface extraction PC cluster. Triangles for an isosurface are generated in parallel using a *marching cubes* algorithm[10] accelerated with an interval method[5] based on Chazelle's filtering search. They are then sent to the appropriate rendering PCs.

We have experimented with lossless compression methods to reduce communication requirements. Even with compression, we find that low-latency, high-bandwidth communication between the isosurface extraction PCs and rendering PCs is critical.

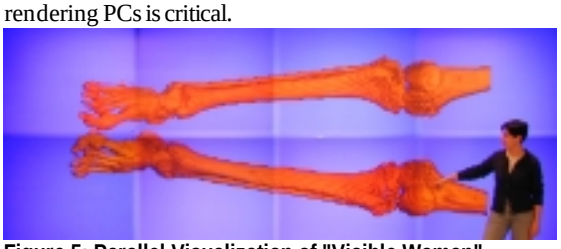


Figure 5: Parallel Visualization of "Visible Woman"

Figure 5 shows the result of using our parallel visualization system to visualize part of the Visible Woman data set.[13] We are currently focusing on better isosurface extraction algorithms, large-scale storage server development, and load-balancing methods to improve the utilization of computing resources.

Parallel MPEG-2 Decoding

MPEG-2 is the current standard format for delivering high-quality video streams for entertainment, collaboration and digital art. Our goal is to develop fast pure-software MPEG-2 decoding methods on a PC cluster to bring HDTV or even higher resolution MPEG-2 videos to a scalable display wall. To achieve the 60 fps real time frame rate including the overhead in scaling and loading pixels into the frame buffer, a decoder should be capable of decompressing one frame in less than about 14 ms. We approach the problem in two steps: developing a fast decoder on a single PC and designing a fast parallel, scalable decoder for a PC cluster. The key research challenges here are coordination among PCs to split an MPEG-2 stream and fast communication among PCs to decode high-resolution streams in real time.

To improve the MPEG-2 decoding performance on a single PC, we exploited both instruction level parallelism and memory/cache behavior. We develop our decoder based on the open source *MPEG Software Simulation Group* reference design, which decodes 720p HDTV (1280×720) at about 13 fps on a 733 MHz Pentium III PC. We extensively use Intel MMX/SSE instructions to accelerate arithmetic operations and carefully design the data structures and their layouts to improve the data cache utilization. Our preliminary result is a decoder capable of decompressing 720p HDTV stream at over 56 fps on a 733 MHz Pentium III PC. The speed-up is over a factor of four.

To further improve the performance, we use parallel decoding on a PC cluster. Previous work on parallel MPEG decoding is done almost exclusively on shared memory multiprocessors.[2] They parallelize MPEG-2 video decoder at either the *picture* or *slice* level. However, the amount of data movement among the PCs is too high if these methods are used for a PC cluster. We develop a novel *macroblock* level parallelization. We use a single PC to split an MPEG-2 stream into multiple sub-streams at macroblock level and send them to the PCs in the display cluster to be decoded, scaled and displayed.

With the previous picture-level or slice-level parallelization, the per-link bandwidth requirement of the decoding PC depends on the whole video size. With our macroblock-level parallelization, it depends only on the size of the portion that the local node is decoding. This makes our approach highly scalable. Our preliminary result shows that with 4 PCs (in a 2×2 setup) decoding 720p HDTV streams in parallel, the aggregate communication bandwidth requirement among all nodes is only about 100 Mbits/sec. As a comparison, this number can be as high as 1.7 Gbits/sec when a picture or slice level parallelization is used.

Multi-layered Video Input

Video resolution has always been limited by the TV standards. In order to take advantage of the high resolution of a scalable display wall, we are working on methods to create video streams at a scalable resolution that matches the display resolution, using a small number of commodity video cameras. The main research challenge is the coordination among video cameras.

The traditional approach is to use juxtaposed cameras with edge overlapping and stitch multiple images together.[17] It has several disadvantages. First, juxtaposed cam-

eras make zooming awkward – the cameras must be synchronized and the angles between them must be adjusted mechanically at the same time. Second, since each camera has its own focal point, scenes with a lot of depths can look unnatural with multiple focal points. Third, it requires many video cameras. For example, it requires 28 640 × 480 video cameras for a 4,096 × 1,536 resolution display wall. The aggregate communication requirement of the video streams is also too high for the network. We would like to overcome all of these problems.

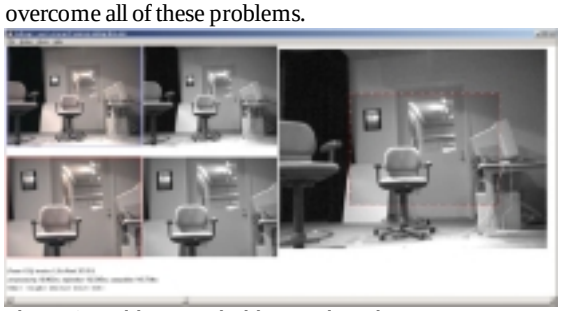


Figure 6: Multi-Layered Video Registration Program

Our approach is called layered multi-resolution video. We use a number of cameras to cover different fields of view. Each camera can be panned, tilted, and zoomed individually. We are developing a fast registration algorithm to find the correspondence of the different layers and merge them into one. This method not only solves the three problems above, but also fits nicely into MPEG-4 video compression framework. Our current registration algorithm runs at 30 registration-passes per second for 2 images. Figure 6 demonstrates the registration process. Our goal is to develop a registration algorithm that runs at real time.

Multi-channel Immersive Sound

Sound guides the eyes, enhances the sense of reality, and provides extra channels of communication. Since the visual display is spread over a large surface, large amounts of the displayed data might be out of the visual field of any user. Sound can be used to draw directional auditory attention to an event, causing users of a large display system to turn their heads toward the sound and thus bringing important visual information into their field of view. To investigate the integration of immersive sound with a large-scale display wall, we use a large number of speakers positioned around the space in front of the display wall to provide immersive sound synthesis and processing in real time. The key challenge is the coordination of multiple sound devices to create immersive sound.

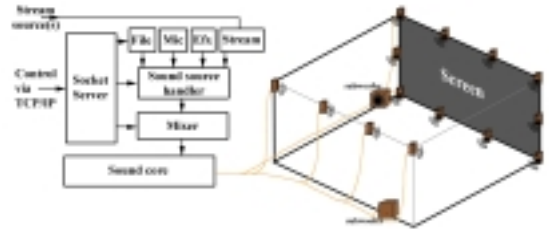


Figure 7: Multi-Channel Sound System

The display wall sound system is implemented on commodity PCs, using inexpensive multi-channel sound cards. These cards are designed for digital home-recording use, and can be synchronized through SPDIF / AESEBU cables and special calls to the software drivers. We have written a sound server that takes commands from any computer via a TCP / IP connection. The server can playback sound files through any combination of the 16 speakers in the present configuration (See Figure 7). Other possible sound sources include onboard synthesis of sound effects, microphone signals, sound streams from any machine on the network or web, and effects (reverb, echo, etc.) processing of any sound source.

User Interfaces

A large collaborative space presents interesting challenges for user interfaces, both display and control. Because of the scale of the display wall, it is important to track human positions, recognize gestures, and construct imagery and sound appropriate for the user's position. Many methods developed in the past require users to carry cumbersome tracking or sensing devices. Our focus has been on developing natural methods for users to interact with the system. We use multiple cameras in the viewing field to track human and input device. We also develop image processing algorithms to understand gestures in a collaborative environment. The main research challenge is the coordination of among commodity input devices and with the computers in the display wall PC cluster.

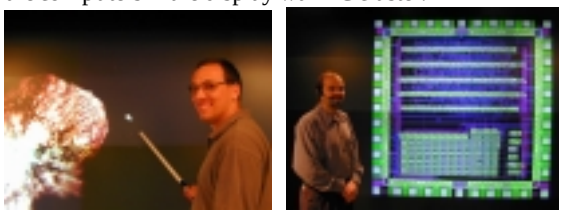


Figure 8: a) Magic Wand Input b) Voice Recognition

We write a multi-input mouse server program that runs on a master cursor control computer. Any other computer can take control of the display wall mouse by running a mouse client program and connecting to the server. This has allowed us to quickly construct and test a number of new pointing devices, including a swivel chair (the Quake Chair), voice input mouse control, and pressure sensitive floor panels. Figure 8a and Figure 8b shows the use of a camera-tracked wand as a pointer device and a wireless microphone as a speech recognition device, respectively. Research challenges include allowing multiple cursors at once, as well as further refinement and integration of camera tracking.

Methods to Design Application Tools

It is important and non-trivial to bring many applications to a scalable display wall and run them at the *intrinsic* resolution supported by the display surface. Most video-wall products use special-purpose hardware to scale relatively lower-resolution content, such as NTSC, VGA, SVGA, HDTV formats to fit large display surfaces. Only a few expensive solutions use high-end graphics machines to render directly in the intrinsic resolution of a multi-projector display system. Coordination and communication are the two main challenges in developing tools to port off-the-shelf applications to a scalable display wall using its native display resolution.

We have investigated four methods to design tools for applications: custom-designed, distributed application, distributed 2D primitive, and distributed 3D primitive. The

following subsections illustrate each method by an example.

Figure 9: Looking at Image with Still Image Viewer

Custom-Designed Method: Our first tool on the display wall is a *Still Image Viewer*, which allows a naive user to display still images and perform cross fading between images on the wall. The image viewer contains two parts: a controller program and an image viewer program. An image viewer program runs on every PC in the display cluster. The controller program runs on a different PC and sends commands over the network, such as loading an image from the disk, displaying a cached image, or cross fading between two cached images. The image viewer loads JPEG images from a shared file system, decodes only its portion of the image, and synchronizes with other viewers on other PCs prior to swapping the frame buffer. The controller program also implements a scripting interface so that the users can write scripts to control image and video playback that are synchronized with our multi-channel sound playback. Many students have made multimedia presentations on our display wall using the image viewer and the multi-channel sound system. Figure 9 shows an image of the International Space Station on the display wall.

Figure 10: A Distributed Building Walkthrough Program

Distributed Application Method: We distribute application-level input commands to bring a *Building Walkthrough* system designed for a uniprocessor system to the display wall. We run an instance of the building walkthrough program on every PC in the display cluster. In order to coordinate among these programs, we run another instance on the console PC. A user drives the walkthrough using the console PC. The console translates the user inputs and sends the camera information and screen space information to each PC that drives a tile of the display wall. PCs in the display cluster execute copies of a uniprocessor walkthrough application, each of which renders a different part of the screen from its own copy of the scene database. They synchronize frame updates with network messages under the control of console. This method provides interactive frame rates (e.g. 20 fps) without noticeable synchronization problems. Figure 10 shows the walkthrough program being run with a 3D model created by Lucent Technologies.

Figure 11: Windows 2000 Virtual Display Driver

Distributed 2D Primitive Method: We have developed a *Virtual Display Driver* (VDD) to bring existing Windows applications to the display wall, using a distributed 2D primitive method. VDD is a Windows display driver that "fakes" a high-resolution graphics adapter to the Windows 2000 operating system. It leverages the feature in Windows 2000 that supports multiple monitors on a single PC. VDD intercepts all *Device Driver Interface* (DDI) calls and execute them remotely as remote procedure calls on the PCs in the display cluster. The users can drag application windows from the regular CRT display into our virtual display, the contents of which are subsequently drawn on the display wall. All drawing done by the application on VDD is performed in the intrinsic resolution of the virtual display, which is the same as the display wall. Therefore, users can see a lot more details in any Windows applications than with existing commercial video-walls. Figure 11 shows Microsoft PowerPoint and Internet Explorer running on our Display Wall through VDD. At the close range where people are standing in front of the display wall, both applications show adequate details and no fuzziness with line drawings and text.

Figure 12: GlQuake Running on the Display Wall

Distributed 3D Primitive Method: We developed a user-level, *Distributed OpenGL* tool using a 3D primitive distribution method. Unlike the distributed 2D primitive method where our tool works at the device driver level, the distributed OpenGL library lives at the user level. We take advantage of the fact that on all Windows platforms, an application's calls to the OpenGL API are made through a *Dynamically Linked Library* (DLL), opengl32.dll. Our approach is to implement our own opengl32.dll to intercept all the OpenGL calls, and forward them to the PCs in the display cluster. These PCs receive the RPC calls and directly execute them, with the exception that the view frustums are properly modified so that each projector renders only its own tile portion of the screen space. This distributed OpenGL mechanism allows many off-the-shelf Windows OpenGL applications to run on the display wall without any modifications. We have brought up many such applications including games, CAD and visualization tools.

Figure 12 shows the game GlQuake being run on the display wall using our distributed OpenGL mechanism. Currently, we are investigating methods for integrating our parallel rendering algorithms into this OpenGL library.

Content Creation and Design Implications

We started studying content creation and design methods at the same time as other research topics. We taught two design courses using the display wall. The main point of these courses is to provide opportunity and experience utilizing desktop-size screens to create effective wall-size images. Figure 13 to 15 show students' creations on the display wall.

Figure 13: Multiple Small Windows © IAN BUCK

Compared to the traditional, expensive display walls, the inexpensive aspect of the scalable display wall makes a big difference in content creation. Suddenly, we are presented with a new design space available to all users, in particular non-technical users. This rapid democratization of billboard-size display space is quite provocative. Students in the design class are asked to imagine future applications and implications when many such walls are widely in use, and to investigate the best uses for these large displays.

Figure 14: Sketches on a Digital Canvas © JON HARRIS

One implication of a wall-size image is that it completely fills our visual field, which creates a one-to-one experience with the onscreen imagery. There is no border or frame for scale reference as on small monitors. This single shift creates a whole new design paradigm.[16] Areas of interest and focus must be added into the image composition. A second implication is that a group can interact with information on just a portion of the screen while others focus on a different area. Different viewers can be at different distances from the high-resolution screen and move around in the room space while viewing. Third, objects can be seen life-size or intensely magnified. For example, an image of a dense computer chip reads like a road map. Fourth, there is not necessarily a need to rapidly change the images, as they can be so densely filled with data that it takes a while to absorb it all. Often, a single high-resolution screen can be displayed for 10 to 20 minutes and remain continuously interesting. Fifth, the light from the screen can become the room light for the working group. All of these elements, especially the frameless nature of the image, require new thinking and new ways of approaching design.[1,19]

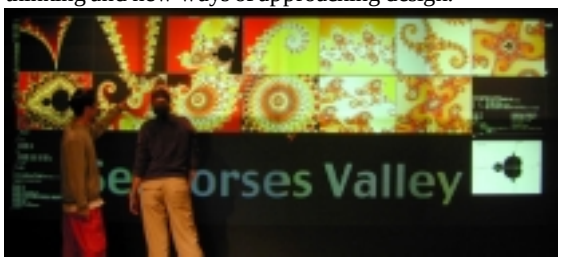


Figure 15: A Fractal Image © WILMOT KIDD

This new design paradigm motivates future work in composition tools for large-format displays. Self-expression has a new form. TCL scripting adds the dimension of time to wall presentations, providing capabilities for timed displays and dissolves from image to image. By synchronizing music and sounds to changing images, the wall has become a storytelling space for presentations of 5 to 10 minutes, as complex and engrossing as any short film or video. The wall room, with its billboard size images, has been used three times as a performance art and theater space. Virtually everyone who visits the display wall expresses some kind of emotional response about being in the huge visual and aural space.[9]

Summary and Conclusions

The Princeton Scalable Display Wall prototype system has been operational since March 1998. It has been used as an infrastructure to conduct our research as well as to teach two design courses.

The approach of using a multiplicity of commodity parts to construct a scalable display wall system works well, but it requires us to address design tradeoffs to deal with coordination, communication and resource allocation issues. We have successfully addressed these tradeoffs and developed solutions in several research areas as outlined in this paper. In seamless rendering, we have developed a combination of optical edge-blending and software image manipulation for projector alignment. In parallel rendering, we have developed a "sort-first" screen partitioning method that achieves good load balance and parallel speedup. In parallel data visualization, we have developed a parallel isosurface extraction algorithm for a PC cluster architecture. In parallel MPEG-2 decoding, we have developed a fast splitter and a fast decoder that achieve real-time decoding entirely in software with minimal communication overhead. In layered multi-resolution video, we interactively combine multiple video streams with a fast registration algorithm. And in application tools design, we developed four methods to let existing applications use the native resolution of the display system while minimizing communication requirements.

Study of user interface issues and human perceptions is very important in building a collaborative environment with a scalable display wall system. We have developed and experimented with several user interfaces beyond the traditional keyboard and mouse, including a gyroscope mouse, a "magic wand" implemented by multi-camera

tracking, and a speech recognition user interface. Our experience shows that natural, unencumbered user interfaces based on passive sensors are useful in such an environment and that it is very desirable to allow multiple users to control a shared display wall simultaneously.

Finally, in teaching design courses using our display wall system, we have found that the resolution and scale of the display require new ways of approaching design. For instance, vast amounts of information can be presented in a single image, rather than as a sequence of images as would be required in a desktop display. Typographic layouts where the font sizes can range from 2 to 600 points bring new capabilities to the use and meaning of text. Sound, especially spatial sound integrated with imagery, is critical for storytelling. A design aesthetic is emerging for large scale, high-resolution images that are dependent on the center of the images rather than on the frame of the wall. Perhaps, from the high magnifications seen in wall size imagery, we will discover new insights and experiences that had not previously been available. ■

Acknowledgements

The Princeton Display Wall Project is supported in part by Department of Energy under grant ANI-9906704 and grant DE-FC02-99ER25387, by Intel Research Council and Intel Technology 2000 equipment grant, and by National Science Foundation under grant CDA-9624099 and grant EIA-9975011. The research programs of Adam Finkelstein and Thomas Funkhouser are also supported, respectively, by an NSF CAREER award and an Alfred P. Sloan Fellowship. We are also grateful to Arial Foundation, Interval Research, Microsoft Corporation and Viewsonic for their generous equipment and software donations.

We would like to thank John DiLoreto for building special large-format screens, and several Intel colleagues Konrad Lai, Dick Hofsheier, Steve Hunt, Paul Pierce, and Wen-Hann Wang for sharing their ideas, projector-mount design, and contents. We also would like to thank all students who took the design classes and who conducted independent studies using the display wall for their content creation.

References

1. R. Arnheim. *The Power of the Center*. University of California, Berkeley, CA, 1988.
2. A. Bilas, J. Fritts, and J. P. Singh. "Real-Time Parallel MPEG-2 Decoding in Software." In *Proceedings of International Parallel Processing Symposium*, 1997.
3. M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. "Virtual Memory Mapped Network Interface for the Shrimp Multicomputer." In *ACM/IEEE Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp 142-153, April 1994.
4. Y. Chen, C. Dubnicki, S. Damianakis, A. Bilas, and K. Li. "UTLB: A Mechanism for Translations on Network Interface." In *Proceedings of ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pp 193-204, October 1998.
5. P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. "Speeding Up Isosurface Extraction using Interval Trees." *IEEE Transactions on Visualization and Computer Graphics*, Vol 3(2), pp 158-170, June 1997.
6. T. W. Crockett. "An Introduction to Parallel Rendering." *Parallel Computing*, Vol 23, pp 819-843, 1997.
7. C. Dubnicki, A. Bilas, K. Li and J. Philbin. "Design and Implementation of Virtual Memory-Mapped Communication on Myrinet." In *Proceedings of the IEEE 11th International Parallel Processing Symposium*, April 1997.
8. K. Li and Y. Chen, "Optical Blending for Multi-Projector Display Wall System." In *Proceedings of the 12th Lasers and Electro-Optics Society 1999 Annual Meeting*, November 1999.
9. M. Lombard and T. Ditton. "At the Heart of It All: The Concept of Presence." http://www.ascusc.org/jcmc/vol3/issue2/lombard.html
10. W. Lorensen and H. Cline. "Marching cubes: a high resolution 3D surface construction algorithm." *ACM Computer Graphics (SIGGRAPH '87 Conference Proceedings)*, Vol 21(4), pp 163-170, 1987.
11. T. Mayer. "New Options and Considerations for Creating Enhanced Viewing Experiences." *Computer Graphics*, Vol 31(2), pp 32-34, May 1997.
12. S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering." *IEEE Computer Graphics and Applications*, Vol 14(4), pp 23-32, July 1994.
13. Visible Human Project at the National Library of Medicine. http://www.nlm.nih.gov/research/visible/
14. R. Raskar, M. S. Brown, R. Yang, W.-C. Chen, G. Welch and H. Towles. "Multi-Projector Displays Using Camera-Based Registration." In *Proceedings of IEEE Visualization 1999*. October 1999.
15. R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. "Load Balancing for Multi-Projector Rendering Systems." *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Los Angeles, CA, August 1999.
16. B. Shedd. "Exploding the Frame: Seeking a New Cinematic Language", *SMPTE 135th Conference*, 1994
17. R. Szeliski, and H.-Y. Shum. "Creating Full View Panoramic Image Mosaics and Environment Maps." In *Proceedings of ACM Siggraph 1995*, 1995.
18. http://www.trimension.com/
19. E. R. Tufte. *Visual Explanations.* Graphics Press, Cheshire, CT, 1997

High-Performance Visualization of Large and Complex Scientific Datasets

Out-of-Core Surface Simplification

Peter Lindstrom

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

This work was performed under the auspices of the U.S. Department of Energy by
University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Talk Overview

introduction

spatial clustering methods

surface segmentation methods

wrap-up

Introduction

geometric datasets are growing rapidly in size

- scientific simulations
- medical imaging
- laser range scanning
- remote sensing
- computer-aided design

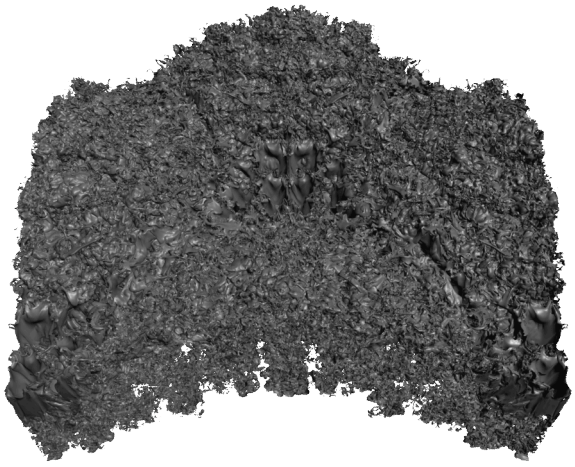
datasets often too large for common tasks

- storage and transmission
- in-core processing and visualization on desktop computers

Large-Data Example #1

Richtmeyer-Meshkov instability simulation at LLNL

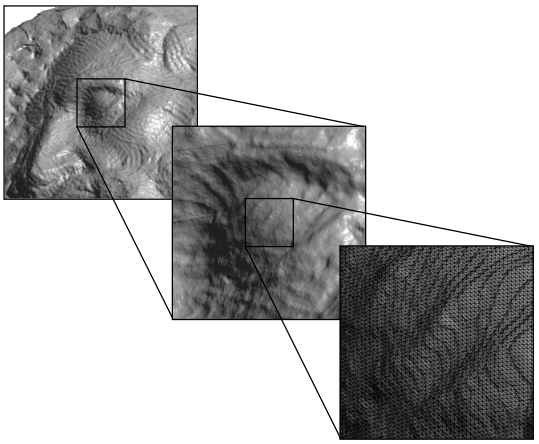
- $2K^3$ voxels, 27,000 time steps, 470 million triangle isosurface



Large-Data Example #2

Digital Michelangelo project

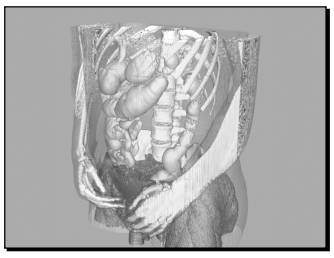
- _ millimeter resolution range scan: 370 million triangles



Large-Data Example #3

Visible Human project

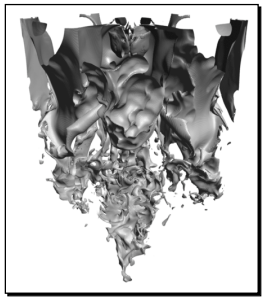
- 0.33 millimeter resolution scan: over 100 billion voxels



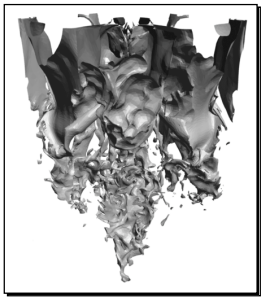
Managing Large Data

one solution to large-data problem: simplification

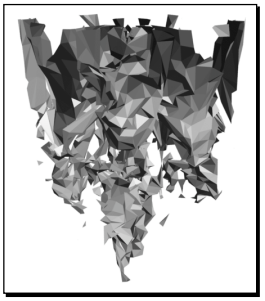
- approximate dataset with fewer primitives while satisfying error tolerance or size constraint
- enables levels of detail, multiresolution representations, progressive/adaptive/view-dependent refinement



2.4M faces



100K faces



5K faces

Triangle Mesh Simplification

most surfaces are represented as triangle meshes

common triangle mesh coarsening operations

- vertex removal (*Schroeder et al. '92*)
- edge collapse (*Hoppe et al. '93*)
- half-edge collapse (*Kobbelt et al. '98*)
- triangle collapse (*Hamann '94*)
- vertex pair contraction (*Garland & Heckbert '97*)
- vertex clustering (*Rossignac & Borrel '93*)

all operations instances of vertex set partitioning

- (except general vertex removal)

Vertex Set Partitioning

partition vertices into disjoint sets (clusters)

- $V = V_1 \cup V_2 \cup \cdots \cup V_n$

collapse each set into single (possibly new) vertex

- $V_i \rightarrow v_i$

discard degenerate triangles

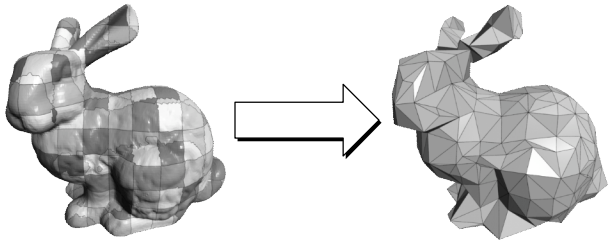
- **"surviving" triangles span three different clusters**

variations between simplification operations

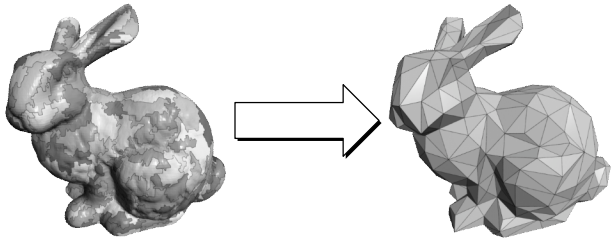
- **legality of partitions (e.g. clusters must be edge-connected)**
- **vertex positioning (e.g. subsampling vs. optimization)**

Examples of Vertex Set Partitions

uniform partitions (spatial clustering on 3D grid)



adaptive partitions (error-driven edge collapse)



The Dilemma

surface simplification can reduce data size, but...

simplification traditionally done in-core

- **load entire surface mesh into memory**
- **queue up sequence of fine-grained coarsening operations (e.g. vertex removal, edge collapse) in order of least error**
- **proceed by iteratively coarsening mesh**
- **memory: $O(n)$ storage, random (i.e. incoherent) access**
- **running time: $O(n \log n)$**
- **conclusion: *does not scale well***

Out-of-Core Simplification

recent research focused on external memory methods

- **key concepts: data segmentation, coherent/sequential access, disk-based algorithms (e.g. B-trees, external sorts)**
- **simplification of arbitrarily large surfaces now possible**

two distinct approaches to out-of-core simplification

- **spatial clustering**
  - ***Lindstrom '00, Lindstrom & Silva '01, Shaffer & Garland '01, Fei et al. '02, Garland & Shaffer '02***
- **surface segmentation**
  - ***Bernardini et al. '99, El-Sana & Chiang '00, Prince '00, Cignoni et al. '02, Choudhury & Watson '02***

Talk Overview

introduction

spatial clustering methods

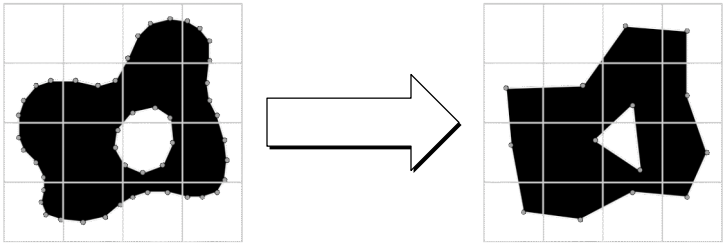
surface segmentation methods

wrap-up

Spatial Clustering

idea: partition space that surface lies in into 3D cells

- vertex set partitions given geometrically by cell containment
  - no topological constraints or considerations
- cell geometry can be adapted to surface
  - e.g. smaller cells in detailed regions, adapt cell shape to surface
- simplest partition: uniform 3D Cartesian grid



OoCS: Uniform Spatial Clustering
Lindstrom '00

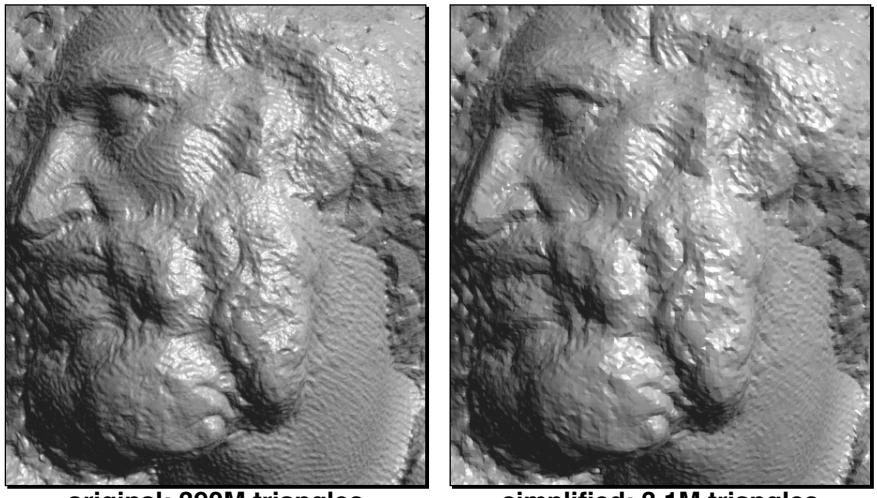
algorithm overview

- partition space using a fixed 3D rectilinear grid
  - sparse data structure (hash table) to represent occupied cells
- position vertices by minimizing *quadric error*
  - weighted sum of squared distances to set of planes
- features
  - fast linear-time, single-pass algorithm
  - reads arbitrarily large "triangle soup" stream sequentially
- drawbacks
  - requires enough core memory to store simplified mesh
  - non-adaptive partitions yield poor quality
- basis for many out-of-core spatial clustering methods

OoCS Algorithm

1. for each triangle t in T_{in}
2. compute quadric matrix $\mathbf{Q}_t = \mathbf{n}_t\mathbf{n}_t^T$
3. for each vertex v of t
4. map v to cluster cell c
5. add $\mathbf{Q}_t$ to $\mathbf{Q}_c$
6. if t not degenerate, add t to T_{out}
7. for each cluster c
8. compute vertex $\mathbf{x}_c$ from $\mathbf{Q}_c$
9. add $\mathbf{x}_c$ to $\mathbf{X}_{out}$

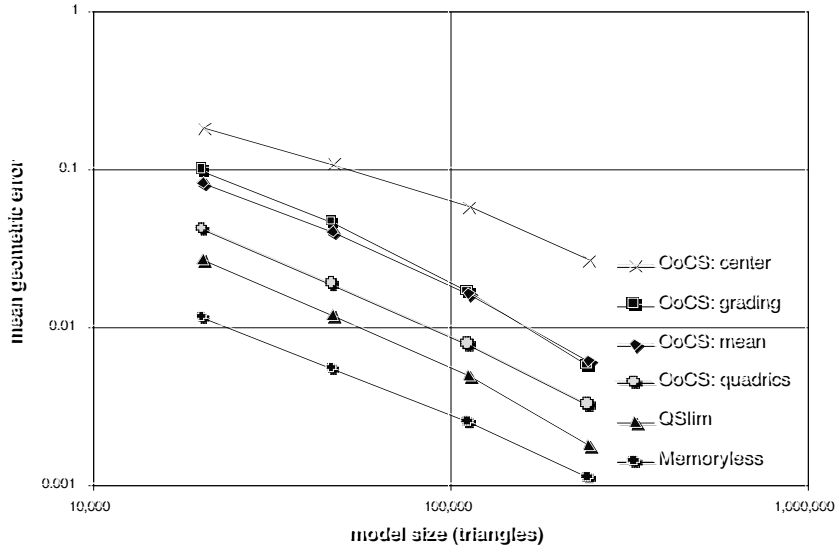
OoCS Results: St. Matthew Statue (close-up view)



original: 390M triangles | simplified: 3.1M triangles

dataset courtesy of the Digital Michelangelo Project

OoCS Results: Mesh Quality



OoCS Results: Resource Usage

simplification speed

- processes 250K triangles per second on desktop PC
  - roughly one hour to simplify billion-triangle mesh

memory requirements

- 63–72 bytes per *output* triangle (depends on hash table load)
  - compare with 80 bytes per *input* triangle for typical in-core method
  - allows 15 million-triangle *simplified* mesh on 1GB RAM computer

Output Sensitive Methods

handling arbitrarily large inputs is nice, but…

output size limited by available RAM

- what if simplified mesh is not accurate enough?
  - cannot use finer grid because of memory constraints

disk cheaper, orders of magnitude larger than RAM

- so can we use disk instead of RAM during simplification?
- yes, but we must avoid slow random accesses!

OoCSx: Memory Insensitive Clustering
Lindstrom & Silva '01

achieve coherent disk access via external sort

- only random access in OoCS is to clusters' quadric information (via in-core hash lookup)
- sort quadric information on disk, then process sequentially

OoCSx Algorithm Overview

simplification

- scan input mesh
  - read triangles, compute plane equations
  - output cluster ID, plane equation for each vertex
  - output non-degenerate triangles to triangle file
- sort plane equation file on cluster ID
- compute cluster quadrics, output optimal vertices

re-indexing

- sort triangle file on given vertex field
- scan and replace cluster ID with vertex ID
- repeat for each of three vertex fields

OoCSx Results

simplification speed

- **30-70K triangles per second (vs. 250K for OoCS)**

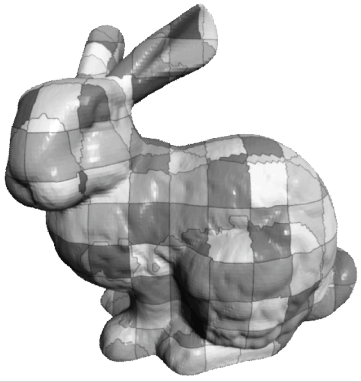
resource usage

- **temporary disk: ~5 times size of input mesh**
  - **depends on external sort used**
- **memory: arbitrarily little (specified by user)**
  - **4 MB used in current implementation**

Drawbacks of Uniform Spatial Clustering

uniform grid not well adapted to surface features

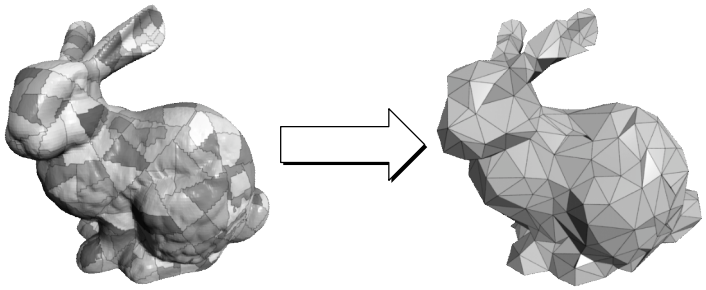
- **spatial proximity does not imply well-shaped clusters**
- **uneven and/or disconnected surface patches frequent**
- **fixed-resolution grid limits size of smallest feature preserved**



Adaptive Spatial Clustering
Shaffer & Garland '01

idea: analyze mesh and repartition space

- accumulate quadrics on uniform grid in first pass (as in OoCS)
- use PCA of primal/dual quadrics to determine better partitions
  - construct BSP-tree top-down for arbitrary convex cells
- recluster mesh on irregular grid in second pass

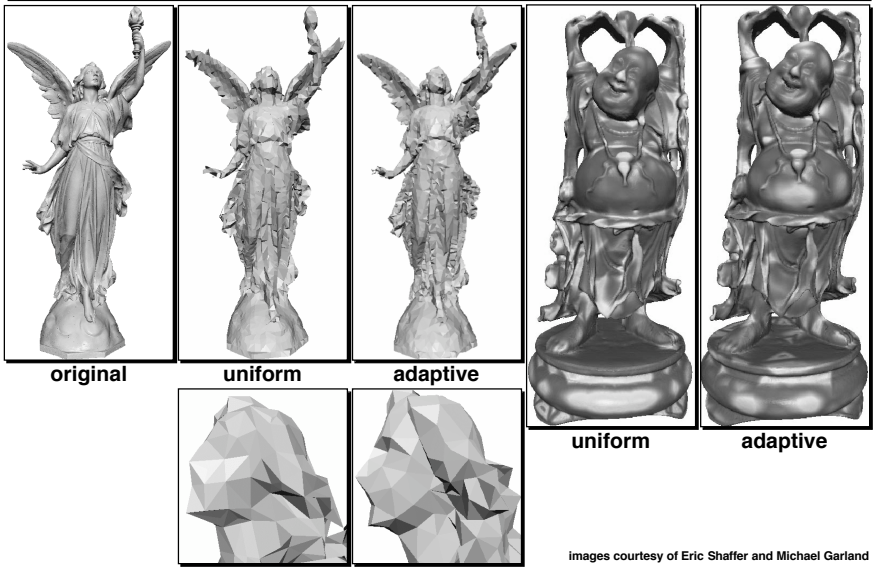


Adaptive Spatial Clustering

algorithm characteristics

- features
  - higher quality, better distribution of triangles than uniform spatial clustering
- drawbacks
  - higher grid resolution (thus more memory) needed in first pass
  - still output sensitive memory requirements

Adaptive Spatial Clustering Results



images courtesy of Eric Shaffer and Michael Garland

Other Spatial Clustering Methods

Fei et al. '02a (SIGGRAPH 2002 tech. sketch)

- simplification sensitive to translation of cluster grid
  - degree of sensitivity (DOS) depends on amount of surface detail
  - compute local DOS by clustering on two interlaced grids
  - use more detail in regions of high DOS

Fei et al. '02b (Computer Graphics Forum 2002)

- following initial uniform clustering pass, use per-cell quadric error to determine which cells to refine/coarsen
  - introduce split planes based on PCA of quadric matrices
  - additional pass(es) used to incrementally, adaptively refine/coarsen mesh
  - more memory efficient but slower adaptation than in *Shaffer & Garland '01*

Talk Overview

introduction

spatial clustering methods

surface segmentation methods

wrap-up

Surface Segmentation

idea: segment surface into small, disjoint patches

- **ensure patches small enough to fit in-core**
- **simplify each patch using in-core method (e.g. edge collapse)**
- **stitch together simplified patches**
- **repeat if necessary**

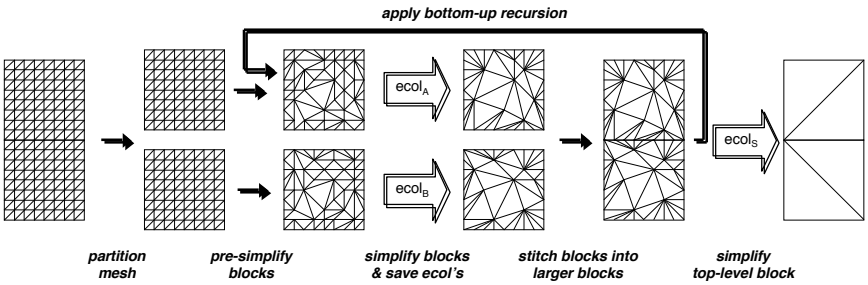
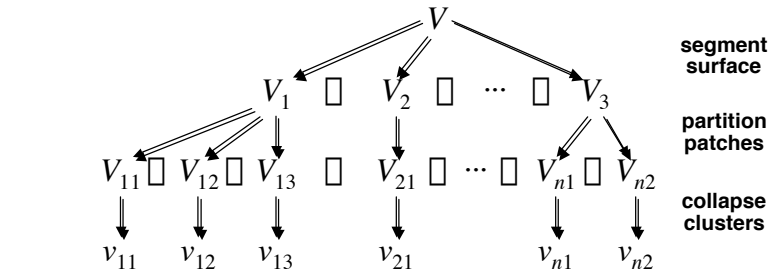
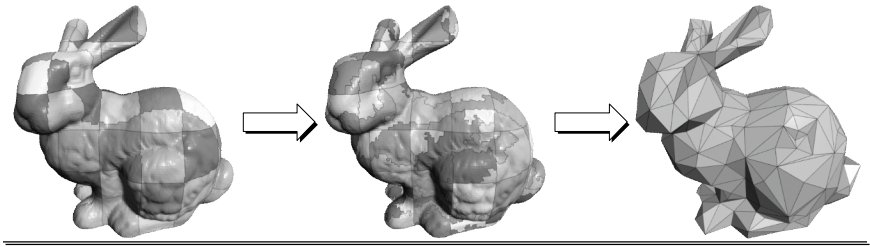


figure courtesy of Hugues Hoppe

Vertex Set Partitioning using Surface Segmentation





Uniform Spatial Surface Segmentation *Bernardini et al. '99*

surface segmentation on a uniform rectilinear grid

- 3D grid partitions mesh into small patches
- simplify patches in-core to given error tolerance
  - must leave patch boundaries intact to allow future merging
- stitch patches together along common boundaries
- displace grid by half a grid cell in each direction
- repeat procedure to coarsen patch seams
- features
  - fairly easy to implement, higher quality than spatial clustering
- drawbacks
  - two orders of magnitude slower than spatial clustering
  - outputs only single level of detail

OEMM: Adaptive Surface Segmentation
Cignoni et al. '02

octree-based external memory mesh data structure

- rectilinear octree grid segments surface
- general data structure for out-of-core mesh processing
  - not only for simplification

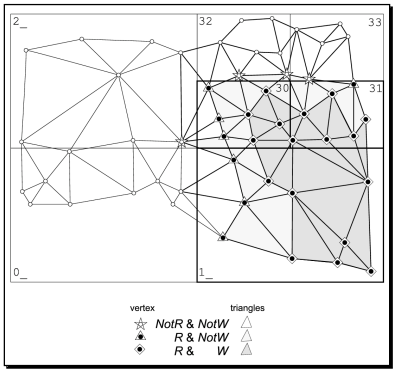


figure courtesy of Paolo Cignoni

OEMM Overview

similar approach to Bernardini et al. '99, but

- octree grid adapts in resolution to better match surface detail
  - siblings merged when triangle count drops below limit
- outputs progressive mesh, e.g. for view-dependent rendering
- by loading grid cells neighboring current patch, edges can be collapsed across patch boundaries
  - no need to keep patch boundaries at full resolution

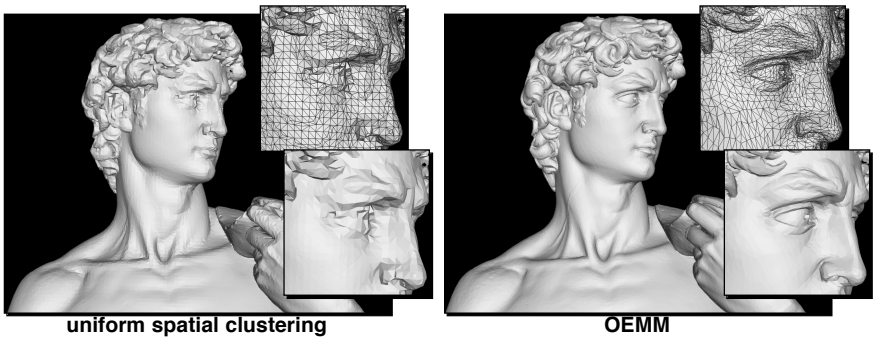
OEMM Results

speed

- **one-time OEMM construction: ~10K triangles per second**
- **simplification: ~10K triangles per second**

quality

images courtesy of Paolo Cignoni

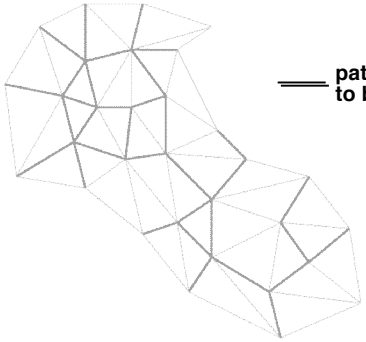


uniform spatial clustering **OEMM**

Error-Driven Surface Segmentation
El-Sana & Chiang '00

disk-based implementation of edge collapse

- **uses B-tree disk data structure to index mesh, priority queue**
- **simplification done in batches by loading pieces of mesh**
  - **surface patches given by faces spanned by edges to be collapsed**



patch spanning edge to be collapsed

figure courtesy of El-Sana & Chiang

Error-Driven Surface Segmentation

algorithm overview

- compute mesh connectivity, initialize priority queue on disk
- for each batch of lowest-cost edges
  - dequeue edges and load incident faces into RAM
  - merge faces into larger patches whenever possible
  - collapse edges, recompute edge costs, write back to disk
- repeat until desired accuracy/size reached
- features
  - entirely error-driven surface segmentation
- drawbacks
  - poor scalability; patches small when input-to-RAM ratio large
  - non-trivial error metrics may force out-of-order edge collapses

Other Surface Segmentation Methods

Prince '00

- extension of *Hoppe '98* terrain simp. to arbitrary surfaces
- uses octree partition and edge collapse
- contrary to *Cignoni et al. '02*, patch boundaries frozen
- 1K triangles per second simplification speed reported

Choudhury & Watson '02

- "simplification in reverse" (i.e. refinement from coarse to fine)
- segments surface into face clusters via binary splitting
- face clusters grouped together on disk
  - increased refinement yields increased locality of reference
  - hence virtual memory disk access possible with little thrashing

Talk Overview

introduction

spatial clustering methods

surface segmentation methods

wrap-up

Summary of Simplification Methods

spatial clustering

- **clusters vertices based on spatial proximity**
  - **adaptive space partitioning possible**
- **features**
  - **very fast (50–250K triangles per second)**
  - **simple to implement**
- **drawbacks**
  - **low quality results**
  - **topology not preserved, non-manifold simplices possible**
- **use when**
  - **time or space is at a premium**
  - **topology simplification is desired**

Summary (continued)

surface segmentation

- clusters vertices primarily based on error
- iterative simplification lends itself to progressive mesh output
- features
  - high quality results
  - can preserve topology
- drawbacks
  - rather slow (1-10K triangles per second)
  - must implement iterative method plus external memory support
- use when
  - quality is more important than speed
  - progressive streaming or view-dependent rendering is needed

Future Work

improved spatial clustering methods

- metric spaces other than Euclidean
  - cluster based on surface geometry (normals, curvature)

hybrid methods

- *Garland & Shaffer '02*
  - first-phase spatial clustering using quadric error metric
  - pass quadric information to second-phase edge collapse

exploit parallelism for "interactive" simplification

alternatives to conventional simplification

- out-of-core remeshing and similar techniques

Acknowledgements

thanks to

- **the *LLNL ASCI VIEWS* team for datasets and support**
- **the *Digital Michelangelo Project* for sharing their datasets**
- ***Yi-Jen Chiang*, *Paolo Cignoni*, *Jihad El-Sana*, *Michael Garland*, and *Hugues Hoppe* for contributing figures and images**
- ***Eric Shaffer* for providing simplified meshes**

References

- **F. Bernardini et al. "Case Study: Scanning Michelangelo's Florentine Pieta." SIGGRAPH 99 course #8.**
- **P. Choudhury & B. Watson. "Completely Adaptive Simplification of Massive Meshes." Northwestern University tech. report CS-02-09, 2002.**
- **P. Cignoni et al. "External Memory Management and Simplification of Huge Meshes." IEEE Trans. on Visualization and Computer Graphics, to appear.**
- **J. El-Sana & Y.-J. Chiang. "External Memory View-Dependent Simplification." Computer Graphics Forum 19(3) 2000, pp. 139-150.**
- **Fei et al. "An Adaptive Sampling Scheme for Out-of-Core Simplification." Computer Graphics Forum 21(2) 2002, pp. 111-119.**
- **Fei et al. "Detail Calibration for Out-of-Core Model Simplification through Interlaced Sampling." SIGGRAPH 2002 tech. sketch, p. 166.**
- **M. Garland & E. Shaffer. "A Multiphase Approach to Efficient Surface Simplification." Visualization '02, to appear.**
- **H. Hoppe. "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering." Visualization '98, pp. 35-42.**

References (continued)

- P. Lindstrom. "Out-of-Core Simplification of Large Polygonal Models." SIGGRAPH 2000, pp. 259-262.
- P. Lindstrom & C. T. Silva. "A Memory Insensitive Method for Large Model Simplification." Visualization '01, pp. 121-126.
- C. Prince. "Progressive Meshes for Large Models of Arbitrary Topology." Master's Thesis, University of Washington, 2000.

A Memory Insensitive Technique for Large Model Simplification

Peter Lindstrom*
LLNL

Cláudio T. Silva†
AT&T

Abstract

In this paper we propose three simple, but significant improvements to the OoCS (Out-of-Core Simplification) algorithm of Lindstrom [20] which increase the quality of approximations and extend the applicability of the algorithm to an even larger class of compute systems.

The original OoCS algorithm has memory complexity that depends on the size of the output mesh, but no dependency on the size of the input mesh. That is, it can be used to simplify meshes of arbitrarily large size, but the complexity of the output mesh is limited by the amount of memory available. Our first contribution is a version of OoCS that removes the dependency of having enough memory to hold (even) the simplified mesh. With our new algorithm, the whole process is made essentially independent of the available memory on the host computer. Our new technique uses disk instead of main memory, but it is carefully designed to avoid costly random accesses.

Our two other contributions improve the quality of the approximations generated by OoCS. We propose a scheme for preserving surface boundaries which does not use connectivity information, and a scheme for constraining the position of the "representative vertex" of a grid cell to an optimal position inside the cell.

CR Categories: E.5 [Files]: Sorting; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Surface and object representations.

Keywords: polygonal surface simplification, large data, out-of-core algorithms, external sorting, quadric error metrics.

1 INTRODUCTION

In recent years there has been a rapid increase in the raw size of polygonal datasets. Several technological trends are contributing to this effect, such as the development of high-resolution 3D scanners, and the need to visualize ASCI-size (Accelerated Strategic Computing Initiative) datasets. A useful paradigm for visualizing large datasets is to generate levels of detail. Over the last decade, there has been substantial research in designing algorithms for generating level-of-detail approximations of triangle meshes. In this paper, our focus is on algorithms which have low memory complexity.

A simplification algorithm receives an input mesh of complexity n, and outputs a mesh of complexity m (where $m < n$). Often, the user sets the target size of the output, and the algorithm attempts to minimize the overall error of the approximation. One important aspect of the design of a surface simplification algorithm is its memory usage. In general, different algorithms have different main memory dependencies on n and m. For different applications, it is useful to have algorithms which are memory efficient with respect to n or m (but ideally both). The memory dependency on n affects the usefulness of a given algorithm in the sense that it limits the size of models that can be simplified.

In general the memory requirement of a given algorithm grows with both m and n (for exceptions, see e.g. [29, 30]). The dependency on m has direct implications on the maximum accuracy of the approximation. As an example, an efficient terrain simplification algorithm is presented in [13], whose memory complexity is analyzed to be $3n + 192m$ bytes, where n and m are the number of vertices in the input and output, respectively. In order to generate a high-quality approximation with one eighth of the input points, i.e., $m = \frac{1}{8}n$, one would need to have $27n$ bytes of memory, or nine times as much as the size of the input. Often, the memory complexity is much higher on both n and m (e.g., [21] uses $160n$ bytes for general surface simplification), and generating approximations of large datasets is usually quite hard.

The OoCS algorithm proposed by Lindstrom [20] is a big step forward in that it has no dependency on n, thus allowing for simplification of extremely large datasets. One contribution of our work is to remove the main memory dependency on m from OoCS, thus allowing for an arbitrarily accurate approximation of an arbitrarily large dataset. Our new algorithm, OoCSx, uses *constant memory*, no matter how large the dataset or approximation error.

One might argue that the ability to produce simplified models that are still too large to represent in-core is of little practical value, since the main reason for simplifying the model in the first place is to reduce its complexity to something more manageable. However, we see several important uses of our new algorithm. First, in many situations it is not known beforehand how much RAM will be available on the client machine on which the simplified mesh is to be used, as is generally the case with multi-level-of-detail datasets provided through data repositories. Second, OoCS does not provide a mechanism for specifying the exact size m of the simplified model, and trial and error may be necessary to find a grid resolution that leads to a detailed simplification that, along with the auxiliary data structures used in OoCS, fits in-core. Our memory insensitive algorithm, on the other hand, is able to finish and output a simplified model regardless of the grid resolution. Third, many applications demand a strict error bound, in which case trading memory for mesh accuracy is not a practical option. As we shall see, even when an explicit error bound is not given, the mesh may be so geometrically complex that the most detailed simplification to fit in-core is of unacceptable visual quality. Finally, our work nicely complements the recent trend of developing efficient out-of-core scientific visualization techniques (see, e.g., [7, 11, 32]). With tools like these in hand, further out-of-core processing of a simplified mesh becomes practical.

Our new technique uses disk instead of main memory. In fact, OoCSx generally needs more disk space than OoCS needs main memory. On the other hand, disk is often much cheaper and more readily available than random access memory. The naive use of disk has the potential for considerable slowdown (as in the case of operating system paging). Our algorithm is carefully designed to avoid

*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 7000 East Avenue, L-560, Livermore, CA 94551, USA; pl@llnl.gov.

†AT&T Labs-Research, 180 Park Avenue, Room D265, PO Box 971, Florham Park, NJ 07932, USA; csilva@research.att.com.

random accesses, thus achieving simplification speeds which, although slower than OoCS, are still quite practical. Our experiments show that OoCSx is typically between two to five times slower than OoCS, while using constant main memory. However, when insufficient main memory is available for OoCS to store the simplified model, OoCSx runs faster. Of course, for large enough models, OoCS is not able to finish at all.

Because OoCS does not make use of connectivity information, it has no way of detecting whether an edge is a boundary edge or not. As a consequence, boundaries are generally poorly preserved by OoCS. We propose a technique for preserving boundaries that does not use any connectivity information. Finally, we sketch a technique for enforcing maximum errors, which constrains the optimal cluster representative to lie inside its grid cell while minimizing the approximation error.

2 RELATED WORK

Polygonal simplification has been a hot topic of research over the last decade, with a vast number of published algorithms. Many of the early simplification algorithms were designed to handle modest size datasets of a few tens of thousands of triangles. Recent improvements in scanning and storage technology, however, have lead to datasets as large as billions of triangles [19, 23]. As a result, a number of methods, particularly for out-of-core visualization, have been proposed for coping with models that are too large to fit in main memory, e.g. [3, 5–8, 17, 24, 28, 31, 32].

Rossignac and Borrel proposed one of the earliest simplification algorithms [26]. Their algorithm partitions space into cube-like cells from a uniform rectilinear grid, and replaces all mesh vertices within a grid cell by a single representative vertex. While simple and fast, their method produces rather low quality meshes, in part due to the simple vertex positioning scheme used in their original algorithm. Lindstrom's OoCS algorithm [20] is also based on vertex clustering on a uniform grid, but has a lower time and memory complexity, and uses a quadric error metric to improve the mesh quality. This method was recently extended by Shaffer and Garland [27], who make two passes over the input mesh. During the first pass, the surface is analyzed and an adaptive (instead of uniform) partitioning of space is made. Using this approach, a larger number of irregular grid cells (and thus samples) can be allocated to the more detailed portions of the surface. However, their algorithm requires more RAM than OoCS in order to maintain a BSP-tree and additional quadric information in-core.

Bernardini et al. describe a radically different approach to out-of-core simplification [4]. Their method splits the model up into separate patches that are small enough to be simplified individually in-core using a conventional simplification algorithm. Special care has to be taken along the patch boundaries. A similar technique was proposed by Hoppe for creating hierarchical levels of detail for height fields [15], which was later generalized by Prince to arbitrary meshes [25]. While conceptually simple, the time and space overhead of partitioning the model and later stitching it together adds to an already expensive in-core simplification process, rendering such a method less suitable for simplifying very large meshes.

El-Sana and Chiang [10] propose an external-memory algorithm to support view-dependent simplification of datasets that do not fit in main memory. Similar to [4, 15, 25], they segment the model into sub-meshes that can be simplified independently and later merged in a preprocessing phase. The segmentation and stitching are made simple by ensuring that edges are collapsed in edge-length order, and guaranteeing that sub-mesh boundary edges are longer than interior edges. During run-time, only the portions of the view-dependence tree that are necessary to render the given level of detail are kept in main memory.

3 OUT-OF-CORE SIMPLIFICATION

In order to describe our new simplification algorithm, we will first provide a brief review of OoCS. For full details see [20]. The input to the algorithm is a set of triangles, stored as triplets of vertex coordinates in a file, and the resolution of a three-dimensional grid. (See the appendix for a disk-based technique on how to transform from indexed meshes to this *dereferenced* format.) The algorithm, which is loosely based on the clustering algorithm by Rossignac and Borrel [26], computes for each cluster grid location a representative vertex. (A set of vertices constitute a "cluster" if they all lie inside the same grid cell.) The position of the representative vertex is chosen so as to minimize the *quadric error* [14] measured with respect to the triangles in the cluster. For each triangle $t = (\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t)$, an associated *quadric matrix* $\mathbf{Q}_t$ is computed:

$$\mathbf{Q}_t = \bar{\mathbf{n}}_t \bar{\mathbf{n}}_t^{\mathsf{T}} \tag{1}$$

$$\bar{\mathbf{n}}_t = \begin{pmatrix} \mathbf{x}_1^t \times \mathbf{x}_2^t + \mathbf{x}_2^t \times \mathbf{x}_3^t + \mathbf{x}_3^t \times \mathbf{x}_1^t \\ -[\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t] \end{pmatrix} \tag{2}$$

where $\bar{\mathbf{n}}_t$ is a 4-vector made up of the area-weighted normal of t and the scalar triple product of its three vertices.[1] $\mathbf{Q}_t$ is then distributed to the clusters associated with each of t's three vertices by adding $\mathbf{Q}_t$ to their quadric matrices.

After summing up all per-triangle quadric matrices in a cluster, we obtain a quadric matrix $\mathbf{Q}_S$ that contains shape information for the piece of surface passing through the grid cell:

$$\mathbf{Q}_S = \sum_t \mathbf{Q}_t = \begin{pmatrix} \mathbf{A} & -\mathbf{b} \\ -\mathbf{b}^{\mathsf{T}} & c \end{pmatrix} \tag{3}$$

Using this decomposition of $\mathbf{Q}_S$, the 3×3 linear system $\mathbf{Ax} = \mathbf{b}$ is solved for the "optimal" representative vertex position $\mathbf{x}$ that minimizes the quadric error. That is, $\mathbf{x}$ is the position that minimizes the sum of squared volumes of the tetrahedra formed by $\mathbf{x}$ and the triangles in the cell. When clustering vertices together, the majority of triangles degenerate into edges or points and can be discarded, thereby reducing the complexity of the model.

3.1 OoCSx

The main idea of our modification of the OoCS algorithm is to eliminate the list of occupied clusters which OoCS allocates in main memory and uses for keeping track of their quadrics. Instead of directly computing quadrics, OoCSx computes $\bar{\mathbf{n}}_t$ for the current triangle t and outputs this information to a file, three times for each of the three vertices, together with the information for what grid cell the vertex belongs to. At the same time, we also output another file which contains the non-degenerate triangles (those triangles that have vertices in three different clusters) represented as indices to the grid cells. Then, we externally sort the file containing the vectors $\bar{\mathbf{n}}_t$ using the grid locations as the primary key. After this, all the information related to one grid cell is placed contiguously in the file. By scanning it, it is possible to compute the quadrics and the optimum location of the representative vertex for a particular grid cell. After the vertices in the simplified mesh have been computed, we are left with the task of associating the grid cell references in the triangle file with vertex representatives. This step is described in more detail below.

Here are the steps of OoCSx in detail:

(1) **Read triangles, compute quadric information for later use**. For each triangle $t = (\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t)$ in the input mesh, we compute $\bar{\mathbf{n}}_t$ (Equation 2). Note that we do not compute any

[1] In this paper, $\bar{\mathbf{a}}$ denotes a 4-vector, and $\hat{\mathbf{a}}$ is a unit-length 3-vector.

quadric matrices at this point. For each vertex $\mathbf{x}_i^t$ of t, we also determine the grid location $G(\mathbf{x}_i^t)$ (as an integer ID) that the vertex will be mapped to. As triangles are read, we output this information to two files:

- A "plane equation" file, which contains 3 entries for each triangle, one for each vertex. Each entry is of the form: $\langle G(\mathbf{x}_i^t), \bar{\mathbf{n}}_t \rangle$. Using 32-bit integers to represent G and 32-bit floats for $\bar{\mathbf{n}}_t$, this file takes 20 bytes of disk per entry.
- A "triangle cluster" file, which contains records of the form $\langle G(\mathbf{x}_1^t), G(\mathbf{x}_2^t), G(\mathbf{x}_3^t) \rangle$. Each record takes 12 bytes, and is written only when $G(\mathbf{x}_1^t) \neq G(\mathbf{x}_2^t) \neq G(\mathbf{x}_3^t)$.

(2) **Sort "plane equation" file using G as the sort key**. This step is performed using an external sort algorithm, which is discussed below.

(3) **Compute quadrics and output optimal vertices**. In order to find the representative vertex for a given cluster, we need to sum up all the quadrics that contribute to its position. Because the "plane equation" file has been sorted on cluster IDs (i.e., G), all the vectors $\bar{\mathbf{n}}_t$ that contribute to a given grid cell are together in the file. That is, in a single scan, we can sum all the $\bar{\mathbf{n}}_t \bar{\mathbf{n}}_t^\mathsf{T}$ into a quadric matrix $\mathbf{Q}_S$, which is used to compute the representative vertex position.[2]

As we find the representative vertex $\mathbf{x}$ for a given grid cell G, we output 16-byte records $\langle G(\mathbf{x}), \mathbf{x} \rangle$. Note that we get this file already in "sorted" order for free.

(4) **Replace cluster IDs in triangle file with corresponding vertices**. At this point, the file with the representative vertices and the "triangle cluster" file hold the complete simplified mesh. A more useful format for this data is to "dereference" the triangle cluster file and create a file which lists the vertices of each triangle. This can be done in three passes, one for each of the three fields $G(\mathbf{x}_i^t)$. In each pass, the triangle file is sorted on the current vertex field. After each sort, the cluster IDs are scanned and replaced with entries from the representative vertex file, which is read sequentially, in tandem. Many applications prefer an indexed mesh representation, for which one would replace the cluster IDs with vertex indices.

Time and Space Complexity

The memory usage of the OoCSx algorithm we have described does not depend on the size of the input dataset. The algorithm just needs to have enough memory to hold the data structures for one triangle and perform the other calculations for computing the quadrics and optimal vertices. In fact, we use slightly more memory in our external sort implementation, which by default uses four megabytes of memory. Overall, on a PC running Linux, the code never uses more than five megabytes of memory (eight megabytes on IRIX due to larger executables) regardless of the size of the input dataset or the level of approximation desired.

The time complexity of OoCS is $O(n)$, since it only performs a single scan over the mesh file and keeps all the information regarding the quadrics in main memory. Because of the need to sort several files, OoCSx has time complexity $O(n \log n)$.

[2] Although our input and output files use single-precision floating point numbers, we perform the in-memory computations in double precision. 32-bit floats do not provide enough precision for the computations done for very large models like the St. Matthew statue and fluid isosurface.

External Sorting

At the center of OoCSx are a series of external sorts. External sort algorithms are very important for the design and implementation of I/O-efficient algorithms (see [1, 16]). There are several issues in implementing external memory algorithms, and these issues can greatly affect the overall performance of a system. In general trying to mimic the interface of the C **qsort** routine, although often pursued, does not seem the most efficient implementation technique. In our experience with different external sorts [2, 12, 18], the most efficient implementation uses a combination of radix and merge sort, for which the keys are compared lexicographically. A particularly efficient external sort is **rsort** written by John Linderman at AT&T Research [18]. We use **rsort** for the results presented in this paper. Luckily, it is relatively easy to generate keys which can be compared lexicographically (see the man page for **fixcut**, also from Linderman). In OoCSx, we only need integer keys. For these, we simply have to write them in big-endian format.

3.2 Quality Improvements

Surface Boundary Preservation

Because OoCS does not make use of connectivity information, it has no way of detecting whether an edge is a boundary edge or not. Consequently, surface boundaries are not well preserved by the method. We propose a variation on the technique used by Garland and Heckbert [14], which makes use of planes parallel to the boundary edges and orthogonal to their incident triangles.

Building on this idea, we can create an edge quadric. For each half-edge e of each triangle, we compute a plane $\bar{\mathbf{m}}_e$ that passes through the two vertices of e. The normal vector $\mathbf{m}_e$ of this plane is orthogonal to both e and the normal of the face that e belongs to (Figure 1). The distance of a point to this plane provides a measure of how close the point is to the associated edge. We are here only concerned with distances parallel to the plane of the incident face—the per-triangle quadrics from Equation 2 already penalize deviations orthogonal to the face. Using these definitions, we distribute for each half-edge $e = (\mathbf{x}_1^e, \mathbf{x}_2^e)$ its plane equation $\bar{\mathbf{m}}_e$ to the clusters corresponding to its two vertices. After adding up all the plane equations (4-vectors) in a cluster, we compute a quadric matrix $\mathbf{Q}_B$ for the boundary as:

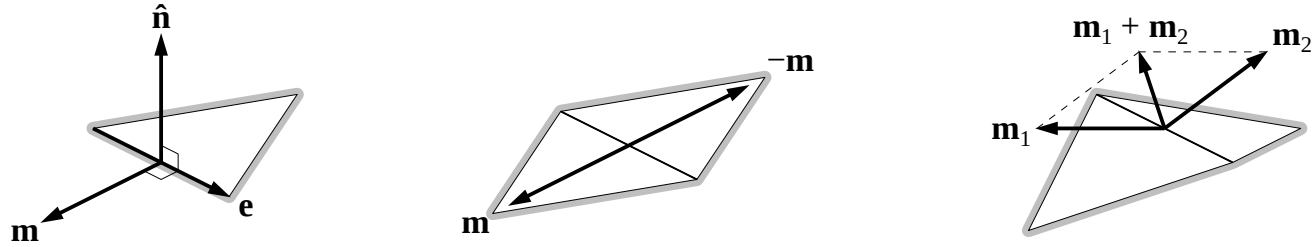
$$\mathbf{Q}_B = \left(\sum_e \bar{\mathbf{m}}_e \right) \left(\sum_e \bar{\mathbf{m}}_e \right)^\mathsf{T} \tag{4}$$

$$\bar{\mathbf{m}}_e = \begin{pmatrix} \mathbf{m}_e \\ -\frac{1}{2}(\mathbf{x}_1^e + \mathbf{x}_2^e)^\mathsf{T} \mathbf{m}_e \end{pmatrix} \tag{5}$$

$$\mathbf{m}_e = \|\mathbf{e}_e\| (\mathbf{e}_e \times \hat{\mathbf{n}}_e) \tag{6}$$

$$\mathbf{e}_e = \mathbf{x}_2^e - \mathbf{x}_1^e \tag{7}$$

Note that all edges, whether manifold or on the boundary, are treated equally. What makes the algorithm sensitive to boundary edges is that, when adding the implicit plane equations $\bar{\mathbf{m}}_e$, there is no opposing half-edge from the neighboring triangle to cancel $\bar{\mathbf{m}}_e$. This is illustrated in Figure 1(b), where the plane equations for two adjacent coplanar faces exactly cancel each other. For non-coplanar faces, the plane equations will not totally cancel, but a residual vector (the normal vector of a new plane) remains that penalizes positions away from the edge in the plane that bisects the dihedral angle formed by the two triangles. The sharper an edge is, the larger this penalty becomes. When used as part of an error measure, this would tend to preserve sharp edges, which is often desired. Based on this argument, non-manifold edges would also tend to be preserved, which is likely desirable since they typically form sharp creases in the mesh. Note that this scheme makes no use



(a) vector **m** orthogonal to boundary edge (b) coplanar faces (c) non-coplanar faces

Figure 1: Illustration of the vectors used for surface boundary preservation. The boundary normal **m** is orthogonal to the face normal $\hat{\mathbf{n}}$ and the vector **e** along the edge *e*. For manifold edges that share two coplanar faces, the boundary normals cancel. In the case of non-coplanar faces, the residual vector $\mathbf{m}_1 + \mathbf{m}_2$ lies in the plane that bisects *e*'s dihedral angle.

of connectivity information, yet implicitly accounts for the feature edges in the mesh.

The final quadric for the cluster is computed as a linear combination $\lambda\mathbf{Q}_S + (1-\lambda)\mathbf{Q}_B$ of the surface quadric and the new boundary quadric. Note that we have been careful to weight the boundary quadric so as to ensure scale invariance and compatibility with the area-squared weighted triangle quadrics. We have found that weighting the quadrics equally ($\lambda = \frac{1}{2}$) tends to give good results.

Constrained Optimization over Cell Boundaries

As discussed in [20], the minimum quadric error sometimes falls outside the cluster's grid cell. While rare, the minimum may be arbitrarily far from the grid cell given the right conditions. Our previous approach to handling these degeneracies was to use one of a number of ad hoc methods for clamping the vertex coordinates, such as projecting the vertex onto the grid cell boundary. To ensure that the vertex is contained in the grid cell, but also results in the smallest possible quadric error, we perform a linearly constrained optimization over the grid cell boundary whenever the global optimum is outside it. Because the quadric functional is quadratic and the grid cell constraints are linear, the solution to this optimization problem can be found by solving a set of linear equations (cf. [22]). This optimization problem is made particularly easy by the fact that the linear constraints are all perpendicular to each other and parallel to the coordinate axes, and can therefore generally be solved as a 2D or 1D problem.

4 EXPERIMENTAL RESULTS

Table 1 summarizes our experimental results. We used two machines for our experiments, most of which were performed on a Linux PC with 512 MB of main memory and two 800 MHz Pentium III processors. The simplification of the statue and fluid isosurface was performed on one processor of a SGI Onyx2 with forty-eight 250 MHz R10000 processors and 15.5 GB of main memory. On the SGI, we used one of its one-terabyte striped disks. Overall, OoCSx was between two to five times slower than OoCS, but sometimes the speed difference was even smaller. In one case, for a high-resolution simplification of the blade, OoCSx was faster than OoCS. The reason for this is that OoCS ran out of memory, and numerous page faults occurred. This happened while trying to simplify the blade to one quarter of its initial size. The ratio in memory usage of OoCS and disk usage of OoCSx varied widely, going from a low of 6 to a high of 245! These variations are due to the dependency on *n*, the size of the input model, in OoCSx, whereas the memory usage of OoCS is proportional to *m*, the size of the output model. For the external sort code **rsort** used in our implementation, we empirically determined the maximum disk usage of OoCSx to be $172T_{in} + 12T_{out}$ bytes.[3] These results indicate that **rsort** requires roughly twice the input size of additional storage. If necessary, there are techniques for lowering the disk overhead of OoCSx. For instance, it would be possible to perform multiple sorts, instead of a single one, and accumulate phases if disk space is at a premium.

model	T_{in}	T_{out}	*RAM:disk (MB)*		*time (h:m:s)*	
			OoCS	*OoCSx*	*OoCS*	*OoCSx*
dragon	871,306	47,236	4:0	5: 150	6	13
		113,058	9:0	5: 152	7	14
		244,568	21:0	5: 153	9	17
buddha	1,087,716	62,346	5:0	5: 187	7	16
		204,766	20:0	5: 191	10	19
blade	28,246,208	507,104	49:0	5: 4,850	2:46	13:14
		1,968,172	160:0	5: 4,899	3:11	14:30
		7,327,888	859:0	5: 4,993	19:14	17:04
statue	372,963,401	3,012,996	261:0	8:64,004	44:22	2:37:24
		21,506,180	3,407:0	8:64,256	51:23	2:49:30
fluid	467,614,855	6,823,739	588:0	8:80,334	55:56	3:11:48
		26,086,125	3,427:0	8:80,510	1:08:48	3:23:42
		94,054,242	-	8:81,345	-	4:19:09

Table 1: Run-time performance of OoCS and OoCSx. The results reported for the dragon, buddha, and blade were computed on a Linux PC. The statue and fluid models were simplified on a SGI Onyx2. Even on the 15.5 GB SGI, not enough RAM was available for OoCS to produce the finest level of detail of the fluid dataset.

Figure 4 shows the effect of using edge quadrics in the simplification of the boundary (shown in red) of the bunny. From this figure, it is evident that the boundaries have been preserved with better visual accuracy. This subjective result is also supported numerically by Figure 2, which shows the maximum (Hausdorff) and root mean square (RMS) distances between closest points on the boundaries for several levels of detail of the bunny. These error measures were evaluated symmetrically by considering all points on the boundaries of both the original and the simplified model. Clearly, the use of boundary quadrics greatly reduced the boundary errors. In addition, we found that the use of boundary quadrics did not negatively impact the errors measured over the surface interiors. Instead, using boundary quadrics reduced both boundary and surface errors for models with boundaries, and did not result in a measurable increase in surface error for models without boundaries.

Figure 3 is an isosurface of a time slice from a large-scale turbulent-mixing fluid dynamics simulation, consisting of $2,048 \times 2,048 \times 1,920$ voxels at 27,000 time steps [23]. This surface is challenging to simplify due to its highly complex topology and wispy geometry. Table 1 lists the performance data for simplifying the entire isosurface. To avoid too much clutter in the images presented here, we also extracted a small piece (one third of a percent) of the volume and simplified it independently (Figure 6). As can be

[3] This usage is for the intermediate files only, and does not include the space needed for the input and output files.

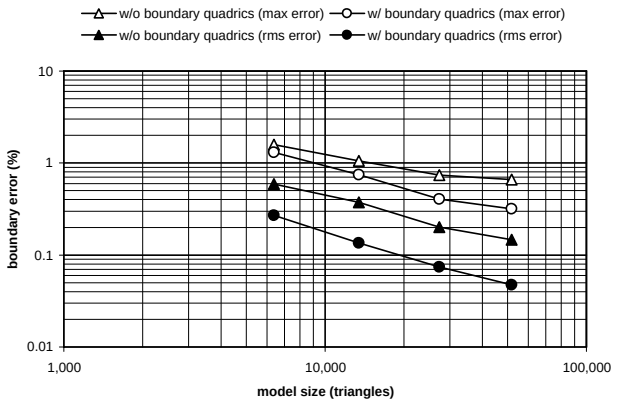


Figure 2: Maximum and root mean square boundary error for bunny model, simplified with and without boundary quadrics.

seen in Figure 6(e), there is significant loss in topological structure and geometric detail as the triangle count drops to a few million. A simplification of a complex dataset like this requires more triangles than can be stored in RAM on most computers, and must be simplified using a memory insensitive method such as OoCSx. Notice also the improved boundaries in Figure 6(d) over the model simplified without boundary quadrics (Figure 6(b)).

Finally, we evaluated the effect of performing constrained optimization over the cell boundary in those cases where the optimal vertex position lies outside the cell. We compared this approach to (1) leaving the vertex outside the cell, and (2) projecting it onto the cell boundary. In all cases, the constrained optimization performed as well or better than the other two approaches, both in terms of maximum and RMS error. Figure 5 shows an example where constrained optimization resulted in nearly a factor of six reduction in the maximum error over leaving the vertices unclamped. Notice how the artifacts near the lower jaw, ears, and hind leg are eliminated by clamping and optimizing the vertices, leaving a more visually pleasing model.

5 CONCLUSIONS

In this paper, we proposed improvements to the out-of-core simplification (OoCS) technique [20]. First, we described OoCSx, a memory insensitive variation of OoCS. The key feature of OoCSx is its ability to efficiently simplify arbitrarily large datasets using a constant amount of main memory. OoCSx uses a disk-based technique for storing information about the simplified mesh and arranging it in a cache-coherent manner. We also discussed an efficient implementation of OoCSx and compared its performance with OoCS. Second, we proposed a technique for preserving surface boundaries without making use of connectivity information. Our approach is to compute and minimize an edge-based quadric error for all edges of the mesh, regardless of their topological type. We showed that this technique can dramatically improve the shape of boundary curves, with little or no loss in geometric quality over the remaining surface. Finally, we proposed using a linearly constrained optimization over grid cell boundaries to compute vertex positions whenever the global optimum is outside the grid cell.

One shortcoming of the current approach is that the overall simplification has constant feature size. Similar to [27], it would be interesting to extend OoCSx to simplify the mesh adaptively. Taking this one step further, we will investigate how to adapt our out-of-core algorithms to perform dynamic view-dependent refinement of the mesh for interactive visualization. Another drawback of OoCSx is that it requires significant amounts of disk space. The per-triangle quadric information stored on disk constitutes a large portion of the overall space requirements. We believe that careful encoding of these 4-vectors, using normal quantization [9] and per-grid-cell coordinate representations, will allow this information to be represented using as little as 32 bits per vector. Finally, many datasets come with surface attributes such as scalar field values, normal and curvature information, and color. We hope to extend our simplification code to take into account and preserve such information.

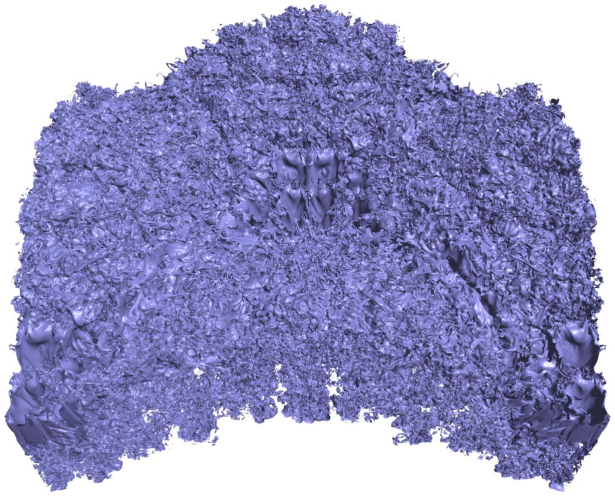


Figure 3: 470 million triangle isosurface of entire fluid dynamics dataset.

Acknowledgements

This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48. We would like to thank the reviewers for useful comments. Many thanks to Glenn Fowler and John Linderman for several discussions and access to their external sorting code. We wish to thank Stanford University and the Digital Michelangelo Project for providing the bunny, dragon, Buddha, and St. Matthew datasets, and Kitware for the turbine blade model. Thanks to David Bremer, Mark Duchaineau, and Randy Frank for preparing the fluid dynamics dataset.

Appendix: Dereferencing Indexed Meshes

The file format we assume in our algorithm is different from the indexed mesh formats commonly used for main memory techniques. In main memory, it is common to store a list of vertex coordinates (x,y,z), and a list of triangles, represented by three integers that refer to the vertices of the given triangle. Before such datasets can be used in our algorithm, they need to be "normalized", a process that dereferences the pointers to vertices. This process is thoroughly explained in [7]. For completeness, we briefly explain how to normalize such a file with V vertices and T triangles. In an initial pass, we create two (binary) files, one with the list of vertices, and another with the list of triangles. Next, in three passes, we dereference each index in the triangle file, and replace it with the actual position for the vertex. In order to do this efficiently, we first (externally) sort the triangle file on the index we intend to dereference. This takes time $O(T \log T)$ using an (external memory) mergesort. Then, we perform a synchronous scan of both the vertex and the (sorted) triangle file, reading one record at a time, and appropriately outputting the deferenced value for the vertex. Note that this can be done efficiently in time $O(V+T)$ because all the vertex references are sorted. When we are done with all three passes, the triangle file will contain T records with the "value" (not reference) of each of its three vertices.

References

[1] J. Abello and J. Vitter. *External Memory Algorithms and Visualization*. American Mathematical Society Press, 1999.

[2] L. Ammeraal. *Algorithms and Data Structures in C++*. Addison Wesley, 1996.

[3] C. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel Accelerated Isocontouring for Out-of-Core Visualization. *Proceedings of IEEE Parallel Visualization and Graphics Symposium 1999*, pages 97–104, October 1999.

[4] F. Bernardini, J. Mittleman, and H. Rushmeier. Case study: Scanning Michaelangelo's Florentine Pietà. In *ACM SIGGRAPH 1999 Course notes, Course #8*, August 1999.

[5] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. The Ball-Pivoting Algorithm for Surface Reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, October - December 1999.

[6] Y.-J. Chiang and C. T. Silva. I/O Optimal Isosurface Extraction. *IEEE Visualization '97*, pages 293–300, November 1997.

[7] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive Out-of-Core Isosurface Extraction. *IEEE Visualization '98*, pages 167–174, October 1998.

[8] M. B. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. *IEEE Visualization '97*, pages 235–244, November 1997.

[9] M. F. Deering. Geometry Compression. *Proceedings of SIGGRAPH 95*, pages 13–20, August 1995.

[10] J. El-Sana and Y.-J. Chiang. External Memory View-Dependent Simplification. *Computer Graphics Forum*, 19(3), August 2000.

[11] R. Farias and C. Silva. Out-of-Core Rendering of Large Unstructured Grids. *IEEE Computer Graphics & Applications*, 21(4):42–50, 2001.

[12] G. Fowler. AST **sort**. http://www.research.att.com/sw/download.

[13] M. Garland and P. Heckbert. Fast Polygonal Approximation of Terrains and Height Fields. Technical Report CMU-CS-95-181, Carnegie Mellon University, 1995.

[14] M. Garland and P. Heckbert. Surface Simplification Using Quadric Error Metrics. *Proceedings of SIGGRAPH 97*, pages 209–216, August 1997.

[15] H. H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. *IEEE Visualization '98*, pages 35–42, October 1998.

[16] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[17] S. Leutenegger and K.-L. Ma. Fast Retrieval of Disk-Resident Unstructured Volume Data for Visualization. In *External Memory Algorithms and Visualization*, DIMACS Book Series, American Mathematical Society, vol. 50, 1999.

[18] J. Linderman. **rsort** and **fixcut** man pages. April 1996 (revised June 2000).

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

[20] P. Lindstrom. Out-of-Core Simplification of Large Polygonal Models. *Proceedings of SIGGRAPH 2000*, pages 259–262, July 2000.

[21] P. Lindstrom and G. Turk. Fast and Memory Efficient Polygonal Simplification. *IEEE Visualization '98*, pages 279–286, October 1998.

[22] P. Lindstrom and G. Turk. Evaluation of Memoryless Simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, April - June 1999.

[23] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimitis, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System. *Proceedings of Supercomputing 99*, November 1999.

[24] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Proceedings of SIGGRAPH 97*, pages 101–108, August 1997.

[25] C. Prince. Progressive Meshes for Large Models of Arbitrary Topology. M.S. thesis, University of Washington, 2000.

[26] J. Rossignac and P. Borrel. Multi-Resolution 3D Approximation for Rendering Complex Scenes. In *Modeling in Computer Graphics*, pages 455–465, 1993.

[27] E. Shaffer and M. Garland. Efficient Adaptive Simplification of Massive Meshes. *IEEE Visualization '01*, October 2001.

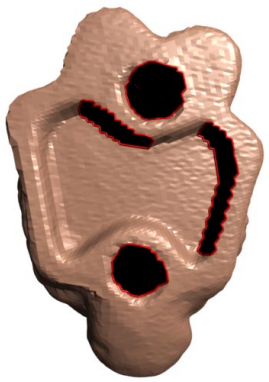
[28] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree. *IEEE Visualization '99*, pages 371–378, October 1999.

[29] C. Silva, J. Mitchell, and A. E. Kaufman. Automatic Generation of Triangular Irregular Networks Using Greedy Cuts. *IEEE Visualization '95*, pages 201–208, November 1995.

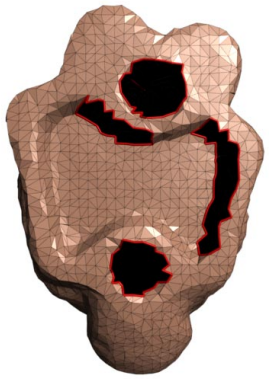
[30] C. Silva and J. Mitchell. Greedy Cuts: An Advancing Front Terrain Triangulation Algorithm. *Proceedings of the 6th ACM Workshop on Advances in GIS*, pages 137–144, November 1998.

[31] P. M. Sutton and C. D. Hansen. Accelerated Isosurface Extraction in Time-Varying Fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):98–107, April - June 2000.

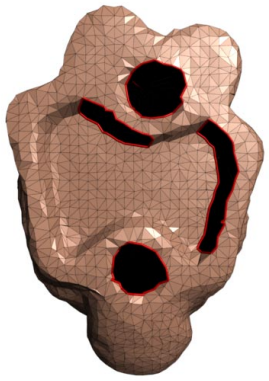
[32] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-Core Streamline Visualization on Large Unstructured Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, October - December 1997.



(a) original; $T = 69,451$



(b) w/o boundary quadrics; $T = 14,761$

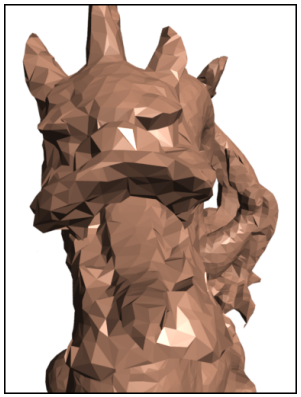


(c) w/ boundary quadrics; $T = 14,761$

Figure 4: Bunny model simplified with and without the surface boundary preservation technique.

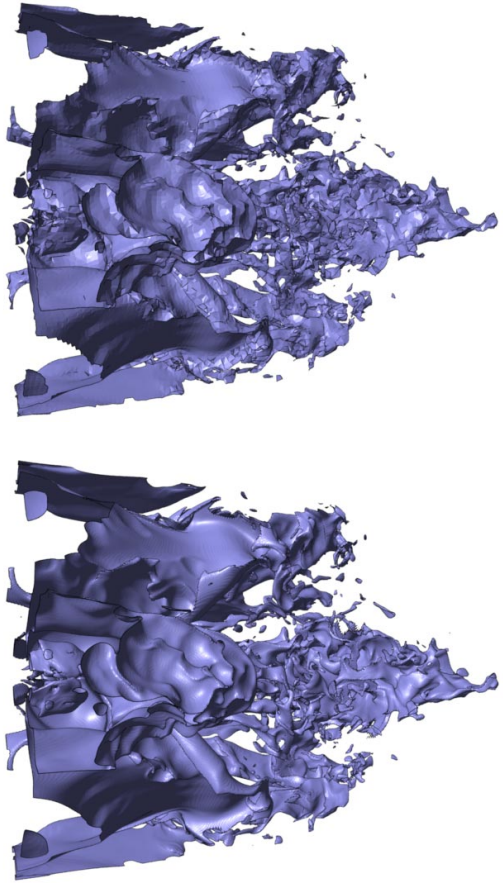
(a) original; $T = 871,306$

(b) without clamping; $T = 20,292$

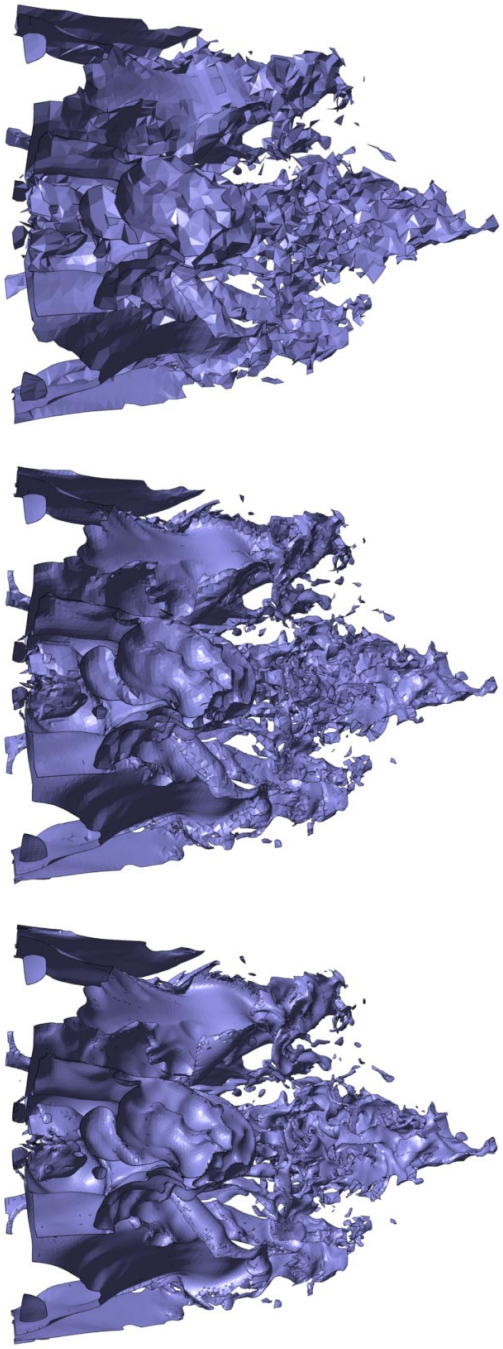


(c) with clamping; $T = 20,292$

Figure 5: Use of constrained optimization over cell boundaries to reposition stray vertices.



(a) original; $T = 467,614,855$



(b) without boundary quadrics; $T = 26,086,125$

(c) with boundary quadrics; $T = 94,054,242$

(d) with boundary quadrics; $T = 26,086,125$

(e) with boundary quadrics; $T = 6,823,739$

Figure 6: Small subset of isosurface of turbulent-mixing fluid dynamics simulation. The triangle counts correspond to simplifications of the entire dataset.

External Memory Management and Simplification of Huge Meshes

P. Cignoni, C. Montani, C. Rocchini, R. Scopigno

Abstract— **Very large triangle meshes, i.e. meshes composed of millions of faces, are becoming common in many applications. Obviously, processing, rendering, transmission and archival of these meshes are not simple tasks. Mesh simplification and LOD management are a rather mature technology that in many cases can efficiently manage complex data. But only few available systems can manage meshes characterized by a huge size: RAM size is often a severe bottleneck.**

In this paper we present a data structure called Octree-based External Memory Mesh (*OEMM*). It supports external memory management of complex meshes, loading dynamically in main memory only the selected sections and preserving data consistency during local updates. The functionalities implemented on this data structure (simplification, detail preservation, mesh editing, visualization and inspection) can be applied to huge triangles meshes on low-cost PC platforms. The time overhead due to the external memory management is affordable. Results of the test of our system on complex meshes are presented.

CR Descriptors: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - *Curve, surface, solid and object representation*; **I.3.6 [Computer Graphics]: Methodology and Techniques.**

Additional Keywords: Out-Of-Core Algorithms, Hierarchical Data Structures, Mesh Simplification, Level of Detail, 3D Scanning, Texture Synthesis.

I. Introduction and state of the art

Very large triangle meshes, i.e. meshes composed of many millions of faces, are common in many applications: range scanning, volume data visualization, terrain visualization, etc. For example, huge meshes (up to Giga triangles sizes) can be produced by scanning Cultural Heritage artifacts [3], [16] or by processing large volumetric dataset (e.g. the data produced by the Visible Human project or the DOE ASCII project). Obviously, such complex meshes introduce severe problems in the archival, manipulation, visualization and geometric processing. Huge mesh management encompasses different processing goals:

- Efficient visualization, for selective inspection and presentation (direct raw mesh visualization is inefficient when we want to focus on a small dataset region).
- Mesh editing functionalities, to improve the quality of the data (e.g., 3D scanned meshes need smoothing filters or operators for the triangulation of holes).
- High quality simplification capabilities, to allow the construction of LOD representations (even if alternative representations exist, e.g. based on point-based primitives [24], topology is crucial in a number of applications and triangles are still the standard graphics primitive).
- Finally, applications' specific functionalities can be required (e.g. computing digital measures on the model or supporting data conversion tools for rapid prototyping).

The adoption of an *External Memory* (EM) technique is mandatory whenever we want to process a huge mesh on a limited core memory footprint. The design of EM solutions is a very active research area, and many groups are working on this issue in the graphics community as well. Recent results are: EM isosurfaces fitting [4], EM reconstruction of surfaces from point clouds [2], EM visualization [29], or EM solutions for the simplification of huge meshes [13], [17], [7], [18], [28]. Let us focus on the mesh simplification task.

A. Mesh Simplification

In the context of geometric processing, mesh simplification can be considered a crucial task, and core memory is often the bottleneck [10]. Almost all simplification tools require the whole mesh to be loaded in main memory. If we consider Quadric Error edge-collapse simplification [11], the space complexity can be estimated as a factor of the mesh size (176 byte for each face). Therefore, we can process around 1.1M-1.3M faces on a system with 256MB RAM.

Various techniques have been presented to face the problem of huge mesh simplification: Hoppe's hierarchical method for digital terrain management [13], that can be extended to 3D meshes as shown in [22], [8]; the clustering solution proposed by Lindstrom [17]; and the spanned mesh simplification algorithm by El-Sana et al. [7].

Hoppe hierarchically divides the mesh in blocks, simplifies each block by *collapsing edges* (the collapse of elements incident on the boundary of the block is forbidden) and then traverses bottom-up the hierarchical structure by merging sibling cells and again simplifying. This approach has either a bottleneck on the output size (because a complete bottom-up traversal of the tree is required to remove elements incident on the inter-cell boundaries) or on the simplification accuracy (intermediate results present unpleasant runs of original high resolution elements located on cells boundaries). Moreover, this approach cannot be extended easily to support other geometric processing tasks, because the elements shared by adjacent blocks cannot be modified unless the blocks are merged. These disadvantages are shared with the 3D mesh extensions of the Hoppe's approach [8], [22].

The *clustering* algorithm [23] can be easily implemented in external memory [17] and guarantees excellent time effi-

Istituto Scienza e Tecnologie dell'Informazione - CNR (formerly IEI-CNR and CNUCE-CNR), Area della Ricerca CNR, v. G. Moruzzi 1, 56124 Pisa, ITALY. E-mail: {cignoni|montani|rocchini}@iei.pi.cnr.it , r.scopigno@cnuce.cnr.it .

ciency. Unfortunately, the accuracy of the mesh produced is much lower if compared with the accuracy of methods based on edge collapse. The simple criterion adopted (unify all mesh elements that are contained in the same cluster) implies that every shape feature whose size is smaller than a cluster cell is removed. Clustering performs an *accurate* (it is based on quadric error metrics) but *regular* sub-sampling in the model space. Therefore, it is not able to simplify large sections of the mesh which have a low curvature variation and span multiple cells. A disadvantage of the original clustering approach is that intermediate simplification results are maintained in main memory. This prevents simplification when an intermediate reduction rate is requested. This latter problem has been recently solved by Lindstrom and Silva [18] by storing the output mesh and intermediate data on disk (out-of-core sorting is used to detect and compose the quadrics associated to each grid cell); output size independence is obtained at the expenses of two to five times slower simplification times. Another improvement over the general Clustering approach has been proposed recently by Shaffer and Garland [28]. A higher quality approximation is obtained at the expenses of a small time overhead (around two times slower than standard clustering) by replacing the regular grid with an adaptive subdivision based on BSP trees. The external memory implementation needs multiple scan of the data: initially, a uniform grid is used to quantize the input data and compute quadrics; then, this info allows to build an adaptive subdivision of the space (by a BSP tree), that is used in the last step to simplify the mesh. Even if this method gives an improved accuracy with respect to standard clustering (given a budget of K output vertices, these are positioned on the surface in an adaptive manner), the accuracy is lower than that produced with edge-collapse methods and the output is often non-topologically clean, as well as all clustering solutions (see some discussion in Section VIII).

The spanned mesh simplification algorithm by El-Sana et al. [7] starts from an indexed mesh with explicit topology. It keeps all the edges of the mesh (with the adjacent faces) into an external memory heap, ordered according an error criterion based on edge length (using edge length does not ensure high accuracy in simplification. Implementing an ordering criteria based on quadric error metrics [11] is not easy, due to the more complex data loading required for the initial evaluation of the QEM for each edge and for the update of QEM after each edge collapse. Given the k edges on top of the heap (k depends on the core memory size), it loads in memory the associated adjacent faces pairs and reconstructs the mesh portions spanned by these edges. Then, the edges having all their incident faces loaded in memory can be collapsed. This approach reaches a good computational efficiency if we are able to load in memory a large percentage of the data (i.e. large contiguous regions). In the case of huge meshes the shortest edges could be uniformly scattered and it could happen that most of the spanned sub-meshes loaded in memory consist of only a few triangles, therefore requiring a very frequent loading/unloading of very small regions. A positive advantage of this method is that the simplification order performed by the external memory implementation is exactly identical to the one used by an analogous in-core solution (thus, the mesh produced by the external memory implementation is identical to the one of the in-core solution).

Beside the specific limitation of each one of the above techniques, most of them have been designed to support just simplification, and extending these approaches to support also other geometric processing algorithms can be not straightforward.

B. Objectives

Our goal is therefore to support general huge mesh management on low-cost platforms, by providing mesh manipulation, editing, filtering, simplification and inspection features under the constraint of a limited memory size. None of the existing systems support these features, especially if we consider PC-based systems. Our system is based on a hierarchical data structure, called Octree-based External Memory Mesh (OEMM). Hierarchical schemes [25] have been often used in geometric processing and interactive visualization [1], [8], [9], [13], [15], [22], [24], [29], but in all these cases the hierarchical structure sets strong limitation on how and where processing can be performed. For example, the hierarchical simplification approach [13], [8], [22] simplifies some boundary elements only in the very last step of the bottom-up simplification process (i.e. boundary elements can be managed only when the corresponding leaf nodes are merged). Our external memory structure is not just another space subdivision or data paging scheme. Peculiar characteristics of our approach are: (a) it supports a global indexed representation (built on any huge mesh given in input as a triangle soup); (b) it allows any partial data load/update/write-back operation, by performing an automatic *on the fly* re-indexing of the loaded data portion: in this way, any loaded portion is represented in core memory with indexed lists containing only the loaded vertices and faces. Data subdivision is performed using a standard octree-based regular split; elements spanning adjacent cells are identified in the construction phase, consistent id's are assigned to the corresponding vertices in adjacent nodes (vertex indexing also satisfies the lexical order of the corresponding octree nodes) and, finally, each border element is assigned to a single node of the octree. This allows data loading of any subset of the mesh, which is converted on the fly in a single, consistent mesh indexed on the local subset of vertices. The potential boundary elements contained in the interior of the loaded region can therefore be treated as any other element, while a *tagging strategy* (the peculiar characteristic of our approach) allows easy detection and management of the elements located on the boundary of the current region. This makes simple the design of the external memory version of many geometric algorithms. Therefore, the underlying space decomposition is completely hidden (and managed by the data structure), and coding geometrical algorithms working on data partitions becomes easier.

Thanks to the freedom of accessing any small subset of

the mesh consistently, we can easily implement different mesh processing algorithms on the *OEMM* data structure, such as: *mesh editing* and *selective inspection*; high quality *mesh simplification* (based on the quadric error metric approach [11]); *detail preservation* (based on bump- or rgb-texture resampling, to encode the high frequency detail lost during simplification [5]). The bottleneck on either input and output data size is thus removed.

The paper is organized as follows. Some definitions are introduced in Section II. Then, the *OEMM* hierarchical structure is introduced in Section III. Details on the construction of an *OEMM* representation from a triangle soup (list of faces, not indexed) are given in Section IV. Section V presents the data management procedures (traversal, loading, updating). Section VI describes how to implement external memory mesh simplification. Other mesh processing tasks have been implemented on the *OEMM* data structure, and are briefly described in Section VII. Finally, Sections VIII and IX report results and conclusions.

II. Definitions

Mesh Terminology. A mesh is called *indexed* if all the triangles are encoded by storing a triple of references to their vertices (either with explicit pointers or with integer indices). Conversely, it is called *raw* (or triangle soup) if the triangles are described with a triple of 3D points and sharing of vertices among adjacent triangles is not considered.

Octree Terminology. Given an axis aligned box B containing the dataset, we recursively partition it in eight sub-regions [20]. Sub-regions are numbered according to their relative coordinates in lexicographic order (see Figure 1 for a 2D example), which defines a total ordering between octree leaves according to a DFS visit [25]. Given an octree node n, we denote with B_n the bounding box corresponding to that node. Each bounding box B is identified by two 3D points $B.min, B.max$.
To avoid ambiguities hereafter when we say that a point p is *contained* into a bounding box B we mean that its coordinates are greater than or equal to $B.min$ and less than $B.max$. In this way any point is contained in one and only one *leaf* node of the octree.

III. Octree-based External Memory Mesh

The Octree-based External Memory Mesh data structure (*OEMM*) provides support for the management of generic processing on huge meshes, under the constraint of limited core memory. *OEMM* is based on a hierarchical geometric partition of the dataset with no vertex replication and consistent vertex indexing between leaf nodes which shares a reference to the same vertex. This hierarchy is coupled with an element tagging strategy that permits to manage in a straightforward manner the partial knowledge of geometry and topology (a common situation when only a small portion of the whole mesh is loaded in each instant of time).

A small mesh portion is assigned to each *OEMM* leaf, based on regular hierarchical decomposition. Only the hierarchical structure of the octree is maintained in main memory: each octree leaf holds the external memory address of the corresponding portion of the mesh. An important feature of the *OEMM* is that it maintains a globally indexed representation of the mesh. Therefore, each vertex is uniquely identified by an integer and triangles are described and stored using just three indices (there is no vertex duplication). The vertex indices respect the octree structure and the order defined on the leaves in the following sense: each octree leaf node has associated a unique integer range, and all of its vertex indices lie in this range.

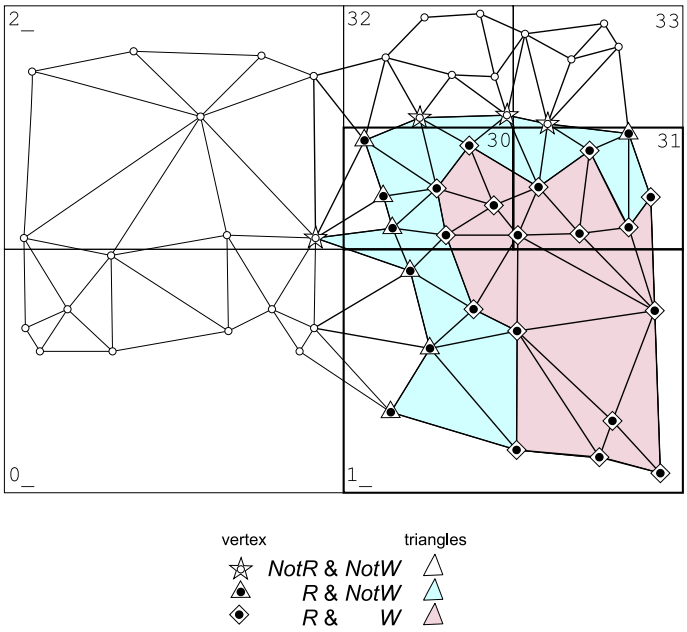


Fig. 1. Flags setting on a section of the mesh currently loaded in main memory (leaf nodes 1, 30, 31 are the ones loaded). Node numbering reflects the lexical order of the nodes.

Definition III.1: **OEMM leaf node.** Each leaf ℓ of the octree stores a pointer to a secondary memory chunk which contains:
• **vertices** - all the vertices **contained** in the bounding box of ℓ ; for each vertex v we also store the indices of the *OEMM* leaf nodes which contain shared faces incident in v;
• **faces** - for each triangle t partially contained in the bounding box of ℓ, t is stored in ℓ only if ℓ is the minimal leaf (according to the lexicographic order) which contains a vertex of t. Therefore, all the triangles completely contained in the bounding box of ℓ are stored in ℓ. In other words, a face is stored in the lowest index leaf that contains a vertex of the triangle.

Maintaining the whole octree structure in main memory is not a memory bottleneck because its memory size is not very large, even for very large meshes. To give an example, if we have an average of 16K triangles in each octree leaf, then the octree structure associated with a 10G faces mesh requires ≈40 MB.
The data structure encoding each *OEMM* node on disk is

as follows:

```
OctreeNode{
  OctreeNode *parent;
  OctreeNode *child[8];
  EM\_Pointer Mesh; // Pointer to external memory
  int vn;           // Vertex number
  int tn;           // Triangle number
  int BaseInd;      // index range of leaf goes from
  int LastInd;      // BaseInd to LastInd
  vector<int> L;    // Set of adjacent leaves with shared data
}
DiskTriangle{
 int v[3];
 data attributes;  // User-defined attributes
}
DiskVertex{
 Coord3d p;
 unsigned char ol[8]; // Indices of adjacent cells containing
                  // triangles incident in vertex p
 bool deleted;
 bool modified;
 data attributes;  // User-defined attributes (color,
                  // quadrics, etc.)
 }
```

This gives a minimal representation of a mesh; if needed, more complex representations (e.g. with explicit topology links between adjacent faces) can be built on the fly at data loading time.

The main purpose of this structure is to allow the user to load in main memory and to modify any small contiguous portion of the mesh, independently of the underlining hierarchical decomposition. By traversing the *OEMM* octree structure and iteratively loading, updating and saving leaves we are able to apply on very large meshes almost any kind of geometric algorithm based on local updates. Moreover, some geometric algorithms that need to work on the whole mesh can be redesigned such that just a portion of the data should be needed at each instant of time.

Loading just a portion of the mesh force us to cope with partial knowledge of the mesh elements. As an example, a vertex on the frontier of the mesh section assigned to the current leaf may have incident faces which are not contained in the current leaf, or some of the faces on the frontier can be defined by vertices whose geometry has not been loaded because it is stored in an adjacent, non-loaded cell. The `ol` field contained in the *vertex* data structure encodes the set of adjacent leaves which contain faces incident in that vertex. To ensure space efficiency, the `ol` has been implemented as a fixed length `unsigned char` field; the values contained in `ol` are indices to a list of adjacent cells stored in the corresponding leaf node, that is the vector `L` of leaf indices (see data structure above). If more than 8 indices are required, they are allocated in a dynamic list (but this situation never arose in all the tests presented). Note that it can be proved that if triangles edges are smaller than half of the smallest octree box, then each vertex can be referenced by, at most, seven other leaves.

Because we load only a portion of the mesh, we must maintain explicit information on which operations can be performed on the currently loaded or referred mesh elements. For this reason a set of flags are added to the `Vertex` and `Triangle` data structure when data are loaded in RAM:

```
Triangle{
vertex* v[3];
int flags;
}
Vertex{
Coord3d p;
int OEMMVertIndex;
int flags;
data attributes;  // User-defined attributes
                  // (color, quadrics, etc.)
}
```

The **vertex flags** hold the following values:
• **readable** and **writable:** a vertex is *readable* if it is contained in one of the currently loaded leaves. A vertex is *writable* if all of the faces incident in it are contained in leaves currently loaded. In this way, a vertex that is referenced by some non-loaded triangle is set *readable* but *non writable*, preventing modifications. On the contrary, if a vertex is not loaded and is referenced by triangles that are loaded then it is tagged as *non readable* and *non writable*. We implicitly assume that *writable* implies *readable*;
• **modified:** a vertex is *modified* when either its coordinates or the set of elements incident in it have been modified or, in some sense, processed (for example, to prevent multiple redundant processing on the same mesh element).

Conversely, the **face flags** hold the following values:
• **readable** and **writable:** a triangle is *readable* if it is contained in one of the currently loaded leaves. A triangle is *writable* if all of the vertex-adjacent triangles are *readable*, or in other words if all its vertices are *writable*;
• **modified:** a triangle is *modified* when its vertex indices have been modified.
An example of flags settings is shown in Figure 1. Flags are initialized by the loading function of the *OEMM* leaves, according to: (a) the values of the `ol` and `L` fields in the *OEMM* representation (see `OctreeNode` and `DiskVertex` data structures), and (b) the current set of leaf nodes loaded.

IV. Building the *OEMM*

We assume that the input mesh comes as a large set of raw, not indexed triangles, stored therefore with just 3D coordinates. We therefore describe OEMM construction considering the worst-case input (if we have in input an indexed mesh, some construction steps described below can be avoided or simplified). In any case, many huge meshes comes as a set of independent indexed meshes (e.g. produced by separate runs of a surface fitting code), and therefore re-indexing them in a common vertex space is needed. The *OEMM* is constructed in two steps:
1. a *raw OEMM* structure is built in secondary memory by processing all input triangles; the *raw OEMM* is a non-indexed *OEMM* . Each *raw OEMM* leaf node ℓ contains *all* triangles $\{t_i\}$ such that at least one of the vertices of t_i is contained in the node bounding box B_ℓ. Note that triangles shared by multiple leaf nodes are replicated in all those nodes of the raw *OEMM* ;
2. the raw *OEMM* is traversed and an *indexed OEMM* is built, i.e. an octree where triangles are indexed us-

ing a global vertex naming strategy. At the end of this phase, vertices and faces of the mesh are partitioned on the *OEMM* leaves according to Definition 3.1, with no redundancy.
In the following paragraph we see some details on how this building process is performed.

A. Building a raw OEMM

This first construction phase is performed in two steps. The goal of the first step is to determine the structure of the *OEMM* octree: we fix a maximal depth of the *OEMM* , we scan all the triangles and count, for each leaf node ℓ of the *OEMM* , how many triangles should be assigned to it. When all faces have been virtually assigned to leaves, sibling leaf nodes are collapsed into the parent node if and only if: the sum of the triangles contained is lower than a user-selected threshold, called *max_triangles*, **and** the resulting merged node has adjacent nodes whose depth in the tree differs from the depth of the current one by no more than three levels (i.e. we build a *restricted* octree [25]). The second condition guarantees that loading a leaf and all of its adjacent leaf nodes has a bounded space complexity.

In the second step we read again the set of raw triangles from secondary storage, and distribute them in the secondary memory buckets corresponding to the octree leaf nodes.

B. Building an indexed OEMM

To build the *indexed OEMM* we perform two complete traversal of the intermediate data structure: firstly, we traverse the raw *OEMM* to build an intermediate indexed *OEMM* where only *internal vertices* are correctly indexed (we call internal the vertices *contained* in the leaf bounding box, and external the others); then, the final indexed *OEMM* is built by indexing also the *external vertices* which belong to the faces shared by the adjacent leaf nodes.

Indexing internal vertices. The indices of the vertices should respect the lexicographic order of the leaves of the *OEMM* . Therefore, the leaves of the *raw OEMM* are read from secondary storage in lexicographic order, and for each leaf ℓ we assign an unique index to each vertex contained in the given leaf, and copy them in the *indexed OEMM*. All the vertices that are not contained in ℓ are indexed with a temporary fake value.
In this step we also setup the per-node and per-vertex list of *OEMM* leaves that contain faces shared with the current leaf node. This can be done easily due to the redundant representation of shared faces in the *raw OEMM*.

Indexing external vertices. The last step computes a correct global index for all the external vertices of the shared faces represented in each leaf. Therefore, each *OEMM* leaf node, with the adjacent ones, is read from secondary memory for the last time, and the global indices assigned to the internal vertices of a cell are propagated to the adjacent ones containing shared triangles as follows:

- all the leaf nodes ℓ_i which share triangles with ℓ are loaded;
- for each vertex $v \notin \ell$ of a shared triangle $t \in \ell$, we replace the fake index initially assigned to v in ℓ with the correct index assigned to v in the leaf node ℓ_j containing v.

V. Working with the OEMM

Working with the *OEMM* involves the iterative application of load/[modify/save] actions onto the *OEMM* leaves. Here we describe the details of these steps.

A. Traversal

In order to apply a geometric algorithm over an *OEMM* we have to define a visiting strategy such that all the vertices and triangles are seen at least once as readable and writable. Loading only a leaf at a time does not allow to get full information on the associated mesh portion and to modify the triangles which are not completely contained in the current leaf. The *OEMM* library implements different atomic data access rules:

- *subtree*: load all the leaves contained in the subtree plus all the leaf nodes adjacent to the nodes of this subtree;
- *bounding-box*: load the minimal set of leaves such that all the vertices contained in the given bounding-box and all the triangles referencing them are loaded.

A geometric algorithm can traverse the *OEMM* choosing any of the previous atomic rules depending on the characteristics of the processing to be performed and on the relative space requirements.

B. Loading Leaves

Loading in main memory a generic set of leaves $S = \{\ell_0, .., \ell_k\}$ means to reconstruct a standard indexed mesh representation from the *OEMM* loaded leaf nodes. This task involves the re-indexing of the mesh faces to a new vertex vector composed only by the loaded vertices (i.e. a vertex vector much smaller than the global *OEMM* vertex list); and to assign the correct flags settings to all faces and vertices.

Vertices re-indexing can be done in linear time because the maximum number of adjacent nodes is bounded by a constant. The original index of each vertex is maintained (see the `int OEMMVertIndex` of the `Vertex` data structure in Section III), in order to guarantee that *non writable* vertices could be placed back in the original position of the corresponding leaf block on secondary memory (see Section V-C).

The flag values (*readable/writable* and *modified*) are assigned as follows (see also Figure 1):

- vertices referenced by triangles outside all $\ell_i \in S$ are tagged *not writable*.
- vertices stored in non-loaded leaves **but** referenced by triangles in $\ell_i \in S$ are replaced with dummy vertices and tagged *not readable, non writable*.

C. Saving Leaves

A modified mesh corresponding to a set of leaf nodes $S = \{\ell_0, .., \ell_k\}$ has to be written back on secondary memory to make these modifications permanent. This step involves a back conversion of the current indexed mesh into a *OEMM* mesh chunk indexed with the global *OEMM* indices. We distribute the vertices to the appropriate *OEMM* leaves, and implicitly assign to each vertex the global index. During the saving step it is important that each vertex referenced by *non loaded* triangles (i.e. the ones that we classify *non-writable*) keeps its original position in the vertex list of the *OEMM* leaf node (the global index of each vertex is implicitly coded with the range of the leaf plus the vertex position inside the leaf).

Then, we distribute triangles to the appropriate *OEMM* leaf; for triangles shared by multiple leaves, the selection is performed by looking at the global index of the vertices, according to definition in III.1. Finally, for each face the indices of its vertices are replaced with the corresponding new global indices.

To ensure correctness of loaded nodes saving back, the following situations must be detected:

- *vertex indices out of range:* if the number of vertices to be saved back in a *OEMM* leaf is bigger than the original leaf range (for example because we updated the leaf to triangulate some mesh holes), then the leaf range should be expanded. Because assigning a wider range to a leaf is a costly operation (involving loading and re-indexing multiple leaves), at *OEMM* creation time we have distributed the leaf ranges uniformly over the 32 bit integer space. In this manner, there is plenty of space between any pair of consecutive leaf range to slightly widen the range. Obviously, if leaf nodes size changes in a drastic manner, an update to the *OEMM* structure could be needed (see Section V-D);
- *vertex coordinates not contained in the current loaded space:* a dangerous situation is when the coordinates of a modified vertex are not contained in the space corresponding to the loaded *OEMM* section (i.e. the union of the bounding box of the loaded leaf nodes). To prevent this situation, we detect every update which modifies the mesh by moving vertices in regions that are still not loaded, abort this update and backtrack.

D. Modifying the OEMM structure

The *OEMM* structure can be dynamically updated due to multiple delete/creation actions operated on the loaded nodes.

Node Merging. Every time a leaf is saved back, we firstly check if it can be collapsed with its siblings nodes in the corresponding parent node. When the number of vertices and triangles of the eight siblings is lower than a given threshold and all the conditions specified in Section IV-A hold, we can merge them in a single leaf. Node merging is as follows: the eight leaves and all other *OEMM* nodes referencing their vertices are loaded; the new range of the vertex indices assigned to the new leaf is computed; vertices are re-indexed; all the triangles of the loaded leaves are remapped with the new vertex indexing (this can involve the updating of some `ol` lists of the adjacent nodes); finally, all the loaded nodes are saved back.

The merging process is executed frequently during external memory mesh simplification (see Section VI).

Node Splitting. Node splitting is the inverse of the previous operation, and it has to be performed when the number of element in a leaf is higher that the maximum leaf size. Again, we have to reindex the vertices of the split sections and to reflect the new vertex indexing on the sibling nodes.

E. OEMM Complexity

While from a theoretic point of view octree's have not a good worst case complexity, they perform really well in practice. Let us assume that the input mesh has some *reasonable* characteristics: the number of triangles incident in a single vertex is bounded by a constant; the size of the faces is not smaller than a minimal value, and therefore the maximal depth of the octree is bounded. Then, we can assert that: loading and saving a leaf node (and some of the adjacent ones) has a cost linear in the size of the mesh elements contained in the loaded/saved nodes.

VI. External Memory Mesh Simplification

Given a triangular mesh we want to reduce its size by adopting a high-quality incremental approach, e.g. based on the iterative collapse of its edges [11]. Locality of the simplification method is a must, to allow us to load and process the mesh one piece at a time. In particular, a Quadric Error Metrics (QEM) method has been implemented in our system, and is described in the next subsection. Each edge collapse has an error-cost that has to be evaluated for each candidate edge, both at initial time and during the simplification process (every time the given edge is adjacent to some modified mesh component). We assume here that the error-cost can be computed in constant time and that requires a *per-vertex* constant space occupation (i.e. it requires only to access a local neighborhood of the collapsed edge). This last assumption is true for the error estimation techniques used in [27], [11], [19]. At each step of the simplification process the edge with the minimal error cost is collapsed (a heap is used to support ordered selection) and the error evaluation of the adjacent edges is updated. The overall worst case complexity of such an algorithm is $O(v \log v)$, with v the number of vertices.

A. Quadric Error Simplification in the OEMM framework

Quadrics are included in the *OEMM* vertex attribute and used to evaluate edge collapse error. As far concerns quadrics management, there are mainly two approaches: storing and updating quadric errors during edge collapse (see Garland and Heckbert [11]) or re-computing quadrics on the fly as proposed in the *memoryless* approach [19]. To describe how do we manage quadrics, we have to distinguish between what we store on disk, and what we store in RAM. Both our RAM-QEM and OEMM-QEM use the approach of Garland-Heckbert (quadrics are saved and updated during the simplification of the currently loaded section of the mesh). In the external-memory implementation

(OEMM-QEM), when the simplification of the current section is terminated we write back on disk just the mesh (and discard the quadrics). Therefore, when the same leaf is loaded again and simplified further, we will start from a set of newly initialized quadrics. Our experience showed that retaining quadrics on RAM (during simplification of a mesh portion) can be worthwhile, while it is not worthwhile to retain them also among different simplification passes over the same section (due to limited impact on accuracy and the substantial overhead on data loading/writing) and it helps to avoid the *quadric lock* problem [14].

Simplification algorithms usually adopt a priority queue to choose the next edge to be collapsed; for this reason they access the mesh with an order that is inherently non-local. This scattering behavior causes *virtual memory trashing*, making any approach based on standard virtual memory features totally inefficient. Instead of forcing the algorithm to follow the exact edge collapse order, as done for example by El-Sana et al. [7], we choose to slightly change the collapse order in order to catch geometric locality. Therefore we do not keep a global heap with all the possible collapses, but we traverse the *OEMM* (following the lexical order of the leaves) and for each subtree that we load we build a local priority queue and simplify it separately. We have verified empirically that this *local sorting* has a very little influence over the quality of the resulting simplification.

For each loaded subtree, we also load all the adjacent leaves of the *OEMM* . This ensures that all the possible edges of the current subtree (including the ones on the boundary of the subtree) are evaluated for a possible collapse. Therefore, at the end of the traversal the mesh is uniformly simplified (while other hierarchical approaches are constrained to leave untouched the inter-cell boundaries [13], [22], [8]). Let ε be the maximum quadric error the mesh should satisfy, we produce a small sequence of errors (ε_1, .., ε_n=ε) built using a logarithmic increasing rule and iterate QEM simplification n times on the mesh. At each iteration i, we visit all the *OEMM* leaf nodes following the *subtree* traversal rule (see Section V-A); QEM is run on each mesh portion as long as accuracy ε_i is satisfied. During QEM run, all the edges that are incident in *writable* triangles are evaluated for collapse, and the corresponding forecasted error is stored in the heap. The use of the *readable* and *writable* flags is defined easily. We can collapse an edge only if all the vertices connected by an edge to any of the edge's vertices are *writable* and all the vertices connected with an edge with these ones are *readable*. This because for the collapse of an edge we need to *modify* (alias writable permission) the vertices at topological distance 1, and to know the value (alias readable permission) of the vertices at topological distance 2 (because we need to know their data to evaluate the new approximation error of all vertices at topological distance 1).
When we have reached error ε_i on a given *OEMM* mesh portion, we check during the leaf saving procedure (described in Subsection V-C) if it is possible to merge any modified leaf with the siblings leaves, and then we proceed with the next mesh portion. When the user requests a drastic simplification, the final *OEMM* can be composed of one or a few nodes. The traversing scheme ensures that all the edges whose edge stars span on adjacent *OEMM* nodes are considered for collapse at least once in each iteration.

A special case has to be considered, that is the case of edges whose extremes are not contained in two adjacent *OEMM* nodes. This situation is not common in the case of 3D scanned dataset (where data resolution is sufficiently regular), but can occur on CAD data or on irregularly shaped meshes where very *long* or *wide* faces might have vertices contained in non-adjacent nodes. In our approach these faces (spanning non-adjacent *OEMM* cells) are simplified only when, after some simplification steps, they become part of adjacent leaf nodes. Because siblings leaf nodes will be automatically merged during simplification, after a number of steps any "long/wide" face will become either contained in a single leaf node or shared by adjacent leaf nodes. One can object that in this manner the order of simplification of these faces is altered with respect to the standard error-driven order of an *in-core* simplifier. This is true, but we should say that normal meshes contain in general just a few of these "critical" faces (not hundreds or thousands), at least if the data producer has used a solid modeler in a conscious way. Under this assumption and because of their relative size and small number, postponing simplification will not have a drastic effect on the output mesh size/accuracy.

The simplification of mesh topology is needed by many applications, especially when the input data are very complex assemblies. Extending our external memory simplifier to support topology simplification could be easy. Following the approach proposed in [11], given the set of loaded *OEMM* leaf nodes we should only build a uniform grid on the corresponding mesh vertices. This grid supports an efficient detection of the pairs of non-adjacent but close vertices which have to be evaluated for collapse.

B. Detail preservation via resampled textures

Preservation of detail is a must on big meshes, especially if we want to process data with a very complex surface texture (see for example Figure 4) or a complex pictorial detail. In this case, the solutions that evaluate in an integrated manner the approximation of both the shape and some other scalar/vetorial field are in general not adequate, at least if we want to obtain a drastic mesh simplification. Preservation of mesh attributes can be managed as a post-processing phase: a texture can be resampled from the original mesh, containing a discretized representation of the detail removed during simplification (color, high-frequency surface perturbations, other scalar/vectorial fields, etc.) [5]. The resampled texture map is then used at rendering time to paint the detail of original high resolution mesh onto the simplified one [5], [26]. This solution is independent of the simplification process and thus we can simplify the mesh by considering only the shape attribute, leading to very high compression ratios.

The *external memory* implementation of the detail preserving approach is very easy on the *OEMM* framework.

Given a simplified mesh S, we distribute S in an *OEMM* octree having the same structure of the original input mesh *OEMM* . Then the two *OEMM* are traversed in parallel, each face of S is sampled by considering the corresponding mesh section of the original mesh (which is currently loaded in RAM) and the corresponding texture chunk is built. Write back of *OEMM* leaf nodes of the original mesh is not needed, because the data encoded in the *OEMM* is not modified during this phase.

An example of a resampled bump-texture mapped on a very simple mesh obtained by simplification is shown in Figure 7.

VII. Other EM mesh processing tasks

The other tools implemented on top of the *OEMM* representation are described briefly here, for the sake of conciseness.

A *mesh editing* tool has been defined, that allows the user to perform many editing actions which are crucial in a number of applications, e.g. 3D scanning and rapid prototyping. The editing operators provided include: topological check of the mesh, detection of non-manifold components, detection of holes, automatic or user assisted hole-triangulation, elimination on request of complex vertices/faces and small components. Implementing these mesh editing operations on the *OEMM* representation scheme is straightforward.

Obviously, visualization is an important task for the evaluation and the inspection of a mesh. A snapshot of the main window of an external memory visualization session is shown in Figure 2. Implementing an *external memory visualizer* is straightforward, because we only have to define an interface which allows the user to select the *OEMM* leaf nodes to be visualized. The visualization features provided in our prototypal system allow to: visualize a huge mesh by showing the bounding box of all mesh portions contained in the *OEMM* leaf nodes; selective visualization of the mesh sections corresponding to some *OEMM* leaf nodes; color-enhanced visualization of mesh components, to differentiate different topological classes of elements (e.g. for easy visualization of the holes or of the complex vertices detected by the mesh editing module); interactive picking of mesh components; etc. The main goal of this tool is not the pure presentation of the data (which could be implemented also by adopting a point-based approach [24], [21]), but the inspection of the geo-topological characteristics of a given high-resolution mesh (e.g. to evaluate its quality and, in case, to apply editing actions).

VIII. Results

Among the *external memory* algorithms presented, the most complex is the mesh simplification one. We report here the results relative to the simplification of four meshes, all of them obtained by 3D scanning and available at the Stanford 3D Scanning Repository (http://www-graphics.stanford.edu/data/3Dscanrep/):

- the *Happy Buddha* mesh (543,652 vertices, 1,087,716 triangles);
- the *S. Matthew* complete model (186,984,410 vertices and 372,767,445 triangles), representing one of Michelangelo's unfinished statues scanned by the Digital Michelangelo Project [16];
- two *David* models reconstructed at 1mm and 2mm accuracy (respectively: 28,184,526 v. 56,230,343 tr., and 4,128,614 v. 8,254,150 tr.), also scanned by the Digital Michelangelo Project.

Input Data	Simplification			
	quadric error	size (tr.)	time	t/sec
S.Matthew	0 → 1e-5	94,116,116	10:57:37	6.8K
	1e-5 → 1e-3	25,280,206	2:30:54	7.4K
	1e-3 → 1e-1	6,138,792	0:37:05	8.4K
	1e-1 → 1	3,119,222	0:07:29	6.5K
	1 → 10	1,638,646	0:03:21	7.1K
	10 → 100	788,202	0:01:29	9.3K
David 1mm	0 → 1e-2	13,525,698	1:02:24	10.8K
	1e-2 → 1e-1	7,565,958	0:12:31	7.7K
	1e-1 → 1	3,682,158	0:06:32	9.6K
	1 → 10	1,723,895	0:03:07	10.2K
David 2mm	0 → 1	2,517,234	0:07:30	12.5K
	1 → 10	1,413,304	0:01:31	11.9K
	10 → 100	739,485	0:00:52	12.6K

TABLE II

Results obtained in the simplification of the sample meshes. Times are in *hh:mm:ss* (I/O times included). The simplification rate is shown in the last colume (t/sec: simplified triangles per second). The RAM used is around 80 MB.

We did not considered typical CAD datasets. Even if very complex datasets are common in CAD applications, they are usually modeled as a composition (either hierarchical or linear) of medium-sized components, which can often be simplified and managed independently using standard in-core techniques.

Some numerical data on *OEMM* construction and mesh simplification are presented in Tables I and II. The computer used for the tests is a PentiumIII 800 MHz, 256 MB RAM, 30 GB disk running MS WinNT.

The size of the *OEMM* representation (in MB) and the time for the data conversion (from triangle soup to *OEMM*) are shown in Table I. As far concerns the size of the octree, we report here some figures relative to the most complex dataset used, the S. Matthew mesh: the *OEMM* is composed of ≈130K nodes, including internal nodes and empty octree leaves, the triangle per leaf threshold is 16K and the maximum depth of the octree is 8. *OEMM* construction takes a time which is approximately equal to mesh simplification time, and thus rather long. But *OEMM* construction is a data preprocessing phase executed only once, in the framework of the standard pipeline for processing a complex scanned mesh: *OEMM* construction, mesh editing (fixing topology, closing holes, smoothing, etc), mesh simplification. The cost of the conversion process is counterbalanced by the locality of the typical geometric computations (e.g. editing or simplification), which become more efficient on the *OEMM* structure and, obviously, require a small memory footprint. As an example, the simplification

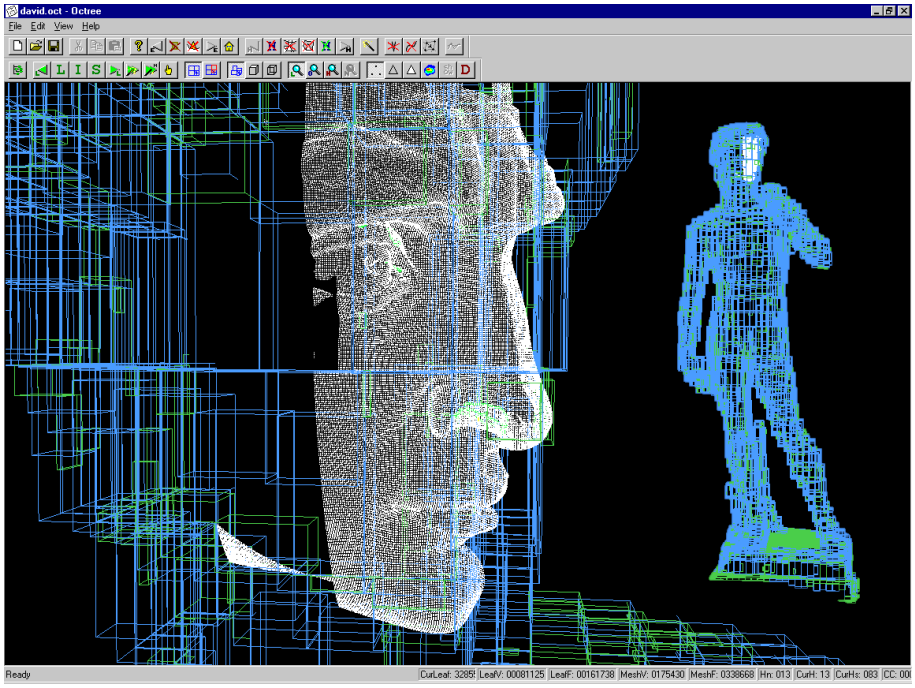
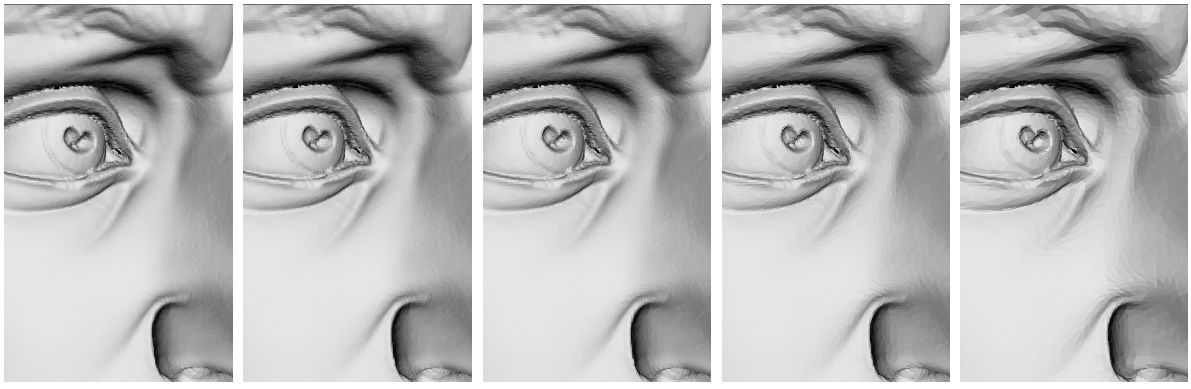


Fig. 2. A snapshot of the main window of the *external memory* visualizer; the loaded mesh section is rendered wireframe, the other *OEMM* leaf nodes are represented by wire-frame bounding boxes.

Input Data			OEMM Repr.			
	mesh size		raw *OEMM*	index.*OEMM*	raw *OEMM*	index.*OEMM*
name	triangles	size	size	size	build time	build time
S.Matthew	372,767,445	7.29 GB	12.5 GB	11.94 GB	2:52:35	8:28:07
David 1mm	56,230,343	1.10 GB	1.85 GB	1.77 GB	0:24:23	1:02:24
David 2mm	8,254,150	166 MB	283 MB	268 MB	0:03:13	0:07:20

TABLE I

THE TABLE REPORTS THE SIZE OF THE TREE SAMPLE MESHES AND OF THE CORRESPONDING *OEMM* REPRESENTATION. TIMES ARE IN *hh:mm:ss* (I/O TIMES INCLUDED).



original data, ~161K tr. (rendered section) | quadr.err.=1e-2, ~39K tr. | quadr.err.=1e-1, ~21K tr. | quadr.err.=1, ~10K tr. | quadr.err.=10, ~6K tr.

Fig. 3. A comparison of the different quality of some simplified David models (1mm David mesh, 53.6 M faces) using the *OEMM* quadric simplification.

of the David and S. Matthew meshes can be performed by using only 80 MB of core memory.

The use of an out-of-core approach introduces some overhead when compared to a standard simplification code working in main memory. We measured empirically the figures of our *OEMM* -based external memory simplifier (OEMM-QEM) with the ones of other codes working in core memory: QSlim v.2, the original QEM implementa-

tion due to M. Garland [12]; RAM-QEM, that is our implementation of the QEM method, running in main memory; and finally our implementation of the OutOfCore Clustering (OOCC) simplifier [17]. Results relative to the Happy Buddha mesh are presented in Table III.

Moreover, one could be interested to know how a standard edge-collapse would perform using just the OS paging mechanism. We run the two *in-core* solutions, QSlim and RAM-QEM, and the *external-memory* OEMM-QEM on a PIII 800 MHz PC with just 128MB of RAM (where no more than 80-90MB are available for user processes); Table IV presents the corresponding running times and global amount of virtual memory[1] (MEM) asked by the process to the OS. The exploding increase of running times when the system starts trashing is evident.

OEMM-QEM and RAM-QEM are based on the same simplification kernel, that is the classical quadric simplification error metric with the addition of weighted factors which take into account the variation of surface normals and the triangle aspect ratio. The difference between OEMM-QEM and RAM-QEM is in the different data traversal and heap management: OEMM-QEM traverses the mesh following the OEMM lexical order, and adopts local heaps to simplify the loaded mesh sections; on the other hand, RAM-QEM uses (analogously to QSlim) a classical global heap and needs to load in memory all of the mesh to initialize the heap and to run simplification. In most cases OEMM-QEM and RAM-QEM produce results (quality and speed) analogous to the ones of QSlim v.2; in some cases, weighting the normal variation improves results accuracy (this is mainly evident in the proximity of discontinue features). Our implementation of the OOCC was as conforming as possible to its original description, including the robust quadric inversion technique described in [17].

The OEMM-QEM consumes around 50% more secondary memory than standard QEM solutions (but secondary memory is nowadays quite an inexpensive resource), but requires a smaller core memory footprint. In any case, consider that the size of the on-disk *OEMM* representation is smaller than the core memory required by an in-core QEM. Therefore, if the core memory is sufficiently large to allow an in-core simplification, it is also sufficiently large to permit the operating system to cache the *OEMM* file in RAM. This explains partially the unexpected results of Table III, where OEMM-QEM simplification time is shorter than the RAM-QEM implementation. Moreover, times are shorter because: the OEMM-QEM local heaps are smaller than the global one used by RAM-QEM (heap construction has complexity $O(n \log n)$); processor cache misses are probably less frequent in the case of OEMM-QEM, because data structure access is more local than that of RAM-QEM.

On larger meshes, the need to perform multiple passes on the dataset (to improve the quality of the simplified mesh, as in the runs reported in Table II) would require multiple loading of the intermediate *OEMM* representations from secondary memory, introducing some overhead with respect to an ideal in-core solution. In fact, the simplification rates reported in the Tables II and III degrade gracefully with the increase of the size of the input mesh. The accuracy of the simplified meshes has been evaluated by using the *Metro* tool [6]. The RMS error (measured as a percentage of the mesh bounding box) is shown in the rightmost columnn of Table III. It is worth to note that *OEMM-QEM* accuracy is slightly lower than our in-core *RAM-QEM*, but at the same time it is still slightly better than the one of *Q-Slim* and obviously much better than *OOCC*.

Happy Buddha (1,087,716 faces)					
	simpl. faces	RAM	time (sec.)	t/sec rate	RMS err
QSlim v.2.0	18,338	195 MB	60	17.4K	0.0131%
RAM-QEM	18,338	160 MB	58	18K	0.0125%
OEMM-QEM					
build (pre-proc)	–	4 MB	58	–	–
simplify	18,338	60 MB	48	21.7K	0.0129%
OOCC	19,071	36 MB	15	69.5K	0.0245%

TABLE III

RESULTS OBTAINED IN THE SIMPLIFICATION OF THE HAPPY BUDDHA MESH, USING FOUR DIFFERENT SIMPLIFICATION CODES.

The simplified meshes produced are shown in Figures 3, 4, 5, and 6.

We performed an empirical comparison with the Out-Of-Core Clustering approach (*OOCC*) [17]. The times of the *OOCC* solution are obviously impressive (see the simplification rate in Table III). On the other hand, the quality of the mesh produced is directly dependent of the regular sub-sampling operated on the mesh (to reach a drastic simplification of a 3D scanned mesh the cluster cell size is generally set much larger than the mean face size). The higher accuracy of the results produced by *OEMM-QEM* is shown in the images presented in Figure 6. Moreover, the meshes produced by the Clustering approach are often non-manifold, and this may introduce problems when we have to apply geometric processing on the output mesh. For example, an *OOCC* run on the 2mm David mesh (from 8M triangles down to 235K) generates more than 21K non-manifold vertices.

One can ask if the improved accuracy of *OEMM-QEM* is worth the processing overhead (*OEMM-QEM* is approximately 3 times slower than *OOCC*). There are a number of applications where data accuracy is a must (visual inspection, rapid prototyping, shape recognition, 3D reconstruction from multiple fragments, etc). In all these cases, a slightly slower simplification time is not a problem: this process is executed only once, and in any case simplification time is a very small fraction of the time needed to produce the raw data (e.g. by 3D scanning) or to analyze it.

A comparison of the different visual accuracy provided by a plain simplified mesh or by the same mesh enhanced

[1]Please note that in Table IV we reported the *working set* used by the simplification process, while the effective maximum size of required RAM has been presented in Table III.

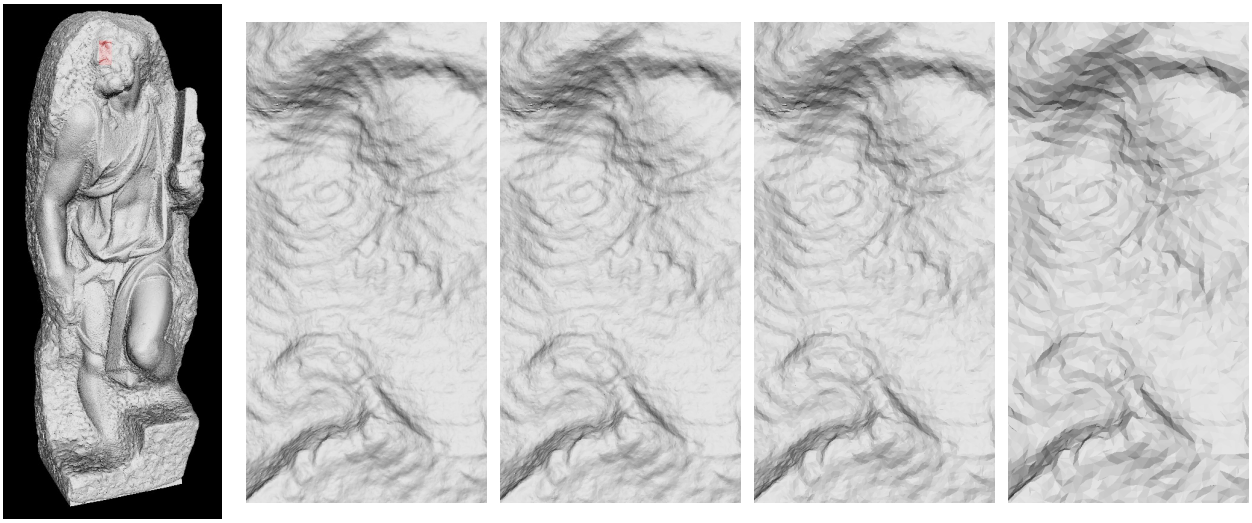


Fig. 4. A simplified model of the S. Matthew statue is shown on the left; a small section of the mesh (S. Matthew's eye and part of the nose) is shown on the right at different accuracies.

HappyBudda (various mesh sizes)										
Input faces:	339,344		408,090		511,138		593,544		746,834	
Output faces:	32,760									
	time (h:m:s)	MEM (MB)	time (h:m:s)	MEM (MB)	time (h:m:s)	MEM (MB)	time (h:m:s)	MEM (MB)	time (h:m:s)	MEM (MB)
QSlim v.2.0	0:00:15	76	0:00:18	90	0:01:46	115	3:17:28	200	n.a.	n.a.
RAM-QEM	0:00:14	94	0:00:16	94	0:00:49	130	0:00:55	180	0:37:19	200
OEMM-QEM	0:00:19	50	0:00:23	50	0:00:29	50	0:00:36	50	0:00:49	50

TABLE IV

The results presented show the poor performances of the in-core solutions (QSlim, RAM-QEM) when the external memory management is demanded to the standard OS paging system; the meshes used in input are simplified versions of the original Happy Buddha.

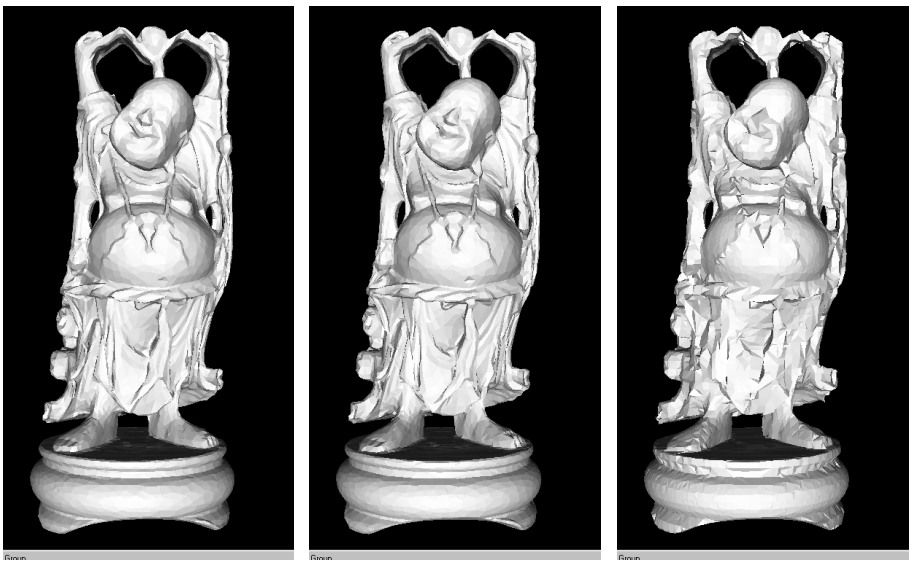
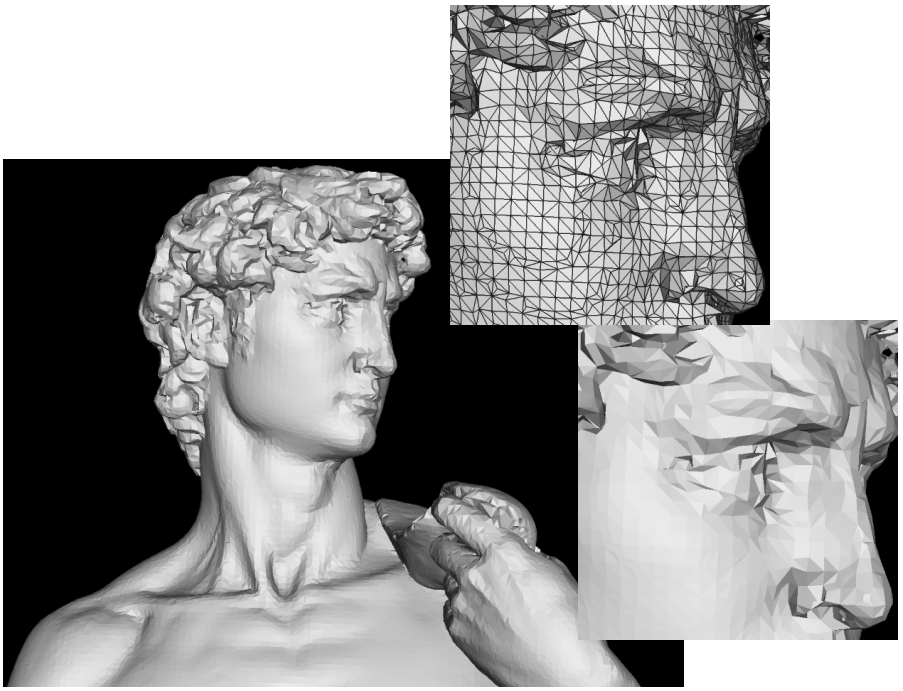
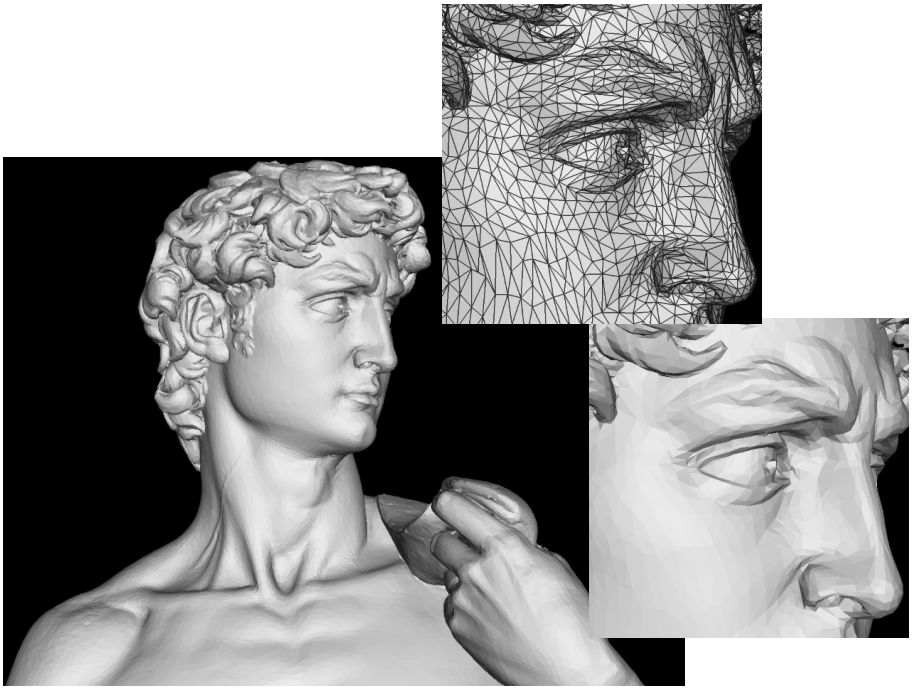


Fig. 5. Results of the simplification of the Happy Buddha mesh.



OOC Clustering, 235K faces



OEMM edge collapse, 215K faces

Fig. 6. A comparison of the different quality of the models obtained from the simplification of the 2mm David mesh (8M faces) using the OOC Clustering solution (top) and the *OEMM* simplifier (bottom).

with a resampled bump-map is shown in Figure 7. Notice how much the visual quality of the drastically simplified mesh (10K faces) is improved by the resampled bump-texture; it appears very similar to a more complex model (1,683K faces) presented in the same image on the left. Considering data size: the 1024*1024 RGB normal map size is 1.5MB, when compressed using PNG format and preserving image quality, and it is texture-mapped to the 10K faces model (size on disk 905KB in un-compressed binary format). This should be compared with the 1.6M

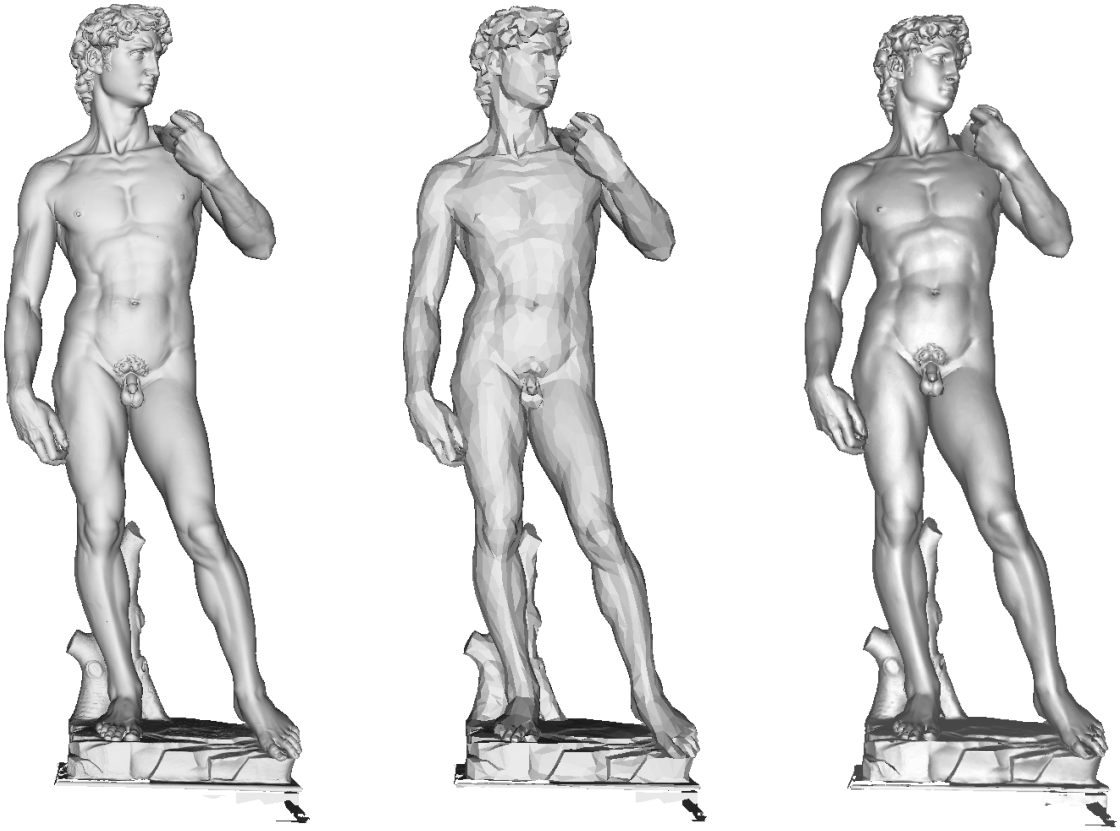


Fig. 7. A comparison of the different visual quality provided by: two simplified David meshes, 1,683K and 10K faces, and the latter enhanced by mapping a re-sampled bump texture (please note that the image with bump-mapping has been created with a different viewer).

faces mesh, which needs 36MB to be stored on disk.

IX. Concluding Remarks

We have demonstrated that even huge meshes can be successfully managed on a low cost architecture. The *OEMM*, an external memory data structure, is at the base of our mesh management and simplification system. It permits to implement in a memory-efficient manner all geometric algorithms that process the mesh via a local update approach, by decoupling mesh size from main memory size and dynamically loading portions of the dataset from secondary memory. The *OEMM* data structure implements an out-of-core global indexed representation on huge meshes, and loading/processing of portions of the data is easy thanks to: the *space subdivision* embedded in the octree representation, the *automatic re-indexing* of the loaded data sections, and the *tagging strategy* (readable/writable tags) that allows the easy detection and management of the elements located on the boundary of the current region. The system presented provides a valid solution for visual inspection, editing, and simplification of huge meshes. As an example, it permits to manage all the post-acquisition phases of the 3D scanning pipeline on a low cost machine. With an acceptable time overhead we can process meshes which cannot be managed on most other architectures. Managing the S. Matthew mesh with an in-core simplifier, for example, would require approximately more than 55GB of core memory. Moreover, an out-of-core solution usually requires a much smaller RAM size than the corresponding RAM-based solutions (in our system, the size of the surface sections loaded can be decided by the user). This appears clear in the results presented in Table III: the simplification of a medium complexity mesh (around 1M faces) works in only 60MB of RAM (or even on a smaller footprint, depending on the size of the loaded subtree selected by the user). Conversely, the RAM-based QSlim solution allocates 195MB to process the same mesh.

It should be noted that the choice of an octree as a partitioning scheme is *not* mandatory. Depending on the mesh processing tasks that have to be carried out, other mesh partitioning schemes can be chosen. For example, if we consider uniformly sampled meshes and tasks that do not drastically alter the size of the mesh (like smoothing filters or hole filling), the octree can be replaced by a simpler uniform grid partition. In this case the interface between the mesh processing algorithm and the *OEMM* remains the same, because it is based on a generic traversal process and the element tagging policy (read/write/modified tags) supported can be easily extended to other decomposition rules.

Possible extensions to the OEMM-based mesh management environment are as follows. We are adding more sophisticated visualization features, which should allow a naive user to navigate and inspect very complex dataset, e.g. meshes produced by 3D scanning Cultural Heritage artefacts, on low cost computers using an LOD approach. We are designing an external memory multiresolution representation, and finally we are planning to include on-the-fly mesh compression techniques to reduce the storage of the *OEMM* leaf nodes.

X. Acknowledgements

We would like to thank Marc Levoy and the Stanford Computer Graphics Group for providing scanned data, and for choosing our simplified meshes as the official simplified models distributed on the project's web.
We acknowledge the financial support of the Progetto "RIS+" of the Tuscany Regional Government and of the EU project IST-2000-28095 "The Virtual Planet".

References

[1] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *1999 Symposium on Interactive 3D Graphics*, pages 199–206, New York, Apr. 26–28 1999. ACM Press.

[2] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, Oct.-Dec. 1999.

[3] F. Bernardini, J. Mittleman, H. Rushmeier, and G. Taubin. Case study: Scanning Michelangelo's Florentine Pieta'. In *ACM SIGGRAPH 99 Course Notes, Course 8*, August 1999.

[4] Y. Chiang, C. T. Silva, and W.J. Schroeder. Interactive out-of-core isosurface extraction. In *IEEE Visualization '98 Proceedings*, pages 167–175. IEEE Press, 1998.

[5] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, and M. Tarini. Preserving attribute values on simplified meshes by re-sampling detail textures. *The Visual Computer*, 15(10):519–539, 1999. (preliminary results appeared in IEEE Visualization '98 Proceedings).

[6] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, June 1998.

[7] J. El-Sana and Y.-J. Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 19(3):139–150, August 2000.

[8] Carl M. Erikson. *Hierarchical Levels Of Detail To Accelerate The Rendering Of Large Static And Dynamic Polygonal Environments*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 2000.

[9] T.A. Funkhouser. *Database and Display Algorithms for Interactive Visualization of Architectural Models*. PhD thesis, CS Division, UC Berkeley, 1993.

[10] M. Garland. Multiresolution modeling: Survey & future opportunities. In *EUROGRAPHICS'99, State of the Art Report (STAR)*. Eurographics Association, Aire-la-Ville (CH), 1999.

[11] M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 209–216. Addison Wesley, August 1997.

[12] M. Garland and P.S. Heckbert. *QSlim v.2 Simplification Software*. School of Computer Sciences, Carnegie Mellon University, URL: http://www.cs.cmu.edu/ğarland/quadrics/qslim.html, 1999.

[13] H. Hoppe. Smooth view-dependent level-of-detail control and its aplications to terrain rendering. In *IEEE Visualization '98 Conf.*, pages 35–42, 1998.

[14] H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. In *Proceedings of the 10th Annual IEEE Conference on Visualization (VIS-99)*, pages pages 59–66, New York, October 25–28 1999. ACM Press.

[15] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In *Computer Graphics 25(4) (SIGGRAPH 91 Proceedings)*, pages 285–288, July 1991.

[16] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D scanning of large statues. In *SIGGRAPH 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 131–144. Addison Wesley, July 24-28 2000.

[17] P. Lindstrom. Out-of-core simplification of large polygonal models. In *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH 2000), ACM Press*, pages 259–262. Addison Wesley, July 22-28 2000.

[18] P. Lindstrom and C.T. Silva. A memory insensitive technique for large model simplification. In *Proc. IEEE Visualization 2001*, pages 121–126. IEEE Press, October 2001.

[19] P. Lindstrom and G. Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2), April 1999.

[20] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[21] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In Kurt Akeley, editor, *SIGGRAPH 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 335–342. ACM Press - Addison Wesley Longman, 2000.

[22] Chris Prince. Progressive meshes for large models of arbitrary topology. Master's thesis, Department of Computer Science and Engineering, University of Washington, Seattle, August 2000.

[23] J. Rossignac and P. Borrel. Multi-resolution 3D approximation for rendering complex scenes. In B. Falcidieno and T.L. Kunii, editors, *Geometric Modeling in Computer Graphics*, pages 455–465. Springer Verlag, 1993.

[24] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH 00)*, pages 343–352. ACM Press, July 24-28 2000.

[25] H. Samet. *The design and Analysis of Spatial Data Structures*. Addison Wesley, Reading, MA, 1990.

[26] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. In *SIGGRAPH 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 327–334. Addison Wesley, 2000.

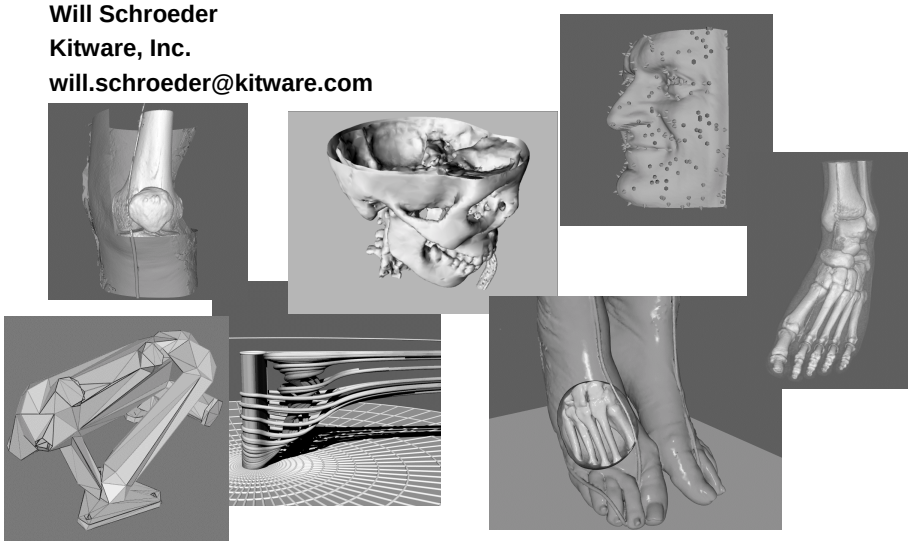
[27] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *ACM Computer Graphics (SIGGRAPH 92 Proceedings)*, volume 26, pages 65–70, July 1992.

[28] E. Shaffer and M. Garland. Efficient adaptive simplification of massive meshes. In *Proc. IEEE Visualization 2001*, pages 127–134. IEEE Press, October 2001.

[29] Shyh-Kuang Ueng, Christopher Sikorski, and Kwan-Liu Ma. Out-of-Core Streamline Visualization on Large Unstructured Meshes. *IEEE Tran. on Visualization & Computer Graphics*, 3(4):370–380, October 1997.

Parallel Processing with VTK

Will Schroeder
Kitware, Inc.
will.schroeder@kitware.com



Credits

Charles Law
Berk Geveci
Ken Martin
Amy Henderson
Sebastien Barre
Brad King
Andy Cedilnik

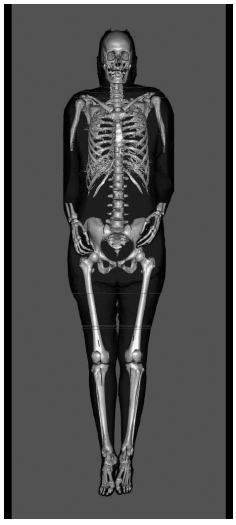
Jim Ahrens
Randy Frank
Mike Papka
Hank Childs
Gary Templet
Pat McCormick

Kitware, Inc. – *These notes taken from VTK Developer's Course*

What is the Problem?

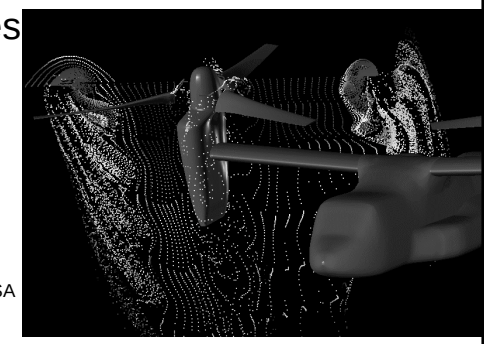
- Analysis codes are producing very large data sets
- Computing clusters are more affordable and widely used
- Have an extensive visualization toolkit with many existing algorithms. We want to effectively use all of these algorithms with large data and parallel visualization

Data Size



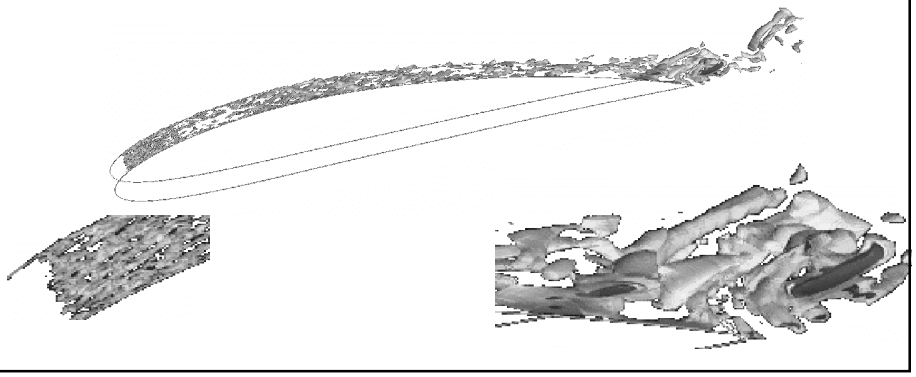
- Visible Woman CT Data
  870 MBytes 1734 Slices at 512x512x2
- Bell-Boeing V-2 2 tiltrotor
  140 Gbytes

Flow Computation:
Robert Meakin
Visualization:
David Kenwright and
David Lane
Numerical Aerodynamic
Simulation Division at NASA
Ames Research Center



Other Large Data

- Modeling turbulence (Ken Jansen RPI)
- 8.5 million tetrahedra, 200 time steps
- 150 million tetrahedra, 2000 time steps (soon)



What is the Solution?

- Expand the capabilities of VTK to take advantage of parallel and streaming visualization
- Use existing tools where they fit, e.g. MPI

Challenges

- Decomposing data into pieces
  - Streaming
  - Parallel processing
- Communicating among processes
  - Synchronization
  - Passing pieces back and forth
  - Communication

Challenges (cont.)

- How to define a piece of data?
- How to convert pieces between different data topologies?
- How to handle boundary conditions?
- How to make the filters operate on pieces?
- How to determine the number of pieces to use?
- How to support asynchronous execution?
- How to make it easy to use?

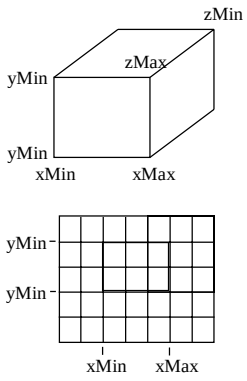
Note: Multi-Threading

- Some VTK Filters are multi-threaded
  - Imaging
  - Streamlines
  - Volume rendering
- Multi-Threading is NOT scalable
  - Shared memory
- Distributed parallel is preferred because it is scalable

Pieces

Imaging: i,j,k extents.

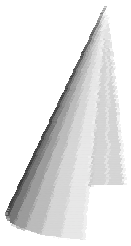
```
image->SetUpdateExtent(0, 128, 0, 128, 0, 90);
image->Update();
```



Graphics

```
data->SetUpdateExtent(0, 2);
data->Update();
```

Piece = 0
NumberOfPieces = 2



Defining a Piece of Data

- Extents work for regular implicit topologies but don't work for unstructured explicit topologies
- Define a piece by its world coordinate bounds, but that can be difficult to produce, define, and enforce
- Most general is to define a piece as a portion of the data (piece N out of M) with no assurances of topological or geometric coherence

Converting a Piece

How to convert a request between unstructured pieces and structured pieces?

- For an unstructured request to structured, use a consistent rule for mapping piece N to an (i_{min}, i_{max}, j_{min}, j_{max}, k_{min}, k_{max}) extent
- For a structured request to unstructured, most algorithms require all of the data, but it can be requested in pieces
- Streamlines are a challenge

Converting a Piece

- For some algorithms the atomic unit is a cell, for others it is a point
- Surface warping based on scalar or vector data is a point based operation, so is hedgehogs
- Streamlines, decimation, contouring are cell based operations
- Glyphing is both, the locations to glyph are point based operations but the glyph to use is cell based

Example

```
void vtkAppendPolyData::ComputeInputUpdateExtents(vtkDataObject *data)
{
  …
  output->GetUpdateExtent(piece, numPieces, ghostLevel);
 …
  if (this->ParallelStreaming)
    {
    piece = piece * this->NumberOfInputs;
    numPieces = numPieces * this->NumberOfInputs;
    }
 …
  // just copy the Update extent as default behavior.
  for (idx = 0; idx < this->NumberOfInputs; ++idx)
    {
    if (this->ParallelStreaming)
        {
        this->Inputs[idx]->SetUpdateExtent(piece + idx, numPieces, ghostLevel);
        }
    }
…
```

Example

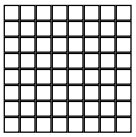
```
void vtkSphereSource::Execute()
{
…
  int piece = output->GetUpdatePiece();
  int numPieces = output->GetUpdateNumberOfPieces();
…
  // Change the ivars based on pieces.
  int start, end;
  start = piece * localThetaResolution / numPieces;
  end = (piece+1) * localThetaResolution / numPieces;
  localEndTheta = localStartTheta + (float)(end) * deltaTheta;
  localStartTheta = localStartTheta + (float)(start) * deltaTheta;
  localThetaResolution = end - start;
…
}
```


Handling Boundary Conditions

- Piece invariant results
- Many algorithms require boundary or neighborhood data to produce the correct results
- We use *ghost levels* to specify this
- Structured algorithms change the size of the update extent to meet their needs
- For unstructured data we add ghost levels

Handling Boundary Conditions

- A central difference gradient algorithm would request an additional pixel around its update extent



- Hard boundaries are due to a lack of data, they can be identified by looking at the whole extent
- Hard boundaries must be handled by the algorithm, either reduce the output size or renormalize, etc.

SC2002
Tutorial M9

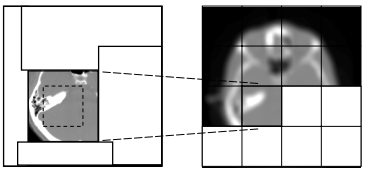
Handling Boundary Conditions

- For unstructured data we specify the number of ghost levels required
- Each filter can increment this number
- Computing surface normals for vertices requires one level of ghost cells

Result Invariance

Imaging

Smooth requires a larger input extent.



Graphics

Some filters require ghost cells.

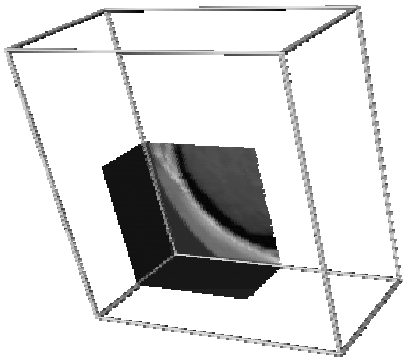
vtkSource::ComputeInputUpdateExtent(vtkDataSet *outdata);

Ghost Level

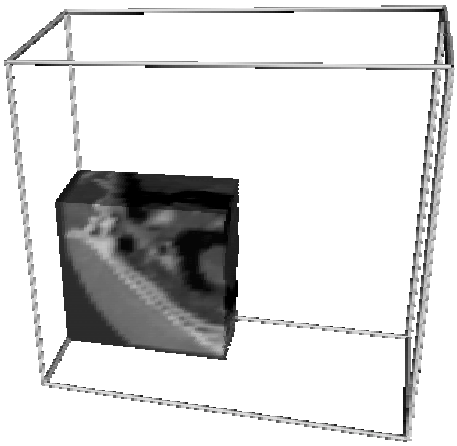
- Ghost cells are marked by a ghost level field array.
- Filters can strip away ghost cells when the are finished with them.

vtkGeometryFilter

With Ghost Cells

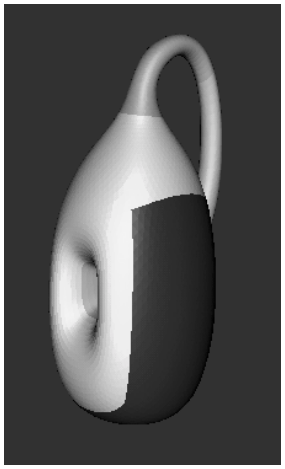


Without Ghost Cells

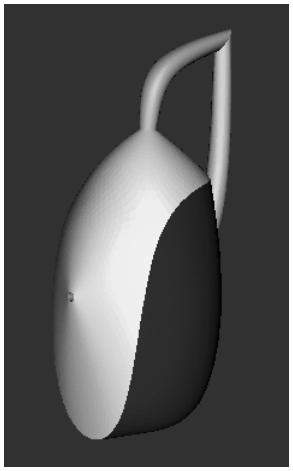


Subdivision Surfaces

With Ghost Cells

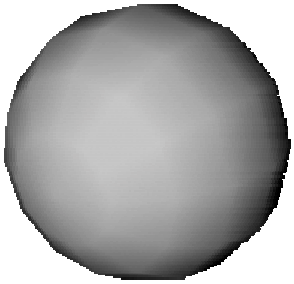


Without Ghost Cells

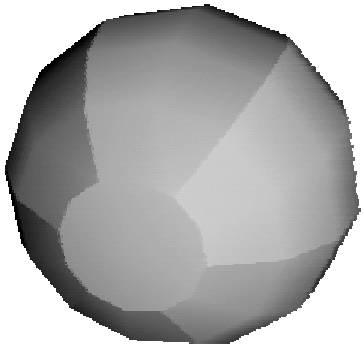


vtkPolyDataNormals

With Ghost Cells



Without Ghost Cells



Example

```
void vtkPolyData::RemoveGhostCells(int level)
{
  // Get a pointer to the cell ghost level array.
  vtkDataArray* temp = this->CellData->GetArray("vtkGhostLevels");
  …
  unsigned char* cellGhostLevels =((vtkUnsignedCharArray*)temp)->GetPointer(0);
  …
  If (this->Polys)
    {
    newPolys = vtkCellArray::New();
    newPolys->Allocate(this->Polys->GetSize());
    for (this->Polys->InitTraversal(); this->Polys->GetNextCell(npts, pts); )
      {
      if (int(cellGhostLevels[inCellId]) < level)
        { // Keep the cell.
        newPolys->InsertNextCell(npts, pts);
        newCellData->CopyData(this->CellData, inCellId, outCellId);
        ++outCellId;
        } // Keep this cell.
      ++inCellId;
      } // for all cells
    this->SetPolys(newPolys);
    newPolys->Delete();
    newPolys = NULL;
    }
…
```

Example

```
int vtkExtentTranslator::PieceToExtentThreadSafe(…)
{
  memcpy(resultExtent, wholeExtent, sizeof(int)*6);
  int ret;
  if (byPoints)
    {
    ret = this->SplitExtentByPoints(piece, numPieces, resultExtent, splitMode);
    }
  else
    {
    ret = this->SplitExtent(piece, numPieces, resultExtent, splitMode);
    }
…
if (ghostLevel > 0)
    {
    resultExtent[0] -= ghostLevel;
    resultExtent[1] += ghostLevel;
    resultExtent[2] -= ghostLevel;
    resultExtent[3] += ghostLevel;
    resultExtent[4] -= ghostLevel;
    resultExtent[5] += ghostLevel;
    }
```


How Do Filters Handle Pieces

- Many graphics algorithms work without changes
  - Hedgehogs, shrinking, warping, contouring
- Some need to specify ghost levels required
  - Normal generation, extract faces
- Others are more difficult
  - Algorithms that propagate point changes through a mesh -- polygonal smoothing
  - Algorithms that propagate topology changes – decimation
  - Algorithms that propagate results -- streamlines

How Do Filters Handle Pieces

Some mesh altering algorithms can be handled with a multipass technique

1. Decimate each piece of a surface but preserve the boundaries at full resolution
2. Combine the resulting pieces
3. Decimate the result to achieve an even reduction in mesh detail, essentially decimate the boundaries

Parallel Execution Models

- Data Parallelism
- Task Parallelism
- Pipeline Parallelism

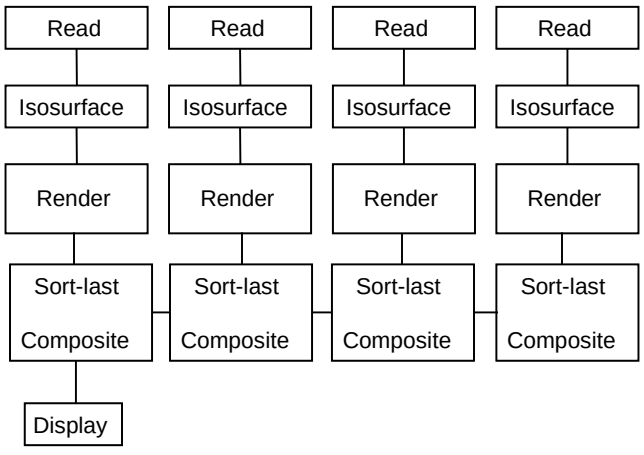
Large-scale data visualization using parallel data streaming, Ahrens et al., IEEE Computer Graphics and Application, July/August 2001

Data Parallelism

- Identical pipelines execute in parallel processing different pieces of the data
- Scalable
- Might require load balancing when processing unstructured data

SC2002
Tutorial M9

Data Parallelism

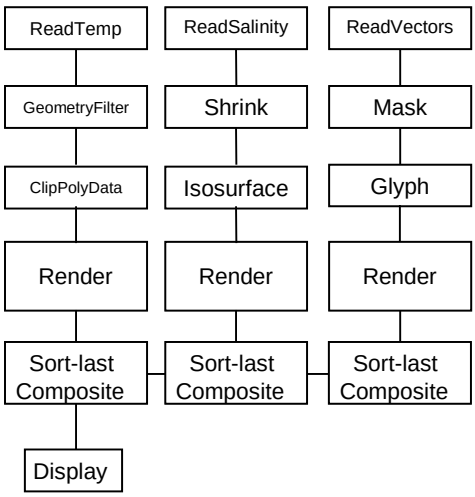


Task Parallelism

- Different pipelines work on the same data performing independent tasks
- Useful when there are multiple resources with different capabilities
- Maximum parallelism limited to the number of independent tasks
- Hard to balance

SC2002
Tutorial M9

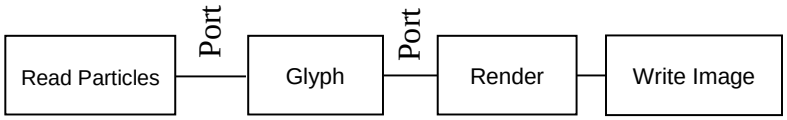
Task Parallelism



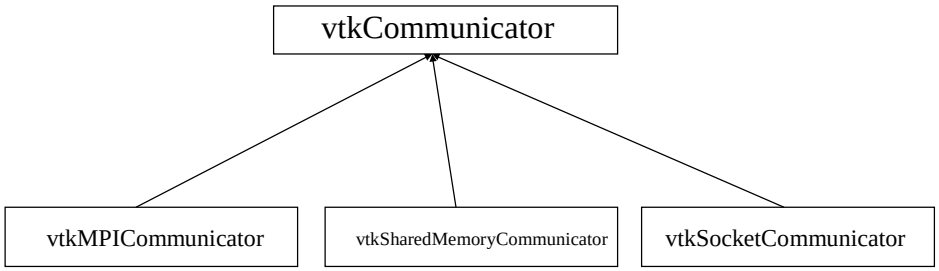
Pipeline Parallelism

- Sequence of filters/pipelines execute in parallel on the same data. Each “module” occupies a different resource.
- Useful when there are heterogeneous tasks, multiple heterogeneous resources
- Parallel execution of heterogeneous resources (Ex: filesystem, processor, graphics card)
- Maximum parallelism limited to pipeline length and ability to balance pipeline stages

Pipeline Parallelism



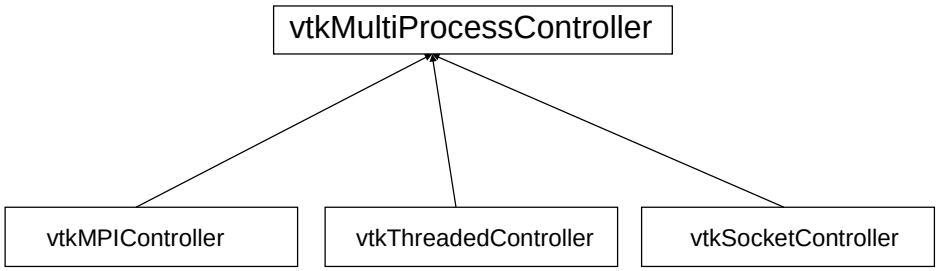
Interprocess Communication



vtkCommunicator

- Abstract class responsible of interprocess communication
- Can send/receive C arrays, vtkDataArrays and vtkDataObjects:
  - vtkCommunicator::Send(vtkDataObject*, int remoteId, int tag);
  - vtkCommunicator::Send(vtkDataArray*, int remoteId, int tag);
  - vtkCommunicator::Send(int*, int length, int remoteId, int tag);
  - vtkCommunicator::Receive(vtkDataObject*, int remoteId, int tag);
  - vtkCommunicator::Receive(vtkDataArray*, int remoteId, int tag);
  - vtkCommunicator::Receive(int*, int length, int remoteId, int tag);
- Sub-classes may have special functionality (for example, asynchronous message handling)

Parallel Execution / RMI



vtkMultiProcessController

- Used to control the flow of multiple processes triggering registered callbacks (remote method invocation - RMI)
  - Initialize(int* argc, char*** argv);
  - Finalize();
  - AddRMI(vtkRMIFunctionType*, void* localArg, int tag);
  - TriggerRMI(int remoteId, void* arg, int argLen, int tag);
  - ProcessRMIs();
  - Barrier();
  - GetNumberOfProcesses();
  - GetLocalProcessId();

Communication Example

(VTK/Parallel/Testing/Cxx/GenericCommunicator.cxx)

```
int main(int argc, char** argv)
{
  vtkMultiProcessController* contr = vtkMultiProcessController::New();
  contr->Initialize(&argc, &argv);
  contr->CreateOutputWindow();

  contr->SetMultipleMethod(0, Process1, 0);
  contr->SetMultipleMethod(1, Process2, &args);
  contr->MultipleMethodExecute();

  contr->Finalize();
  contr->Delete();

  return 0;
}
```


Communication Example (cont.)

```
void Process2(vtkMultiProcessController *contr, void *)
{
vtkCommunicator* comm = contr->GetCommunicator();

  int i;

  // Test sending all supported types of arrays
  int datai[scMsgLength];
  for (i=0; i<scMsgLength; i++)
    {
    datai[i] = i;
    }
  vtkIntArray* ia = vtkIntArray::New();
  ia->SetArray(datai, 10, 1);
  if (!comm->Send(ia, 0, 11))
    {
    cerr << "Client error: Error sending data." << endl;
    }
  ia->Delete();
}
```

```
void Process1(vtkMultiProcessController *contr, void*)
{
  vtkCommunicator* comm = contr->GetCommunicator();

  int i;

  // Test receiving all supported types of arrays
  vtkIntArray* ia = vtkIntArray::New();
  if (!comm->Receive(ia, 1, 11))
    {
    cerr << "Server error: Error receiving data." << endl;
    }
  for (i=0; i<ia->GetNumberOfTuples(); i++)
    {
    if (ia->GetValue(i) != i)
      {
      cerr << "Server error: Corrupt integer array." << endl;
      }
    }
  ia->Delete();
}
```

Ports

- Output -> input port pairs
- Handles update protocol (using RMIs)
- Modified times in different processes
- Transmitting information and data
- Output ports wait on RMI loop

Ports Are Flexible

- Data Parallel / Pieces
- Ports allow task parallel
- Pipeline parallelism is possible

vtkOutputPort / vtkInputPort

- vtkOutputPort
  - SetInput(vtkDataObject*);
  - WaitForUpdate();
  - SetPipelineFlag();
- vtkInputPort
  - GetPolyDataOutput();
  - GetUnstructuredGridOutput()…
  - SetRemoteProcessId();

Port / RMI Example

(VTK/Examples/ParallelProcessing/Generic/Cxx/ParallelIso.cxx)

```
int main( int argc, char* argv[] )
{
  vtkMultiProcessController *controller;

  // Note that this will create a vtkMPIController if MPI
  // is configured, vtkThreadedController otherwise.
  controller = vtkMultiProcessController::New();

  controller->Initialize(&argc, &argv);

  // Use this method to get the place of the data directory.
  char* fname = vtkTestUtilities::ExpandDataFileName(argc, argv,
                                     "Data/headsq/quarter");

  controller->SetSingleMethod(MyMain, reinterpret_cast<void*>(fname));

  controller->SingleMethodExecute();

  delete[] fname;

  controller->Finalize();
  controller->Delete();

  return 0;
}
```

Port / RMI Example (cont.)

```
// This will be called by all processes
void MyMain( vtkMultiProcessController *controller, void *arg )
{
....
 // Obtain the id of the running process and the total
 // number of processes
 myid = controller->GetLocalProcessId();
 numProcs = controller->GetNumberOfProcesses();

 // Create the reader, the data file name might have
 // to be changed depending on where the data files are.
 reader = vtkImageReader::New();
 reader->SetDataByteOrderToLittleEndian();
 reader->SetDataExtent(0, 63, 0, 63, 1, 93);
 reader->SetFilePrefix(fname);
 reader->SetDataSpacing(3.2, 3.2, 1.5);

 // Iso-surface.
 iso = vtkContourFilter::New();
 iso->SetInput(reader->GetOutput());
 iso->SetValue(0, ISO_START);
 iso->ComputeScalarsOff();
 iso->ComputeGradientsOff();

 // Compute a different color for each process.
 elev = vtkElevationFilter::New();
 elev->SetInput(iso->GetOutput());
 vtkMath::RandomSeed(myid * 100);
 val = vtkMath::Random();
 elev->SetScalarRange(val, val+0.001);
```

SC2002
Tutorial M9

Port / RMI Example (cont.)

```
if (myid != 0)
  {
  // If I am not the root process

  // Satellite process! Send data through port.
  vtkOutputPort *upPort = vtkOutputPort::New();

  // Last, set up a RMI call back to change the iso surface value.
  // This is done so that the root process can let this process
  // know that it wants the contour value to change.
  controller->AddRMI(SetIsoValueRMI, (void *)iso, ISO_VALUE_RMI_TAG);

  // connect the port to the output of the pipeline
  upPort->SetInput(elev->GetPolyDataOutput());

  // Multiple ports can go through the same connection.
  // This is used to differentiate ports
  upPort->SetTag(PORT_TAG);

  // Loop which processes RMI requests.
  // Use vtkMultiProcessController::BREAK_RMI_TAG to break it.
  // The root process with send a ISO_VALUE_RMI_TAG to make this
  // process change it's contour value.
  upPort->WaitForUpdate();

  // We are done. Clean up.
  upPort->Delete();
  }
```

```
else
  {
  // If I am the root process

  // Add my pipeline's output to the append filter
  app->AddInput(elev->GetPolyDataOutput());
  app->ParallelStreamingOn();

  // This is the main thread: Collect the data and render it.
  for (i = 1; i < numProcs; ++i)
    {
    downPort = vtkInputPort::New();
    downPort->SetRemoteProcessId(i);

    // Multiple ports can go through the same connection.
    // This is used to differentiate ports
    downPort->SetTag(PORT_TAG);

    app->AddInput(downPort->GetPolyDataOutput());

    // Reference already incremented by AddInput(). Delete()
    // will only decrement the count, not destroy the object.
    // The ports will be destroyed when the append filter
    // goes away.
    downPort->Delete();
    downPort = NULL;
    }
```

Port / RMI Example (cont.)

```
// call back to set the iso surface value.
void SetIsoValueRMI(void *localArg, void* vtkNotUsed(remoteArg),
                    int vtkNotUsed(remoteArgLen), int vtkNotUsed(id))
{
  float val;

  vtkContourFilter *iso;
  iso = (vtkContourFilter *)localArg;
  val = iso->GetValue(0);
  iso->SetValue(0, val + ISO_STEP);
}
```

```
// loop through some iso surface values.
  for (j = 0; j < ISO_NUM; ++j)
    {
    // set the local value
    SetIsoValueRMI((void*)iso, NULL, 0, 0);
    for (i = 1; i < numProcs; ++i)
      {
      // trigger the RMI to change the iso surface value.
      controller->TriggerRMI(i, ISO_VALUE_RMI_TAG);
      }

    // Time the rendering. Note that the execution on all processes
    // start only after Update()
    timer->StartTimer();
    app->Update();
    timer->StopTimer();
    numTris = iso->GetOutput()->GetNumberOfCells();
    val = iso->GetValue(0);
    cerr << "Update " << val << " took " << timer->GetElapsedTime()
         << " seconds to produce " << numTris << " triangles\n";

    // now render the results
    renWindow->Render();
    }
```


Distributed Rendering

- vtkCompositeManager helper class
- Hooks into rendering and interactor methods
- Uses event notification classes in VTK
- Synchronizes cameras and windows
- Sends/composites color buffer and z buffer
- Can be more efficient than ports
- Communication costs for each render

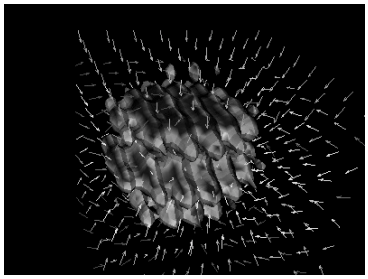
Example

(VTK/Examples/ParallelProcessing/Generic/Cxx/TaskParallelism.cxx)

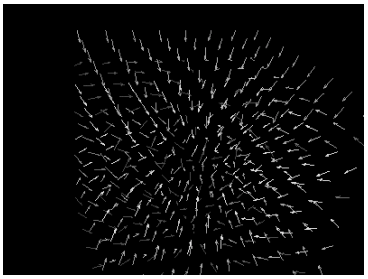
```
// Create the render objects
vtkRenderWindow* renWin = vtkRenderWindow::New();
renWin->SetSize( WINDOW_WIDTH, WINDOW_HEIGHT );

vtkRenderWindowInteractor* iren = vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);

// This class allows all processes to composite their images.
// The root process then displays it in it's render window.
vtkCompositeManager* tc = vtkCompositeManager::New();
tc->SetRenderWindow(renWin);

// Generate the pipeline see task1.cxx and task2.cxx)
vtkPolyDataMapper* mapper = (*task)(renWin, EXTENT, cam);

// Only the root process will have an active interactor. All
// the other render windows will be slaved to the root.
tc->StartInteractor();
```


Example (cont.)

Process 0



Process 1



Issues

- Compositing requires access to a display for each processor.
- When displaying multiple windows on one display, overlapping may become an issue.
- Offscreen rendering may be necessary (with Mesa for example).

Missing / To Do

- Parallel file formats / readers
- Sources which generate ghost levels
- Parallel algorithms (for example stream tracing)
- More efficient image compositing
- Better documentation / examples

Example (VTKParallelExample.cxx)

```
#include "vtkRenderer.h"
#include "vtkMultiProcessController.h"
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkCompositeManager.h"
#include "vtkPolyDataMapper.h"
#include "vtkSphereSource.h"
#include "vtkExtractPolyDataPiece.h"
#include "vtkAssignAttribute.h"

static const int WINDOW_WIDTH = 400;
static const int WINDOW_HEIGHT = 300;
```


Example (cont.)

```
// This function sets up properties common to both processes
// and executes the task corresponding to the current process
void process(vtkMultiProcessController* controller, void* vtkNotUsed(arg))
{

  // Create the render objects
  vtkRenderWindow* renWin = vtkRenderWindow::New();
  renWin->SetSize( WINDOW_WIDTH, WINDOW_HEIGHT );

  vtkRenderer* ren = vtkRenderer::New();
  renWin->AddRenderer(ren);

  vtkRenderWindowInteractor* iren = vtkRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);
```

Example (cont.)

```
// This class allows all processes to composite their images.
// The root process then displays it in it's render window.
vtkCompositeManager* tc = vtkCompositeManager::New();
tc->SetRenderWindow(renWin);

// Create the sphere
vtkSphereSource* sphere = vtkSphereSource::New();
sphere->SetThetaResolution(32);

vtkPolyDataMapper* mapper = vtkPolyDataMapper::New();
mapper->SetInput(sphere->GetOutput());
mapper->SetGhostLevel(1);

vtkActor* actor = vtkActor::New();
actor->SetMapper(mapper);
ren->AddActor(actor);
```


Example (cont.)

```
  // Only the root process will have an active interactor. All
  // the other render windows will be slaved to the root.
  tc->InitializePieces();
  tc->StartInteractor();

  // Clean-up
  actor->Delete();
  ren->Delete();
  iren->Delete();
  mapper->Delete();
  sphere->Delete();
  renWin->Delete();
}
```

Example (cont.)

```
int main( int argc, char* argv[] )
{

  // Note that this will create a vtkMPIController if MPI
  // is configured, vtkThreadedController otherwise.
  vtkMultiProcessController* controller = vtkMultiProcessController::New();
  controller->Initialize(&argc, &argv);

  // When using MPI, the number of processes is determined
  // by the external program which launches this application.
  // However, when using threads, we need to set it ourselves.
  if (controller->IsA("vtkThreadedController"))
    {
    // Set the number of processes to 2 for this example.
    controller->SetNumberOfProcesses(2);
    }
  int numProcs = controller->GetNumberOfProcesses();
```


Example (cont.)

```
  // Execute the function named "process" on both processes
  controller->SetSingleMethod(process, 0);
  controller->SingleMethodExecute();

  // Clean-up and exit
  controller->Finalize();
  controller->Delete();

  return 0;
}
```

ParaView

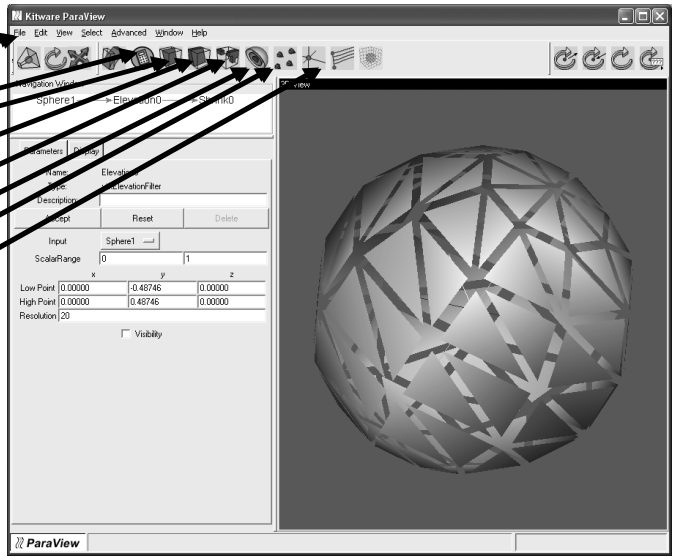
- End user tool
- Open source
- Easily extensible
- New filters in VTK easily added to UI
- Data parallel VTK pipeline
- Sort last tree composite rendering
- http://public.kitware.com/ParaView

ParaView and Tcl

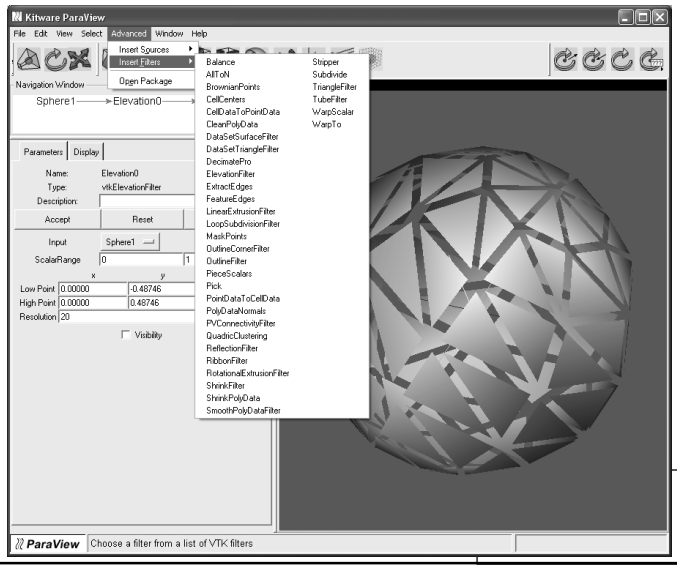
- Based on KW widget set
  - C++ obects/widgets driving TK through hidden Tcl calls
- Uses tcl to create a parallel pipeline
- Uses tcl to identify objects
  - No tags or ids are necessary
- Uses tcl to synchronize UI with pipeline.
  - Tk -> Tcl -> VTK

Standard Operations

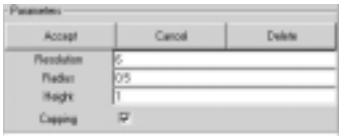
Loading data
Array Calculator
Cutting
Clipping
Threshold
Contour
Glyph
Probe



Access to VTK Filters



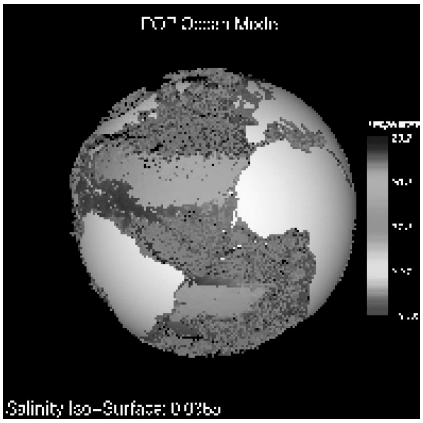
New Filters Added With XML




```
<Module name="ConeSource" root_name="Cone" output="vtkPolyData"
                                    module_type="Source">
  <Source type="vtkConeSource"/>
  <VectorEntry variable="Resolution" type="int"/>
  <VectorEntry variable="Radius" type="float"/>
  <VectorEntry variable="Height" type="float"/>
  <LabeledToggle variable="Capping"/>
</Module>
```


Level of Detail Rendering

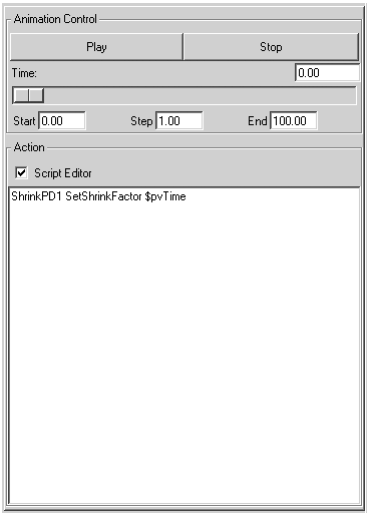
- Decimated models
  - Quadric clustering
  - Fast
  - Parallel
- Frame buffer compression
  - Sub sampling
  - Pixel replication

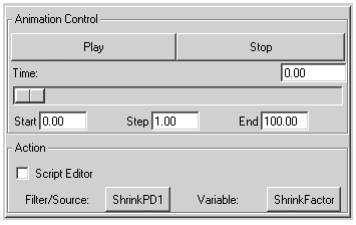


VTK Pipeline

- ParaView creates a VTK pipeline and has a simple pipeline browser.

Animations and Tcl





ParaView Exports Tcl (VTK) Scripts


```
# ParaView Version 0.1
package require vtktcl_interactor

# create a rendering window and renderer
vtkRenderer Ren1
vtkRenderWindow RenWin1
        RenWin1 AddRenderer Ren1
vtkRenderWindowInteractor iren
        iren SetRenderWindow RenWin1

# camera parameters
vtkCamera camera
        camera SetPosition 0 0 3.41078
        camera SetFocalPoint 0 0 0
        camera SetViewUp 0 1 0
        camera SetViewAngle 30
        camera SetClippingRange 2.41452 4.6731
Ren1 SetActiveCamera camera
```

o o o

Large-Scale Data Visualization Using Parallel Data Streaming

James Ahrens and Kristi Brislawn
Los Alamos National Laboratory

Ken Martin, Berk Geveci, and C. Charles Law
Kitware

Michael Papka
Argonne National Laboratory

We present an architectural approach based on parallel data streaming to enable visualizations on a parallel cluster. Our approach requires less memory than other visualizations while achieving high code reuse.

Scientists are using computer simulations to resolve models of real-world phenomenon, including models of Earth's environment, accelerator physics dynamics, and celestial bodies. With additional computing power and algorithmic advances, researchers can resolve these models to more detailed levels, increasing our understanding of the world around us. In engineering and product design, simulation continues to replace physical prototypes, resulting in reduced design cycle times and costs. The key to such applications is the visualization and analysis of simulation results. Simulations are usually run in parallel on clusters of high-bandwidth supercomputers or PCs. The resulting data sets can be so massive that they require parallel computing resources of similar magnitude to effectively visualize them.

While visualizing large data sets isn't a new problem, it remains an important and difficult one. The traditional improvements in hardware capabilities continue to make larger data sets possible and more accessible. Improvements in networking software and hardware are promoting growth in networked computing clusters. We expect that large data set visualization and the size of the data sets will continue to grow. As both large- and small-scale parallel computing resources become commonplace for scientists so must parallel visualization software that effectively uses these resources.

Visualizing large data sets is difficult for many reasons. Current analysis codes produce tera-element data sets distributed over thousands of processing nodes. In some cases, many time steps are never stored to disk but must be visualized while in memory on the processing nodes. This creates a problem, because the visualization must share the already limited resources that the simulation is using. This problem is compounded because the visualization could potentially require more storage than the simulation. Another difficulty is that some traditional visualization algorithms, such as streamline generation or mesh decimation, aren't well suited for operating on distributed data or in parallel. Furthermore, visualizations often result in processing mixed data-set topologies even when the simulation data set is a uniform topology. An isosurface of a rectilinear grid is a common example of this.

To solve these problems, we developed a visualization architecture based on mixed data-set-topology parallel data streaming. Clearly, any viable solution must support parallel execution and visualization. Mixed-topology parallel data streaming goes beyond this to incorporate data streaming so that we can keep the storage resources required for the visualization significantly smaller than those for the simulation. Additionally, it supports such streaming even when the data set's topology changes from one visualization algorithm to the next.

We implemented our architecture within the Visualization Toolkit (VTK).[1] It includes specific additions to support message passing interfaces (MPIs); memory-limit-based streaming of both implicit and explicit topologies; translation of streaming requests between topologies; and passing data and pipeline control between shared, distributed, and mixed memory configurations.[2] The architecture directly supports both sort-first and sort-last parallel rendering.[3]

This article isn't intended to address some known issues in large data-set visualization such as massively parallel I/O, effective load balancing, or parallel rendering, although we briefly discuss their implications.

Related work

While data streaming, parallel visualization, and mixed-topology visualization are all known techniques, they can be difficult and combining all three is a significant challenge.

Researchers have developed a number of out-of-core algorithms that support efficient streaming of large data.[4,5] The idea behind these approaches is to employ out-of-core or incremental algorithms with a controllable memory footprint. These methods include stream-

0272-1716/01/$10.00 © 2001 IEEE

lines, isosurfaces (modified marching cubes from disk), and related computational geometry work.[6-11] Typically, the algorithm will extract pertinent features (for example, an isosurface) and incrementally write the output to disk. Feature extraction is then followed by an interactive visualization of the extracted feature. What these algorithms lack is an overall architecture. Typically, they work independently from and to disk storage. Sometimes the developer can apply them serially but at the cost of constantly reading and writing the data to disk between each algorithm, which is a poor use of the memory hierarchy.

Systems such as Open Data Explorer (OpenDX), Application Visualization System (AVS), Demand Driven Visualizer (DDV), and SCIRun provide a pipeline infrastructure and can support parallel execution.[12] OpenDX (formerly IBM Data Explorer) and AVS are dataflow-based visualization systems, providing numerous visualization and analysis algorithms for their users.[13,14] Both systems' architectures rely on a centralized executive to some degree to instantiate modules, allocate memory, and execute modules. For example, they can achieve task parallelism with a remote module that informs the executive that it's ready to execute and waits for a signal from the executive before continuing.[15] Both systems handle data parallelism in some form in the context of a centralized executive.

SCIRun is a dataflow-based simulation and visualization system that supports interactive computational steering. SCIRun provides threaded-task and data parallelism on shared-memory multiprocessors.[2,16] An extension to SCIRun permits distributed-memory task parallelism.[17] SCIRun also uses a centralized executive and, in this way, resembles OpenDX and AVS.

All these systems provide a tightly integrated programming environment that supports interactive program construction, execution, and debugging via a graphical user interface. The existence of a single point of control for program construction and execution (that is, a GUI) may have led to the creation of a related centralized executive. However, designing an efficient mechanism for controlling many processes from a single centralized executive is difficult. In contrast to these systems, our approach avoids using a centralized executive and therefore provides a more scalable solution.

DDV provides a pipeline-based, demand-driven execution model that handles large data sets by requesting only the minimum amount of data required to produce the results.[18] This is a significant advantage for data sets with a large number of stored or computed fields. DDV and the others haven't yet addressed support for mixtures of task, data, and pipeline parallelism on both distributed and shared-memory multiprocessors.

Other solutions, such as pV3 and Ensight, encompass a variety of techniques and support large or parallel data but are designed more as turnkey applications.[19] pV3 is an implementation of the Visual3 visualization application in the parallel virtual machine (PVM) environment. The application operates on a network of heterogeneous computers that process data in pieces, ultimately sending output to a collector that gathers and displays the results. Although it's successful, pV3 a custom application, not a toolkit. Furthermore, depending on a collector is problematic in a larger data environment. Similarly, Ensight is easy to use but lacks our approach's flexibility and capabilities.

All the approaches we describe here lack the ability to stream data in memory when the data set topologies change. Because many visualization techniques can change the data's topology, this is an important consideration. Even when using unstructured grids, which are general, sometimes using a structured image is more efficient and best represented as an image and not another unstructured grid.

Streaming data

Streaming data through a visualization pipeline offers two main benefits. First, we can run visualization data that wouldn't normally fit into memory or swap. Second, we can run visualizations with a smaller memory footprint resulting in higher cache hits and little or no swapping to disk. To accomplish this, the visualization software must support breaking the data set into pieces and correctly processing those pieces. This requires that the data set and the algorithms that operate on it are separable, mappable, and result invariant:

- *Separable*. The algorithm must be able to break the data into pieces. Ideally, each piece should be coherent in geometry, topology, and/or data structure. Separating the data should be simple and efficient. In addition, the algorithms in this architecture must correctly process pieces of data.
- *Mappable*. To control the data streaming through a pipeline, we must be able to determine what portion of the input data we need to generate a given portion of the output. This lets us control the size of the data through the pipeline and configure the algorithms.
- *Result invariant*. The results should be independent of the number of pieces and the execution mode (that is, single threaded or multithreaded). This means that proper handling boundaries and developing algorithms must be multithread-safe across pieces that may overlap on their boundaries.

Other researchers have discussed an architecture that accomplishes this with regularly sampled volumetric data, such as images and volumes.[20] In that architecture, data consumers, such as rendering engines or file writers, make requests for data that are fulfilled using a three-step pipeline update mechanism.

The first step, Update Information, determines the data set's characteristics. This request is made by the data's consumer and travels upstream to the data's source. The resulting information contains the native data type (such as float or short), the largest possible extent expressed as (i_{min}, i_{max}, j_{min}, j_{max}, k_{min}, k_{max}), the number of scalar values at each point, and the pipeline-modification time. The architecture uses the native data type and number of scalar values at each point to compute how much memory a given piece of data requires. The largest possible extent is typically the data set's size on a disk. This helps determine how to break the data set into pieces and where the hard boundaries are (ver-

sus a piece's boundaries). The architecture uses the pipeline-modification time to determine when cached results can be used.

Many algorithms in a visualization pipeline must modify the information during the Update Information pass. For example, a two-times image-magnification algorithm would produce a largest possible extent that is twice as large as its input. A gradient algorithm would produce three components of output for every input component.

The second step, Update Extents, propagates a request for data (the update extent) up the pipeline (to the data source). As the request propagates upstream, each algorithm must determine how to modify the request—specifically, what input extent is required for the algorithm to generate its requested update extent. For many algorithms, this is a simple one-to-one mapping. For others, such as a two-times magnification or gradient computation using central differences, the required input extent differs from the requested extent. For this reason, the algorithms must be mappable. A side effect of the Update Extents pass is that it returns the total memory required to generate the requested extent. This enables streaming based on a memory limit. For example, a simple streaming algorithm would propagate a large update extent that exceeds the user specified memory limit. Then, it must break the update extent into smaller pieces until it does fit. This requires that the data set be separable. More flexible streaming algorithms can switch between dividing a data set by blocks or slabs and by what axis.

The final step, Update Data, causes the visualization pipeline to process the data and produce the update extent requested in step two. These three steps require a significant amount of code to implement, but surprisingly, their CPU overhead is negligible. Typically, the performance speedup provided by better cache locality more than compensates for the additional overhead. The exception is when boundaries cells are recomputed multiple times, because they're shared between multiple pieces. This is typical in neighborhood-based algorithms, and it creates a tradeoff between piece size (memory consumption) and recomputing shared cells (computation).

This entire three-step process is initiated by the data's consumer such as a writer that writes to disk or a mapper that converts the data into OpenGL calls. In both these cases, the streaming is effective because the entire result is never stored in memory at one time. It's either written to disk in pieces or sent to the rendering hardware in pieces. It's also possible to stream in the middle of a visualization pipeline if there's an operation that requires a significant amount of input but produces a fairly small output.

Streaming within the VTK is simple. Consider the following pseudocode example:

```
// Create the pipeline
MyDataSource source
source SetStandardDeviation( 0.5 )

// Iso-surfacing
ContourFilter contour
contour SetInput (source)
contour SetContourValue( 220 )

// set a memory limit
PolyDataMapper mapper
mapper SetInput (contour)
mapper SetMemoryLimit( 50 )
```

An instance of an analytical volumetric source is created in this example called source. It's then connected to a contour filter that is then connected to a mapper. A 50-Mbyte memory limit is set on the mapper that will initiate streaming if the memory consumption exceeds that limit. The mapper converts the resulting contour data into graphics primitives. The only change made to this program to support streaming is the `SetMemoryLimit` call on the mapper.

Mixed topologies

The last section described how to stream data, but it didn't consider the problems associated with streaming unstructured data or mixtures of structured and unstructured data. Streaming unstructured data has several challenges. First, we must define an extent for unstructured data sets. With regularly sampled volumetric data, such as images, we can use an extent defined as (i_{min}, i_{max}, j_{min}, j_{max}, k_{min}, k_{max}), but this doesn't work with unstructured data. With unstructured data a few options exist. One is to use a geometric extent such as (x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}), but it's an expensive operation to collect the cells that fit into that extent, and such an extent is difficult to translate into the extents used for structured data if they aren't axis aligned. (Consider a curvilinear grid.)

A more practical approach is defining an unstructured extent as piece M out of N possible pieces. Dividing the pieces is done based on cells so that piece 2 of 10 out of a 1,000-cell data set contains 100 cells. The memory-limit-based streaming approach is the same for structured data except that instead of splitting the data into blocks or slabs, the number of pieces, N, increases. This fairly basic definition of a piece dictates that there isn't any control over what cells a piece will contain, only that it will represent about $1/N$ of the data set's total cells.

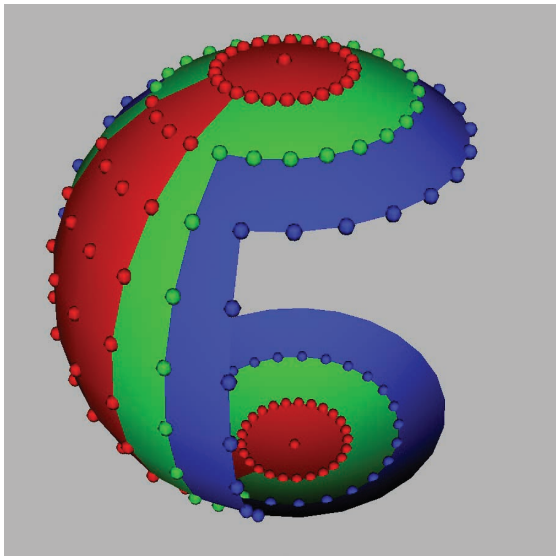
This raises the issue of how to support unstructured algorithms that require neighborhood information. The solution is to use ghost cells, which aren't normally part of the current extent but are included because the algorithm requires them.[21] To support this, we extend the definition of an unstructured extent to be piece M of N with G ghost levels. This requires that any source of unstructured grid data be capable of supplying ghost cells. There's a related issue in that some unstructured algorithms, such as contouring, operate on cells while others, such as glyphing, operate on points. Points on the boundary between two different extents will be shared by them, resulting in duplicated glyphs when processed. To solve this, we indicate which points in an extent are owned by that extent versus the ones that are ghost points. This way, point-based algorithms can operate on the appropriate points and still pass other points through to the cell-based algorithms that require them.

In the end, we require both ghost cells and ghost points to properly process the extents.

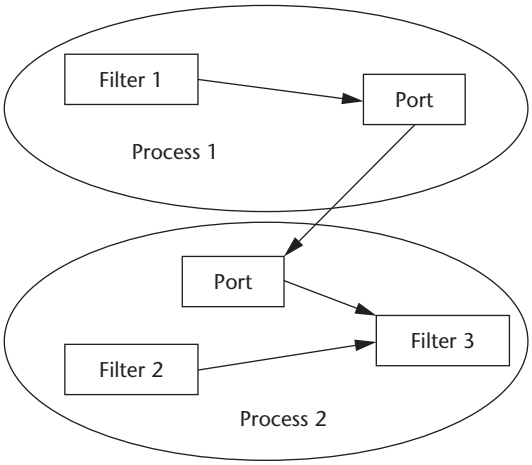
Consider Figure 1, which shows one piece of a sphere. The figure shows the requested extent in red and two ghost levels of cells in green and blue. The point colors indicate their ownership: the requested extent owns all the red points and the green and blue points indicate ownership of the points by other extents. Note that some cells use a mixture of points from different extents.

Now that we have defined extents for both structured and unstructured data, we must define the conversion between them. For most operations that take in structured data and produce unstructured data, the architecture can use a block-based division to divide the structured data into pieces until there are *N* pieces as requested. If this requires ghost cells, the block's resulting extent can be expanded to include them. If ghost-point information is required, it can be generated algorithmically based on the largest possible extent and on some convention regarding what boundary points belong to which extent.

We can convert an extent from unstructured to structured data in a similar manner except that it's inappropriate for most algorithms that convert unstructured to structured data. Consider a gaussian-splatting algorithm that takes an unstructured grid and resamples it to a regular volume. Producing one part of the resulting volume requires all the cells of the unstructured grid that would splat into that extent. With our definition of an unstructured extent, there's no guarantee that the cells in an extent are collocated or topologically related. So to generate one extent of structured output requires that the algorithm examine all the unstructured data. Although the algorithm can do this within a loop, our current implementation requires that when translating from a requested structured extent to an unstructured extent, the entire structured input is requested.



1 Breaking up a sphere into a piece (red) and ghost-level cells and points (blue and green).



2 Pipeline execution across process boundaries.

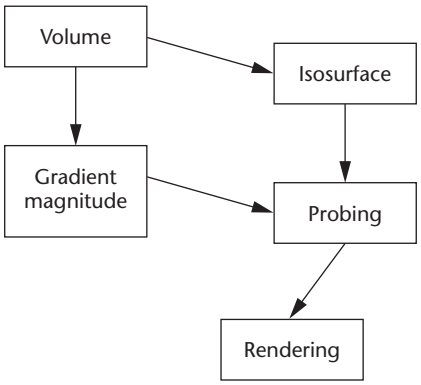
Supporting parallelism

Most large-scale simulations use parallel processing and often the results are distributed across many processing nodes. This requires that the visualization algorithms be capable of operating in such an environment. Supporting parallelism requires some of the same conditions as streaming, such as data separability and result invariance. It also requires asynchronous execution, data transfer, and collection.

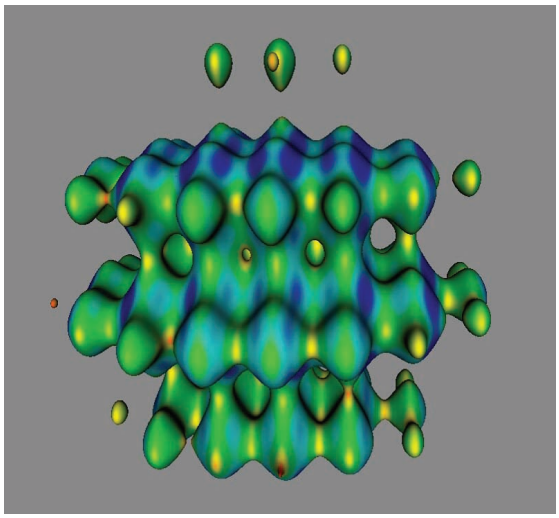
We ensure data transfer by creating input and output port objects that can communicate between filters (algorithms) in different processes. In turn, we require asynchronous execution so that one process isn't unnecessarily blocked, waiting for input from another process. Consider the pipeline in Figure 2. In this example, Filter 3 has two inputs. Its first input, Filter 1, is in another process, so it requires an input and output port to manage the interprocess communication. Before Filter 3 executes, it must make sure both of its inputs have generated their data. A naive approach would be to simply ask each input to generate its data in order. The problem is that while Filter 3 is waiting for Filter 1 to compute its data, Filter 2 is idle.

To solve this, we made two modifications to the three-step update process. The first modification was to add a nonblocking method to the update process called Trigger Asynchronous Update. This method starts the execution of any inputs in other processes. Essentially, this method traverses upstream in the pipeline, and when it encounters a port, the port calls Update Data on its input.

The second modification is to use the locality of the inputs to determine in what order to invoke Update Data on them. We define an input's locality as 1.0 if the input is generated within the same process, 0.0 if the input is generated in a different process, and between 0.0 and 1.0 if the input is partially generated in one and partially in another (such as in a long pipeline where half of the algorithms are in one process and half in another). This locality is computed as part of the Update Information call. So in Figure 2, Trigger Asynchronous Update would be sent to Filter 1, which would cause Filter 1 to start executing because it's in a different process. Filter 2 would ignore the Trigger Asynchronous Update call since there aren't any ports between it and Filter 3. Then, Filter 3 would call Update Data on Filter 2 first, because it has the highest locality. Once Filter 2 has completed executing, Update Data would be called on Port



3 Data-parallel visualization example.



4 Resulting image from the data-parallel example in Figure 3.

2, which could already have the results in memory if Filter 1 (which has been executing since the Trigger Asynchronous Update call) has completed executing.

In addition to these infrastructure changes, we encapsulated process initialization and communication calls into a class so that the user doesn't have to deal with them directly. We created concrete subclasses for distributed-memory and shared-memory processes using MPI and pthreads. Likewise, we wrote a sort-last parallel rendering class that uses interprocess communication to collect and then composite parallel renderings into a final image. We support centralized rendering by collecting the polygonal data together using ports between processes connected to an append filter in the collection process. This architecture can also implement parallel rendering using polygon collection and then parallel rendering such as WireGL.[22]

Given this parallel data-streaming architecture, we can create a data parallel program by simply writing a function that will execute on each processor. Inside that function, each processor will request a different extent of the results based on its processor ID. Each processor can still take advantage of data streaming if its local memory isn't sufficient, letting this architecture process large-scale visualizations.

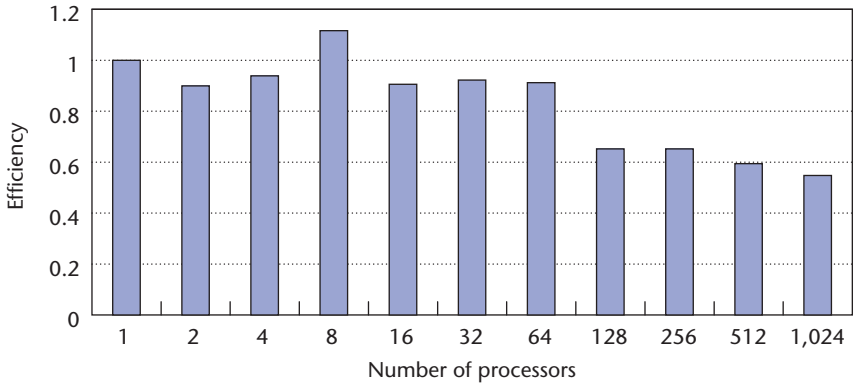
Consider modifying the earlier pseudocode example to support data parallelism and streaming. First, we define a function called `process` that contains the bulk of the pipeline creation and rendering. This function will be invoked by `MultiProcessController`, which encapsulates the setup and initialization of the processes. In this example, we use the `MPIController` subclass of `MultiProcessController`. It's passed into the function as an argument and it provides information such as the process ID and total number of processes. The visualization pipeline is created as usual but the requested piece (*M*) and total number of pieces (*N*) are set on the mapper. This way the mapper of each process will only create its piece of the total *N* pieces. The memory limit is still set in case generating piece *M* of *N* requires excessive memory. Then, the mapper can break down the request into smaller subpieces. An instance of the `TreeComposite` class is created and the render window is assigned to it. This class encapsulates the sort-last parallel rendering technique. Then, a `Render` call is made to the renderer that will start the rendering process, streaming, and finally the tree-compositing. The `main()` function creates an instance of `MPIController`, which is one of the subclasses of `MultiProcessController`; assigns a function for it to execute; and then executes it.

```
process(MultiProcessController ctrl)
{
  myId = ctrl GetLocalProcessId()
  numPrcs = ctrl GetNumProcs()

  // Create the pipeline
  MyDataSource source
  source SetStandardDeviation( 0.5 )

  // Iso-surfacing
  ContourFilter contour
  contour SetInput (source)
  contour SetValue( 220 )

  PolyDataMapper mapper
  mapper SetInput (contour)

  // Set the total number of pieces
  mapper SetNumberOfPieces( numProcs
)
  mapper SetPiece( myId )
  mapper SetMemoryLimit( 50000 )

  Actor actor1
  actor1 SetMapper( mapper )
  RenderWindow renWin
  Renderer renderer1
  renWin AddRenderer( renderer1 )
  renderer AddActor( actor1 )

  // setup the tree composite and
     render
  TreeComposite treeComp
  treeComp SetRenderWindow( renWin )
  renWin Render()
}

main()
```

```
  {
    MPIController controller
    controller Initialize()

    ctrl
SingleMethodExecute(
process )
  }
```

Results

The results we report here are based on using an in-memory analytic function as a data source. We designed this to mimic visualizing data from a running simulation where the simulation data is in the memory. This also avoids dealing with issues of massively parallel I/O, which are beyond the scope of this article. We organized the data as a regular volumetric data set with a double precision scalar value computed at each point. We tested three different visualization examples: the first two on a cluster of eight SGI Origin 2000s, each with 128 shared-memory processors, and the third on a cluster of eight PCs, each with two shared-memory processors.

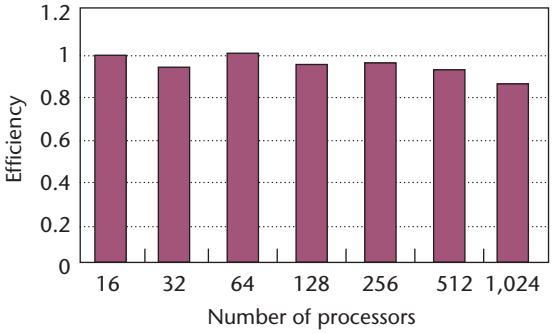


5 Results of a 39-Gbyte data-parallel visualization.

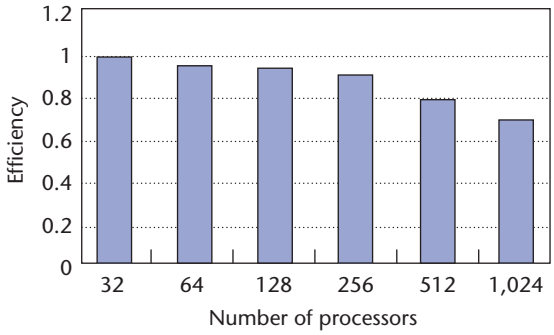
The first visualization example was a data-parallel pipeline that computes an isosurface and gradient magnitude field from the volume. It then color-maps the gradient magnitude onto the isosurface using a probe filter and renders the result using a sort-last parallel rendering technique (see Figures 3 and 4). We ran this example with input data sizes of 39 Gbytes, 1.1 Tbytes, and 0.9 Pbytes on configurations between 1 and 1,024 processors. These sizes each represent a single data set, not multiple time steps. We rendered the polygons produced in software using Mesa.

The 39-Gbyte run produced 20-million polygons. We reported its results in terms of efficiency versus the number of processors (see Figure 5). The efficiency is a measure of how effectively the additional processors are being used. An efficiency of 1.0 represents a linear speedup versus the number of processors. The results are based on the wall-clock processing time required and include any time required to start the processes and allocate memory for each one. The 39-Gbyte test is small enough that for anything beyond 64 processors the startup time dominates the actual calculation time. Consider that linear scaling would result in a 10-second execution on 1,024 processors, while the time required for MPI to start 1,024 processes and for each of them to allocate their memory is about 90 seconds. The results show linear performance up to about 64 processors. Beyond that, the calculation is simply too quick to make using more processors worthwhile. If the visualization were to be generated at the end of each time step, so that the process could be kept running, then using 1,024 processes would be valuable.

We provide the results of the 1.1-terabyte run for 16 to 1,024 processors since running on one to eight processors would be too time consuming (see Figure 6). This run produced 190-million polygons and with the larger problem size the results are nearly linear across the entire range. The worst case is the results for 1,024 processors that show an efficiency of 0.86 for a 418-second execution time. We expected this due to the process initialization time.



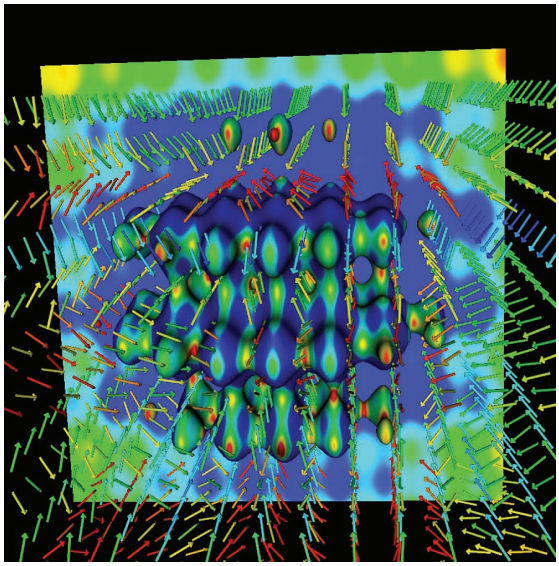
6 Results of a 1.1-Tbyte data-parallel visualization.



7 Results of a 1.1-Tbyte task-parallel visualization.

We tested the 0.9-petabyte run on 1,024 processors. It required 360,000 seconds and produced 16-billion polygons. It's worth noting that the time required for this run was nearly linear with respect to the time required for the 1.1-terabyte run on 1,024 processors. This is because of the streaming of the data. The 0.9-petabyte run requires the same memory footprint as the 1.1-terabyte run.

The second visualization example demonstrates task parallelism, where there are multiple independent visualization pipelines (see Figure 7). In our example, there were three pipelines. The first pipeline is the probed isosurface pipeline that we used in the first



8 Resulting image from the combined task- and data-parallel example.

example. The second pipeline computes a gradient vector field from the input data. Then, it reduces the resolution and creates oriented glyphs at each point. The third pipeline extracts a cut plane from the input data and displays it (see Figure 8). In a fully data-parallel configuration, all three tasks would be run on each processor similarly to the data-parallel example. For contrast, in this test, we distributed the tasks across the processors with the majority of the processors assigned to generating the probed isosurface. Therefore, the example is task parallel with each task using data parallelism across the processors it was allocated. The results indicate successful task parallelism with a slightly less than linear speedup because of poor load balancing between the tasks.

The third example considered pipeline parallelism, where one processor performs some of the visualization while another performs the rest of it. This is common in cases where the graphics resources are available to only some of the processors. We simulated this case by running the data-parallel example on a cluster of eight Windows 2000 machines connected via a gigabit Ethernet. Each machine had two processors and one had an accelerated OpenGL graphics card. We decided to use the screen for hardware-accelerated rendering, which limited us to eight hardware renderers even though there were 16 processors. The hardware rendering consumed less than 1 percent of the total time.

Simple modifications to the first example allowed the use of both processors on a machine for the computation, while we only used one processor to transmit the data to the rendering hardware. We used sort-last compositing to combine the eight hardware renderings into the final buffer. This resulted in a linear speedup from 8 to 16 processors due to the hardware rendering's high performance and the shared-memory data transfer's low cost. This capability is significant, because in many cases, the hardware isn't homogeneous and standard data parallel approaches won't fully use the available resources. In this case, the first processor could render the data while the second processors was computing the next piece. For this hardware configuration, it let us use all 16 processors where otherwise we would have only used eight.

Discussion

Although this article has addressed some difficult issues, we are still addressing others. In many simulations with distributed data, the ghost cells can only be obtained from other processes. Currently, there isn't a standard mechanism for one process to determine where to find specific ghost cells. Ideally, there would be an efficient mechanism so that an algorithm that required ghost cells could determine what process to request them from. Additionally, some algorithms, such as streamlines, require parallel-specific versions to be written that can pass information concerning when a streamline exits one piece and enters another. We're actively researching these issues in hopes of incorporating such capabilities into our architecture. ■

Acknowledgments

This work was supported in part by grants from the US Department of Energy ASCI Views program and the DOE Office of Science. We acknowledge the Advanced Computing Laboratory of the Los Alamos National Laboratory, where we performed portions of this work on its computing resources.

References

1. W.J. Schroeder, K.M. Martin, and W.E. Lorensen, *The Visualization Toolkit An Object-Oriented Approach to 3D Graphics*, Prentice Hall, Upper Saddle River, N.J., 1996.
2. S.G. Parker, D.M. Weinstein, and C.R. Johnson, "The SCIRun Computational Steering Software System," *Modern Software Tools in Scientific Computing*, E. Arge, A.M. Brauset, and H.P. Langtangen, eds., Birkhauser Boston, Cambridge, Mass., 1997, pp. 1-40.
3. S. Molnar et al., "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, vol. 4, no. 4, July 1994, pp. 23-31.
4. M. Cox and D. Ellsworth, "Application-Controlled Demand Paging for Out-Of-Core Visualization," *Proc. IEEE Visualization 1997*, ACM Press, New York, 1997, pp. 235-244.
5. M. Cox and D. Ellsworth, "Managing Big Data for Scientific Visualization," *Exploring Gigabyte Datasets in Real-Time: Algorithms, Data Management, and Time-Critical Design*, Siggraph 97, Course Notes 4, ACM Press, New York, 1997.
6. Y.J. Chiang and C.T. Silva, "Interactive Out-of-Core Isosurface Extraction," *Proc. IEEE Visualization 1998*, ACM Press, New York, 1998, pp. 167-174
7. T.A. Funkhouser et al., "Database Management for Models Larger Than Main Memory," *Interactive Walkthrough of Large Geometric Databases*, Course Notes 32, Siggraph 95, ACM Press, New York, 1995.
8. I. Itoh and K. Koyamada, "Automatic Isosurface Propagation Using an Extrema Graph and Sorted Boundary Cell Lists," *IEEE Trans. Visualization and Computer Graphics*, vol. 1, no. 4, Dec. 1995, pp. 319-327.

9. S. Subramanian and S. Ramaswamy, "The P-Range Tree: A New Data Structure for Range Searching in Secondary Memory," *Proc. ACM/SIAM Symp. Discrete Algorithms*, SIAM, Philadelphia, Pa., 1995, pp. 378-387.
10. S. Teller et al., "Partitioning and Ordering Large Radiosity Computations," *Proc. Siggraph 94*, ACM Press, New York, 1994, pp. 443-450.
11. S.K. Ueng, K. Sikorski, and K.-L. Ma, "Out-of-Core Streamline Visualization on Large Unstructured Meshes," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 4, Oct.–Dec. 1997, pp. 370-380.
12. D. Song and E. Golin, "Fine-Grain Visualization Algorithms in Dataflow Environments," *Proc. IEEE Visualization 1993*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 126-133.
13. G. Abrams and L. Trenish, "An Extended Data-Flow Architecture for Data Analysis and Visualization," *Proc. IEEE Visualization 1995*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 263-270.
14. C. Upson et al., "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics and Applications*, vol. 9, no. 4, July 1989, pp. 30-42.
15. M. Krogh and C. Hansen, "Visualization on Massively Parallel Computers using CM/AVS," *AVS Users Conf.*, 1993, pp. 129-137, http://www.acl.lanl.gov/Viz/abstracts/ParallelAC-AVS.html.
16. C.R. Johnson and S. Parker, "The SCIRun Parallel Scientific Computing Problem-Solving Environment," *Ninth SIAM Conf. Parallel Processing for Scientific Computing*, SIAM, Philadelphia, Pa., 1999.
17. M. Miller, C. Hansen, and C. Johnson, "Simulation Steering with SCIRun in a Distributed Environment," *Applied Parallel Computing, Fourth Int'l Worksho*p (PARA 98), Lecture Notes in Computer Science, no. 1541, B. Kagström, J. Dongarra, E. Elmroth, and J. Wasniewski, eds., Springer-Verlag, Berlin, 1998, pp. 366-376.
18. P.J. Moran and C. Henze, "Large Field Visualization With Demand-Driven Calculation," *Proc. IEEE Visualization 1999*, ACM Press, New York, 1999, pp. 27-33.
19. R. Haimes and D.E. Edwards, *Visualization in a Parallel Processing Environment*, American Inst. of Aeronautics and Astronautics, Reston, Va., 1997.
20. C.C. Law et al., "A Multithreaded Streaming Pipeline Architecture for Large Structured Data Sets," *Proc. IEEE Visualization 1999*, ACM Press, New York, 1999, pp. 225-232.
21. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI, Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, Mass., 1994.
22. G. Humphreys et al., "Distributed Rendering for Scalable Displays," *Proc. Supercomputing*, CD-ROM, ACM Press, New York, 2000.

James Ahrens *is a technical staff member at Los Alamos National Laboratory. His research interests include scientific visualization, computer graphics, parallel and distributed systems, and component architectures. He has a PhD in computer science from the University of Washington and is a member of the IEEE Computer Society.*

Kristi Brislawn *is a technical staff member at Los Alamos National Laboratory. She is interested in developing parallel algorithms and software. She has an MS in applied math from the University of Colorado.*

Ken Martin *is a computer scientist at Kitware, working in scientific and medical visualization. His research interests include computer vision, computer graphics, and software architectures. He received his PhD in computer science from Rensselaer Polytechnic Institute, studying model-based camera pose estimation. He is a coauthor of VTK and The Visualization Toolkit textbook (Prentice Hall, 1997).*

Berk Geveci *is a research and development engineer at Kitware. His research interests include scientific visualization, computational mechanics, and object-oriented programming. He received his MS and PhD in mechanical engineering from Lehigh University.*

C. Charles Law *is a researcher at Kitware. His interests include parallel and large-data visualization, path planning, and maintainability analysis. He has a PhD in neuroscience from Brown University.*

Michael E. Papka *is a software project engineer in the Futures Laboratory of the Mathematics and Computer Science Division at Argonne National Laboratory. He works with fellow group members to design new visualization technologies that combine the use of advanced storage systems, advanced networking, virtual-space technology, and high-end virtual environments to construct advanced tools for scientific research.*

Readers can contact Martin at Kitware, 469 Dlifton Corporate Pkwy., Clifton Park, NY 12065; ken.martin@kitware.com.

A Multi-Threaded Streaming Pipeline Architecture for Large Structured Data Sets

C. Charles Law (*Kitware, Inc.*)
William J. Schroeder (*Kitware, Inc.*)
Kenneth M. Martin (*Kitware, Inc.*)
Joshua. Temkin (*RPI*)

Abstract

Computer simulation and digital measuring systems are now generating data of unprecedented size. The size of data is becoming so large that conventional visualization tools are incapable of processing it, which is in turn is impacting the effectiveness of computational tools. In this paper we describe an object-oriented architecture that addresses this problem by automatically breaking data into pieces, and then processes the data piece-by-piece within a pipeline of filters. The piece size is user specified and can be controlled to eliminate the need for swapping (i.e., relying on virtual memory). In addition, because piece size can be controlled, any size problem can be run on any size computer, at the expense of extra computational time. Furthermore, pieces are automatically broken into sub-pieces and each piece assigned to a different thread for parallel processing. This paper includes numerical performance studies and references to the source code which is freely available on the Web.

1 Introduction

Computer simulation and digital measuring systems are now generating data of unprecedented size. For example, Kenwright [Kenwright98b] describes computational fluid dynamics data sets of sizes ranging up to 600 GByte with larger data (terabytes) foreseen. Machiraju reports computational data sets of similar sizes [Machiraju98]. Measuring systems are generating large data as well. It is anticipated that the Earth Orbiting Satellite (EOS) will generate a terabyte of data *daily*. Volumetric data sources such as CT, MRI, and confocal microscopy generate large volumetric data sets; the addition of time-captured data will multiply the overall sizes of these data sets dramatically.

A primary goal of visualization is to communicate information about large and complex data sets [Schroeder97]. Generally, the benefit of visualization increases as the size and complexity of data increases. However, as data sizes increase, current visualization tools become ineffective due to loss of interactivity; or even fail, as data overwhelms the physical and virtual memory of the computer system. If researchers, engineers, scientists, and users of large data are to take full advantage of advances in computational simulation and digital measurement systems, visualization system must be designed to handle data sets of arbitrary size.

1.1 Why Visualization Systems Fail

Conventional commercial visualization systems such as AVS [AVS89] and IBM Data Explorer [DataExplorer] fail in two important ways when encountering large data. First, interactive control of the visualization process is lost when the time to transmit, process, or render data becomes prohibitively large. (Large may mean millions of cells or primitives, which is relatively small compared to the data sizes quoted previously.) This difficulty causes significant delays in processing results because the ability to rapidly explore data is replaced with a hit-and-miss batch process or other ad hoc methods to reduce data size or extract regions of interest.

While the loss of interactivity is a serious problem, visualizing larger datasets may cause complete system failure. In this second failure mode, the physical or virtual memory address space of the computer is overwhelmed, and the system thrashes ineffectively or crashes catastrophically. The typical response to this problem is to buy larger computers such as a supercomputer, but this solution is prohibitive for all but the wealthiest computer users, and in many cases, may not solve the problem on the largest datasets.

These failures are typically due to one of four problems [Cox97a] [Cox97b]) 1) The data may be too large for local computer memory resulting in excessive thrashing. 2) The data may be too large for local disk (either for storage or paging to virtual memory), making the system unusable. 3) The data may be too large for the combined capacity of remote and local disk. 4) The bandwidth and latency of data transfer causes bottlenecks and results in poorly performing or unusable systems. Creating successful visualization systems for big data requires addressing each of these four problems.

1.2 Goals of Large Data Visualization

We see two fundamental, but opposing, design goals for large data visualization systems.

1. The system must be able to process any size data, on any size computer, without loss of information. The ability of a computer system to treat large data must scale well with processing power and available memory.
2. The system must be allow users to quickly identify important regions in the data, and then enable focused attention on those regions in ever greater detail (to the limit of the resolution of the data).

The quandary for the visualization scientist is that these goals call for the system to be as accurate as possible, and at the same time as interactive as possible. Accuracy is necessary when observing detailed quantitative behavior or comparing data. Interactivity is necessary to understand the overall structure of data or when looking for important features or other qualitative relationships. Often users adopt both goals during a visualization session. Interactive exploration is used to identify features of interest, at which point accurate visualizations are generated in order to understand detailed behavior or data values.

In this paper we focus on the first goal: to process any size data on any size computer, with good scalable characteristics as the computer system grows in memory, computational power, and data bandwidth. We believe this goal is the necessary starting point, since it is necessary to visualize large data before we can visualize it interactively.

2 Approach

Successfully managing large data requires breaking data into pieces, and then processing each piece separately, or distributing each piece across a network for parallel processing. To obtain interactive performance, parallel algorithms must be developed (distributed and/or multithreaded) to achieve the highest possible processing rates, and the total data to be processed must be minimized by employing segmentation algorithms, subsampling, or multiresolution methods.

A subtle but fundamental requirement of a successful system is that it must manage the hierarchy of memory efficiently to achieve maximum data throughput. The hierarchy of memory ranges from tape archival systems, to remote disk, local disk, physical memory, cache, and register memory. The speed of data access can vary by orders of magnitude between each level, and significant computational expense is required to move the data between each level. Therefore, a well designed system will manage the memory hierarchy to avoid moving data between levels, and holding it at the highest possible level until processing is complete.

2.1 Desirable Features

We believe that there are several important characteristics of visualization systems that successfully contend with big data. Some of these include the following.

Graceful Degradation. Ideally, visualization systems should degrade predictably (e.g., linearly) as the data size grows relative to the computer capacity expressed as a function of CPU performance, memory, and data bandwidth. Systems exhibiting this characteristic instill confidence in users, since the desired system performance can be controlled by hardware expenditures—what you pay for is what you get. It also means that no matter the size of the computer or the data, given enough time the system can process the data. Practically what this means is that a single processor PC with 64 megabytes of memory connected to a data source via network or local data bus should generate the same results as a high-end, multiprocessor supercomputer with several gigabytes of memory, the difference being the time taken to complete the visualization.

Minimizes Disk Access. One important lesson learned from conventional virtual memory operating systems is that depending on disk storage for computer memory is undesirable. Most users have experienced running programs which scale gracefully until the system begins swapping, at which point the elapsed time to complete the computation dramatically increases. This is because the time to write or read pages from disk is large compared to the time the CPU takes to perform a single instruction. For example, typical disk access time is on the order of tens of milliseconds (primarily disk head movement). During this same period of time several million operations can be performed by the CPU.

Cached. Modern computer systems are designed with data caches to improve performance. Data caches store pieces of data close to the CPU to minimize delays during read and write operations. Caches dramatically improve performance because frequently used data or code can be accessed faster than less frequently used data or code. Appropriate use of caching when visualizing big data objects can dramatically increase the performance of the system.

Parsimonious. Cox and Ellsworth [Cox97a] [Cox97b] have observed that the amount of data actually generated by a visualization algorithm is small compared to the total amount of data. For example, streamline generation is typically of order $O(N)$, where N^3 is the size of the dataset. Isosurface generation is typically of order $O(N^2)$ since a surface is generated from a volume of data. The implication is that sparse or parsimonious traversal methods can be created which dramatically reduce the amount of data accessed by the system.

Parallel. Parallel processing techniques have demonstrated their ability to greatly accelerate computation, and computer systems are often configured with more than one processor. Visualization systems must take advantage of this capability to deliver timely results. Current visualization systems are parallelized in two ways: either on a per algorithm basis (fine-grained parallelism) or across parallel branches of data flow (coarse-grained parallelism). Both methods of parallelism should be supported.

Hardware Architecture Independent. It is important to develop systems not tied to a particular computer architecture. For example, depending on shared memory parallel processing will fail once data size exceeds shared memory space. Similarly, systems based on a network of heterogeneous computers exclude "typical" engineers and scientists users with a modest single processor system. Visualization systems must be adaptable to a variety of hardware architectures, including the simplest single processor modest memory system of most users.

Demand-Driven. Demand-driven or lazy evaluation systems perform operations on data only when absolutely required. In many cases this avoids unnecessary work. For example, since computing streamlines in a grid requires only

a portion of the original data, loading and processing only the data necessary to perform particle advection can reduce computational and memory costs significantly. Of course, the advantages of caching described earlier point to the fact that lazy evaluation must be tempered by careful consideration of the granularity of data access and evaluation.

Component Based. Commercial visualization systems such as AVS and IBM Data Explorer clearly demonstrate the benefit of component based systems. These systems enable users to run-time configure visualizations by connecting objects, or components, into data flow networks. The network is then executed to generate the desired result. Benefits of this approach include adaptability and flexibility to changing data and visualization needs. The power of component-based systems is that the components are general and can be reused in different applications. Such reuse can be a significant advantage over customized applications, since the effort to maintain and tune components has immediate impact across all applications using them. Tailored applications often suffer from a limited user base (which has impact on long-term survivability of the application), and are often difficult to modify-it is often not possible to drop in a new component to replace an older one.

Streaming. Many visualizations consist of a sequence of operations. For example, in our work we routinely use combinations of data subsampling (extract portion of data), isosurfacing, decimation (reduce mesh size), Laplacian smoothing, and surface normal generation to create high quality images of 3D contour surfaces. These operations are typically implemented by applying a pipeline of reusable components to a stream of data. In conventional use, the size of the data stream is controlled by the size of the input data. When the data is big, however, we would prefer to control the size of the data stream based on run-time configurable memory limits, perhaps on a component-by-component basis.

Other important features of a good software design such as efficiency, ease of use, extensibility, and robustness are also important and assumed.

2.2 Previous Approaches

Two general approaches have been used to process large data sets: use of out-of-core algorithms and design of large data architectures. Multiresolution methods form a third approach, and are typically used in visualization systems with a primary goal of interactivity.

2.2.1 Large Data Algorithms

Feature extraction has been a successful technique for processing large data sets. The idea behind these approaches is to employ out-of-core or incremental algorithms with a controllable memory footprint. These methods include isosurfaces (modified marching cubes from disk) [Chiang98] [Itoh95] and related computational geometry work [Agarwal98] [Rama94] [Sub95] [Teller94] [Vengroff96] [Funk95], streamlines [Ueng98], separation and attachment lines [Kenwright98], and vortex core lines ([Kenwright97]). Typically the algorithm will extract pertinent features (e.g., an isosurface) and incrementally write the output to disk. Feature extraction is then followed by an interactive visualization of the extracted feature.

There are two difficulties with this approach. First, the approach presumes that the extracted features are sufficiently small enough to fit into system memory—a poor assumption as data sizes increase in size. Eventually, the extracted features will be large enough that they will not fit into memory. Second, the extraction of features depends on I/O from disk. Data that must be processed by a series of filters must be read and written to and from disk several times, a poor use of the memory hierarchy, and likely to result in poorer performing visualization systems.

2.2.2 Large Data Architectures

Another approach is the design of architectures to directly support large data visualization. For example, researchers at NASA Ames replace the virtual memory system, using an intelligent paging scheme that recognizes the structure and boundaries of data [Cox97a] [Cox97b]. This approach has demonstrated significant improvements over standard virtual memory. It is best suited for the application of a single algorithm. We believe this approach is not as effective for supporting a pipeline of filtering operations. Using the virtual paging scheme, memory is repeatedly swapped as each algorithm processes the entire dataset, one after the other. Instead, we believe that ingesting a *piece* (subset of the entire dataset) of data, and then processing the entire piece—*without any swapping*—through the pipeline can achieve dramatically better results. This approach, which we call *data streaming*, has the benefit that data remains in higher-performing memory as it is processed.

Another successful system was developed by Haimes. This system, pv3, is an implementation of the Visual3 visualization application in the pvm environment. The application operates on a network of heterogeneous computers that process data in pieces, ultimately sending output to a collector that gathers and displays the results. While successful, pv3 is not a general application solution since it is a custom application, and offers no reusable components. Furthermore, depending on a collector is problematic in the larger data environment. This solution also seems vulnerable to network bandwidth and latency limitations.

3 Multi-Threaded Data Streaming

The key to the structured data streaming pipeline is the ability to break data into pieces. By controlling the piece size carefully, we can avoid swapping and insure that the data is processed in physical memory (or better). The

piece size is set at run-time depending on the size of computer memory, the number of filters in the visualization pipeline, and the size of data. In addition, using the same process to break data into pieces, we can break pieces into sub-pieces for the purpose of multi-threading, each processor taking one sub-piece. Thus our approach naturally supports shared-memory parallel processing.

The overarching goal of this work was to create a multi-threaded, streaming architecture for the structured points (i.e., images and volumes) portion of the *Visualization Toolkit (VTK)* [Schroeder97]. Limiting the problem to structured data greatly simplified the design. In the future we plan on extending our approach to unstructured data.

3.1 Key Principles

The following three principles guided the design of the multi-threaded data streaming architecture for structured data.

1. *Data Separability*. The data must be separable. That is, the data can be broken into pieces. Ideally, each piece should be coherent in geometry, topology, and/or data structure. The separation of the data should be simple and efficient. In addition, the algorithms in this architecture must be able to correctly process pieces of data.
2. *Mappable Input*. In order to control the streaming of the data through a pipeline, we must be able to determine what portion of the input data is required to generate a given portion of the output. This allows us to control the size of the data through the pipeline, and configure the algorithms.
3. *Result Invariant*. The results should be independent of the number of pieces, and independent of the execution mode (i.e., single- or multi-threaded). This means proper handling of boundaries and developing algorithms that are multi-thread safe across pieces that may overlap on their boundaries.

Structured data is readily separable— the topological *i-j-k* coordinate scheme naturally breaks data into coherent pieces and was employed as the separation mechanism in the architecture. Most structured (i.e., imaging) algorithms are mappable since the input pixels required to produce an output pixel are known. And finally, careful design of the algorithms enables proper treatment of the boundary of each piece, thereby insuring that the output remains invariant as the number of pieces changes.

3.2 Architectural Overview

As previously mentioned, the basic idea behind our design is to control the amount of data passing through the pipeline at any given time. To do this, we replace the typical visualization pipeline architecture shown in Figure 1(top) with that shown in Figure 1(bottom). As shown in the figure, we augment the usual process object/data object pair with a third object—a data cache. The purpose of the cache is to manage allocation and access to the data. Furthermore, the cache negotiates with its upstream and downstream filters to configure the pipeline for execution. The negotiation process considers available memory (memory limits are set on a per cache basis), requested memory, algorithm kernel size (to determine whether boundary padding is required), and the size of the input and output data. In addition, each algorithm (i.e., filter) expects to operate on a piece of data rather than the entire data set. In fact, because the filters are designed to operate on pieces of data, it is possible to break a piece into sub-pieces and assign them to separate processor threads for parallel processing with little to no changes in the code.

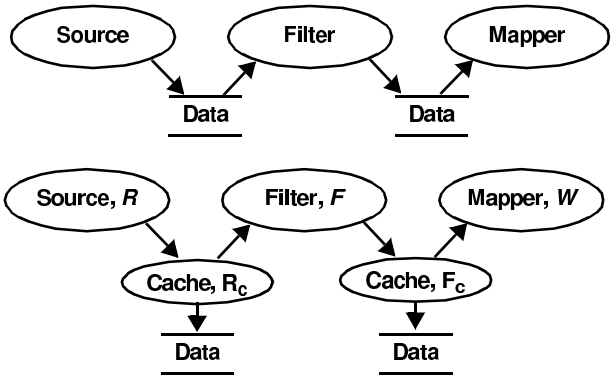


Figure 1. A conventional visualization pipeline (top) compared to a streaming pipeline (bottom).

3.3 Configuring the Pipeline

We will use Figure 1(bottom) to illustrate the streaming architecture. Three process objects (a source *R*, filter *F*, and mapper object *W*), associated caches, and data objects are shown. In this example assume that the mapper is generating output data that may be written to disk or rendered, and further assume that the input data size is greater than a user-specified memory limit.

When the request comes to write the data, a two stage process is initiated (the pipeline is demand driven, so the pipeline only executes when data is requested). In the first stage, the pipeline configures itself for streaming. The writer object requests a portion of data of size S from its input cache F_C. F_C has a memory limit S_C != S. Assume in this example that F_C grants a request of size S_C with $S_C < S$. W accepts this grant and configures itself to process data in pieces of size S_C. The configuration process continues with filter F. The filter is limited to generate an output of size S_C (its cache limit), and has knowledge of the algorithm it implements such as kernel size (how many input pixels are required to generate an output pixel) and the relative difference in size between input and output (e.g., vtkImageMagnify can change output size by integer multiples of input size). This information is translated into a request for input of a particular size S_R from cache R_C. The negotiation process is repeated and filter F configures itself to operate on pieces of appropriate size, S_R. Finally,

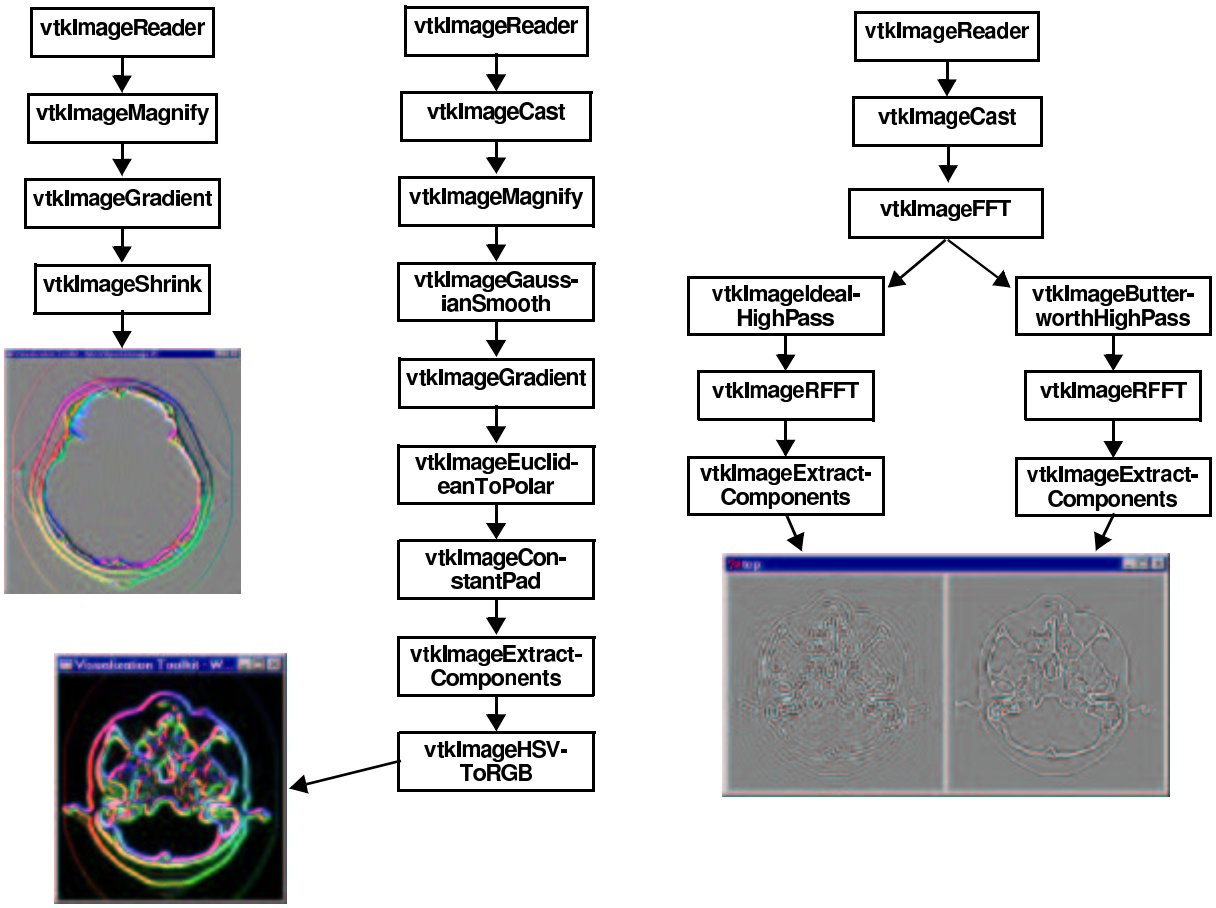


Figure 3. Three pipelines used to test the streaming architecture. The vtkImageCastFilter in pipelines #1 and #2 is used to control the size of the input data. Note that while the images show just a single slice, the entire input volume is processed.

the reader configures itself to generate pieces of size S_R which completes the first stage of the configuration process.

In the second stage, the pipeline begins execution and begins streaming data through it. Note that each process object may operate on different number of pieces depending on the results of the negotiation process. In the worst case, filters like image histogram may have to revisit their input several times (in pieces) to generate a piece of output. This is because the histogram filter requires visiting all input pixels to generate the correct results for a single output pixel.

3.4 Multithreading

Adding multi-threading to the pipeline configuration process is straightforward. Once the initial piece size is negotiated for each filter, pieces are further subdivided into sub-pieces depending on the number of available processors. Then each thread operates on a sub-piece in exactly the same way that pieces are operated on. For best performance each piece should fit in main memory and then each sub-piece will be 1/N the size of the piece, where N is the number of processors.

3.5 Handling Boundaries

Many algorithms require a kernel of data around an input data location to generate a single output value. For example, a Gaussian smoothing operation may take a 2x2 or 3x3 input kernel to generate an output data value. Such algorithms impact the process of generating pieces (and sub-pieces) since the pieces must be enlarged to support algorithm execution. This means that pieces may be generated with overlap, as shown in Figure 2. As long as the overlap size relative to the piece size is relatively small,

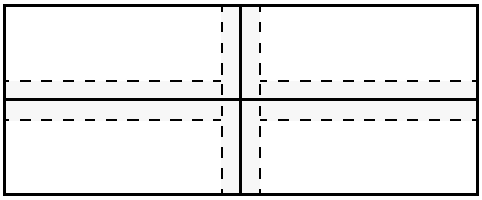


Figure 2. Pieces may be created that overlap one another. The overlap depends on algorithmic kenel size.

the impact on performance is minimal. Also, such overlap remains thread-safe, since separate threads can read from the same memory location—only when writing to the output that overlap must be eliminated.

3.6 Object-Oriented Implementation

The structured data pipeline in the VTK visualization system contains several dozen filters. One of our concerns for implementing this architecture—which is much more complex than the typical visualization system—is that a complex implementation might result in a brittle or unmaintainable system. Fortunately, we were able to embed the complexity of the architecture into three superclasses, including an abstract superclass from which most all filters are derived. Subclasses (i.e., filters) remain relatively simple to create, with the usual care required with multi-threaded implementations.

The three superclasses encapsulating the stream architecture are:

1. vtkImageSource — the superclass for vtkImageFilter and all other process objects; synchronizes pipeline execution.
2. vtkImageFilter — the superclass for most structured filters. It performs the breaking of data into pieces and coordinates multithreading.
3. vtkImageCache — manages the interface between the data objects and the process objects.

Source code for the C++ implementation is available in VTK Version 2.2 (and later versions) from http://www.kitware.com/vtk.html.

4 Results & Discussion

To demonstrate the effectiveness of the streaming architecture, we evaluated it against the three different pipelines shown in Figure 3. Each pipeline was run on a volumetric dataset of size 256x256x93 by 2 bytes, for a total input data size of 12.19 MBytes. Once read into the pipeline, the data was further expanded by the addition of an vtkImageCast filter (transformed the 2-byte short data to 4-byte floats, i.e., doubled the data size) and/or was passed through a vtkImageMagnify filter to further increase data size by 3x3x1 (i.e., a nine-times increase in data size). In addition, the filters were configured to retain their output data as the pipeline executed, and some filters (such as vtkImageGradient) expand their data during execution (vtkImageGradient by a factor of three because a scalar is expanded to a 3-vector). The total data processed by Pipeline #1 is 780 MBytes; by Pipeline #2: 3.76 GBytes; and by Pipeline #3: 475.4 MBytes.

4.1 The Effect of Swapping

The first numerical experiment demonstrates the effect of swapping on performance. Applications that depend on virtual memory suffer severe penalties because the speed of virtual memory is significantly less than physical memory. By controlling the size of data pieces streaming through the pipeline, we can avoid swapping and insure that data is processed only in physical memory.

Pipeline #1 was used to perform the experiment on a small system running Windows/NT with 128 MBytes of physical memory. While the input data is only 12.19 MBytes, the vtkImageMagnifyFilter expanded the data size by a factor of nine, followed by vtkImageGradient, which expanded the data by another factor of three, for a maximum data size of 329 MBytes. The experiment varied the cache size from 10 KBytes to 330 MBytes. (At 330 MBytes we depend entirely on system virtual memory.) We also ran the same data on a two processor system to see the effect of multiprocessing. The results are shown below.

Cache Size (MByte)	Elapsed Time (1 Processor)	Elapsed Time (2 Processors)
330	3934	1740
100	565	327
70	128	59
50	96	58
25	98	50
10	105	55
5	110	59
2	111	60
1	113	61
0.10	134	89
0.03	194	173
0.01	485	553

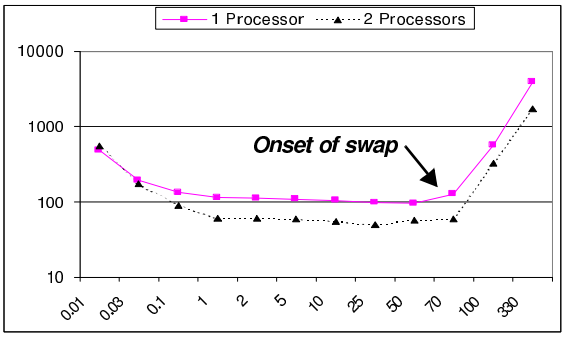


Figure 4. The effect of cache size on elapsed time for Pipeline #1.

The results, plotted in Figure 4, clearly demonstrates the effect of swapping. The best performance for a single processor system occurred when the cache size was set to 50 MBytes, which was 41 times faster than the results obtained when depending on virtual memory. Even when the cache size was set to a tiny 10 KBytes, we observed better performance than was obtained with virtual memory. There is a noticeable penalty as the cache becomes very small, since the overhead of breaking data into pieces

affects performance. The performance for two processors showed similar results, although the effect due to small cache size was greater because each piece of data is divided in half and assigned to each of the two processors.

Another benefit of the streaming architecture is that we can process data whose size is greater than the physical address space of the computer. For example, using the same 32-bit NT, dual-processor system, we were able to process a peak data size of 25 GByte (vtkImageMagnify increased data size by a factor of 676) with a cache size of 50 MByte in approximately 3000 seconds. (This is because each piece is smaller than physical address space and we never need to allocate contiguous memory for the entire data set.)

4.2 Multi-Threading and Cache Size

In the second numerical experiment, we compared the effect of varying cache size from 750 MByte down to 7.5 MByte for each of the three pipelines. In addition, the number of processors is varied between one and eight. The computer system is a large 8-processor Infinite Reality SGI (R10000) with 3.5 GByte of physical memory. Because of the large physical memory and the cache size limit, the system did not swap. Therefore, the difference in elapsed time were due to the effects of more processors (multi-threading) or the overhead of processing data in pieces.

Cache Size	1 Proc.	2	4	8
750 MByte	19.81	12.07	7.99	6.65
375 MByte	19.79	12.08	8.03	6.61
75 MByte	19.83	12.14	8.07	6.67
7.5 MByte	17.54	13.29	11.10	11.15

Figure 5. Elapsed time as number of processors and cache size is varied for Pipeline #1.

Cache Size	1 Proc.	2	4	8
750 MByte	130.6	70.6	41.23	23.51
375 MByte	133.0	70.0	40.95	24.75
75 MByte	129.8	72.2	41.87	28.60
7.5 MByte	138.0	82.2	67.98	64.60

Figure 6. Elapsed time as number of processors and cache size is varied for Pipeline #2.

Cache Size	1 Proc.	2	4	8
750 MByte	181.6	96.81	52.16	29.12
375 MByte	181.4	96.66	52.58	29.78
75 MByte	178.9	96.66	52.25	30.43
7.5 MByte	234.6	131.2	71.56	52.73

Figure 7. Elapsed time as number of processors and cache size is varied for Pipeline #3.

We noticed several interesting features in these results. First, with a single processor, the effect of cache size (breaking data into pieces) was small. In some cases the reduction in cache size actually reduced the elapsed execution time of execution, probably because the data better fit into machine cache. The major exception to this was Pipeline #3. This pipeline is different from the other two in that a branch exists in the pipeline. When generating results, the pipeline evaluates first one branch, and then the other. As the cache at the point of pipeline junction (i.e., vtkImageFFT) is reduced in size, less reusable data is cached. The net result is that for small cache sizes the two filters upstream of vtkImageFFT execute two times: first for the left branch and then for the right branch. (Performance could be improved by increasing the cache size at the point of branching.)

Another striking feature is the effect of reducing cache size combined with adding additional computational threads. Since with single processor systems we observed that the effect of streaming was relatively small, we surmise that the overhead of creating and joining threads becomes significant as the size of the cache becomes smaller. (Note: in the eight-processor case, each piece of data is processed by eight threads in each filter. Therefore, as the piece size becomes smaller, each thread works on less data, so the overhead of thread management becomes proportionally larger.)

5 Conclusions & Future Work

We have successfully designed and implemented an object-oriented architecture that can process structured data of arbitrary size on computers whose physical memory is much smaller than the data size. The architecture achieves this capability by breaking data into pieces as a function of a specified cache (or memory limit) size. In addition, the architecture supports multi-threading automatically without requiring reconfiguration of execution scripts. We found that the effect of streaming (breaking data into pieces) is small for uniprocessor systems, and that the cost of thread management becomes larger as the piece size is reduced. We also demonstrated the capability of systems to process data whose size is much greater than the size of physical memory.

Our ultimate goal is to incorporate the streaming architecture into VTK's unstructured visualization pipeline. This is a difficult task for several reasons. First, the data type changes as it passes through the pipeline; e.g., a structured grid when isosurfaced becomes a polygonal (triangle) mesh. Second, it is difficult to map the input to the output. For example, it is not known beforehand which cells, when isosurfaced, will generate surface primitives, and how many resulting primitives are generated. Third, it is difficult to break data into pieces since there are no natural boundaries in unstructured data. And finally, many algorithms are global in nature. Connectivity and streamlines require data in an unpredictable manner, or in its entirety,

in order to execute.

Incorporating the streaming architecture into the unstructured pipeline may require changes to algorithms and accepting certain compromises. For example, global algorithms such as streamline generation may be recast (use algorithms with local kernels such as LIC [Cabral93]), or decimation [Schroeder92a] may occur in patches with boundary seams visible (not results invariant). It may also be that the architecture is extended to support multipass streaming where filters retain information between each pass.

The computing environment of the future will consist of a heterogeneous mixture of single- and multi-processor computing systems arranged on a high-speed network. While the architecture described here supports a (local) multi-processor, shared memory approach, it is readily extensible to the distributed environment. In a distributed environment, data can be broken into pieces (using the same approach described here) and assigned to systems across the network. Future plans call for distributed support to be built directly into VTK.

6 Acknowledgment

This work was partially supported by the NSF Award #9872147. Thanks to Bill Lorensen for his insightful advice, to James Miller for assisting us in the numerical studies, and our colleagues at GE CRD and GE Medical Systems.

7 References

[Agarwal98] P. K. Agarwal, L. Arge, T. M. Murali, and others. I/O-Efficient Algorithms for Contour-Line Extraction and Planar Graph Blocking. In *Proc. ACM-SIAM Symp. On Discrete Algorithms*, 1998 (to appear).

[AVS89] C. Upson, T. Faulhaber Jr., D. Kamins and others. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications.* 9(4):30–42, July 1989.

[Cabral93] B. Cabral and L. Leedom. Imaging Vector Fields Using Line Integral Convolution. *Computer Graphics (SIGGRAPH '93 Proceedings).* Vol. 27. pp. 240-247.

[Chiang95] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia,and others. External-Memory Graph Algorithms. In *Proc. ACM-SIAM Symp. On Discrete Algorithms* pp. 139-149, 1995.

[Chiang97] Y.-J. Chiang and C. T. Silva. I/O Optimal Isosurface Extraction. In *Proc. Of Visualization '97*. IEEE Computer Society Press, October, 1997.

[Chiang98] Y.-J. Chiang and C. T. Silva. Interactive Out-of-Core Isosurface Extraction. In *Proc. Of Visualization '98*. IEEE Computer Society Press, October, 1998.

[Cox97a] M. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-Of-Core Visualization. In *Proc. Of Visualization '97*. IEEE Computer Society Press, October, 1997.

[Cox97b] M. Cox and D. Ellsworth. Managing Big Data for Scientific Visualization. In ACM Siggraph '97 Course #4 *Exploring Gigabyte Datasets in Real-Time: Algorithms, Data Management, and Time-Critical Design*. August, 1997.

[DataExplorer] *Data Explorer Reference Manual.* IBM Corp, Armonk, NY, 1991.

[Funk95] T. A. Funkhouser, S. Teller, C. H. Sequin, and D. Khorramabadi.Database Management for Models Larger Than Main Memory. In *Interactive Walkthrough of Large Geometric Fatabases*, Course Notes 32, Siggraph '95, August 1995.

[Itoh95] I. Itoh and K. Koyamada. Automatic Isosurface Propagation Using an Extrema Graph and Sorted Boundary Cell Lists. *IEEE Trans. On Visualization and Computer Graphics*. 1(4):319-327.

[Kenwright97] D. Kenwright and R. Haimes. Vortex Identification - Applications in Aerodynamics: A Case Study. In *Proc. Of Visualization '97*. IEEE Computer Society Press, October, 1997.

[Kenwright98] D. Kenwright. Automatic Detection of Open and Closed Separation and Attachment Lines. In *Proc. Of Visualization '98*. IEEE Computer Society Press, October, 1998.

[Kenwright98b] D. Kenwright. Presentation to Scientific Computation Research Center (SCOREC) at Rensselaer Polytechnic Institute, 1998.

[Machiraju98] E. Machiraju, A. Gaddipati, R. Yagel. Detection and Enhancement of Scale Coherent Structures Using Wavelet Transform Products. *Proc. of the Tech. Conf. on Wavelets in Image and Signal Processing* SPIE Annual Meeting, San Diego CA, 1997.

[Machiraju93] .R. Machiraju and R. Yagel, Efficient Feed-Forward Volume Rendering Techniques for Vector and Parallel Processors. *SUPERCOMPUTING'93*, Portland, Oregon, November 1993, pp. 699-708.

[Rama94] S. Ramaswamy and S. Subramanian. Path Cathcing: A Technique forOptimal External Searching. In *Proc. ACM Symp. On Principles of Database Sys.*, pp. 25-35, 1994.

[Schroeder92a] W.J. Schroeder, J. Zarge, and W.E. Lorensen. Decimation of Triangle Meshes. *Computer Graphics (SIGGRAPH `92)*, 26(2):65-70, August 1992.

[Schroeder97] W.J. Schroeder, K.M. Martin, and W.E.Lorensen. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics*. Prentice-Hall, Upper Saddle River, NJ, 1996.

[Sub95] S. Subramanian and S. Ramaswamy. The P-Range Tree: A New Data Structure for Range Searching in Secondary memory. In *Proc. ACM-SIAM Symp. On Discrete Algorithms*, pp. 378-387, 1995.

[Teller94] S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan. Partitioning and Ordering Large Radiosity Computations. In *Proc. Of SIGGRAPH '94*. pp 443-450, July, 1994.

[Ueng98] S. K. Ueng, K. Sikorski, and K.-L. Ma. Out-of-Core Streamline Visualization on Large Unstructured Meshes. *IEEE Transactions on Visualization and Computer Graphics.*

[Vengroff96] D. E. Vengroff and J. S. Vitter. Efficient 3-D Range Searching in External Memory. In *Proc. Annu. ACM Sympos. Theory, Comp.*, pp 192-201, 1996.

Out-Of-Core Scientific Visualization

Cláudio T. Silva
Oregon Health & Science University

Outline

- **Principles of External Memory Algorithms**
- **Isosurfaces**
- **Direct Volume Rendering**

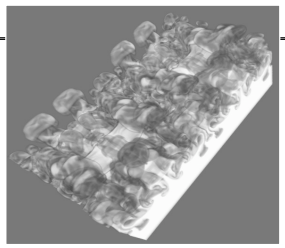
SC 2002
Tutorial M9

Principles of External Memory Algorithms

- **I/O Computational Model**
- **Algorithmic Techniques**
  - Caching & Prefetching
  - External Merge Sort
  - Out-Of-Core Pointer De-Referencing
  - Meta-Cell Technique
  - Indexing (B-Tree-Like Data Structures)

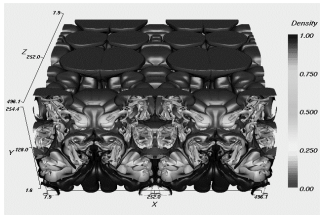
Volume Rendering vs Isosurfaces

- **Direct Volume Rendering**
  - Volume data → Image
  - Looks “inside” the data



McPherson, LANL

- **Isosurfaces**
  - Volume data → Polygon model
  - Slice of the data

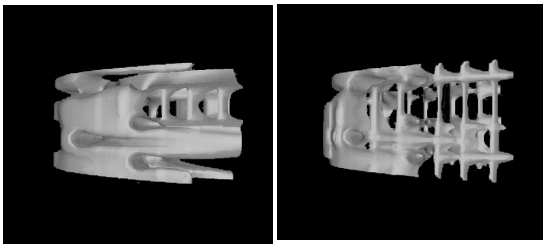


McPherson, LANL

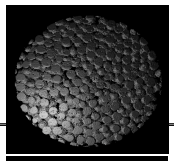
Isosurfaces

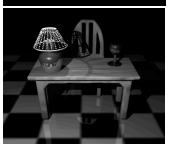
Query:

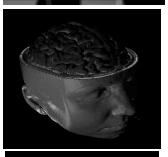
- For a query value q, find and display the isosurface of q: C(q) = {p | F(p) = q}

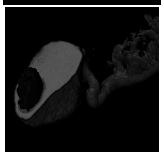


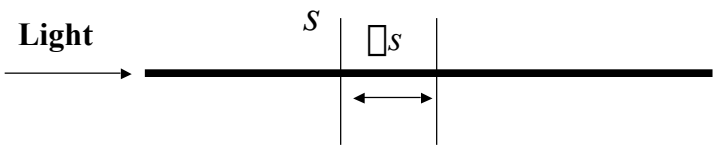
Direct Volume Rendering: Optical Models







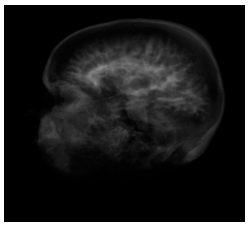




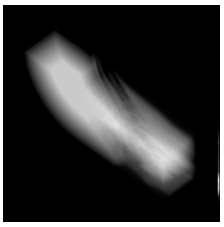
$$I(s+\Delta s) = I(s) + g(s)\Delta s - I(s)\tau(s)\Delta s$$

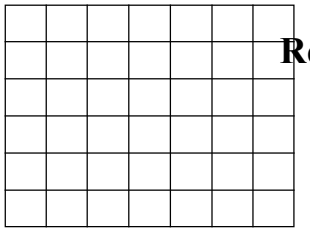
$$I(s) = \int_0^s g(x)\, e^{-\int_0^x \tau(y)\,dy}\, dx$$

Volume Rendering

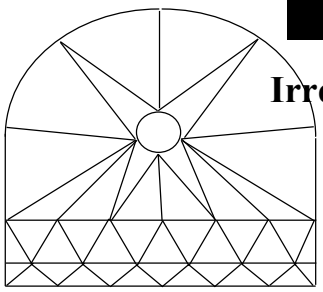


$$f : R^3 \rightarrow R$$





Regular



Irregular

SC 2002
Tutorial M9

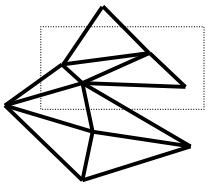
Out-Of-Core Isosurface Computations

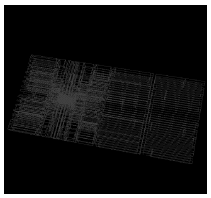
SC 2002
Tutorial M9

I/O-Efficient Hierarchical Partitioning – Meta-Cells

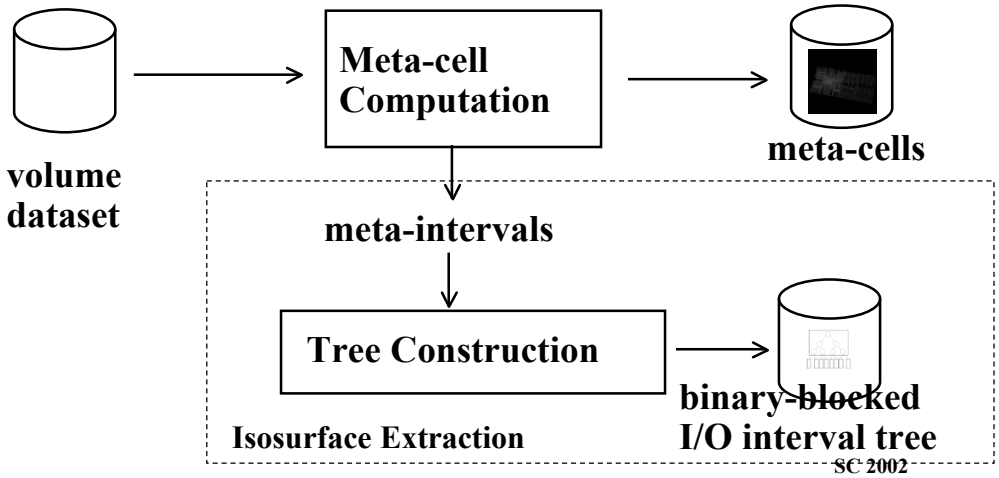
Meta-cell technique – Chiang-Silva, 98:

- partition dataset into clusters of cells (meta-cells)
- out-of-core k-d-tree-like partition
- each meta-cell has pointer references *within* the meta-cell: compact representation, efficient disk space

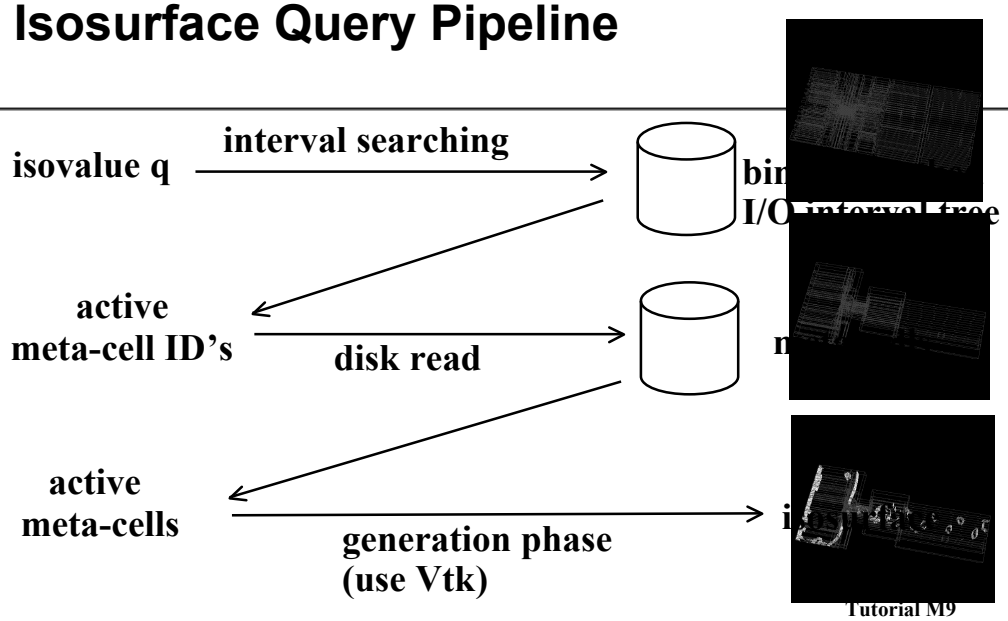




Preprocessing Pipeline

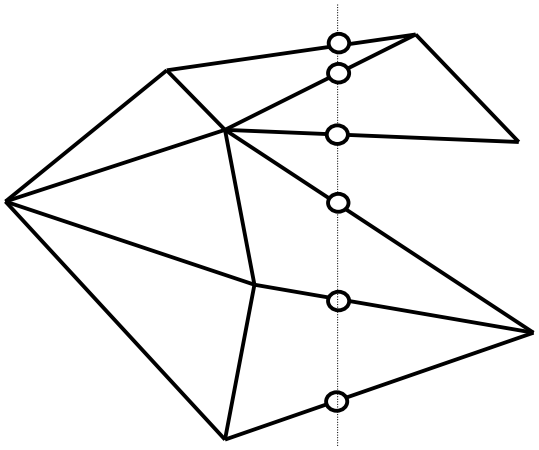


Isosurface Query Pipeline



Out-Of-Core Volume Rendering

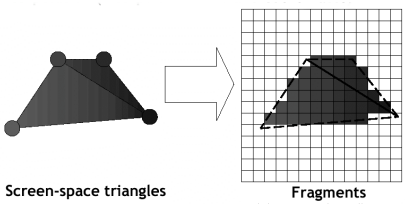
Volume Rendering:
(Intersection) Sampling + Sorting



SC 2002
Tutorial M9

Graphics Pipeline

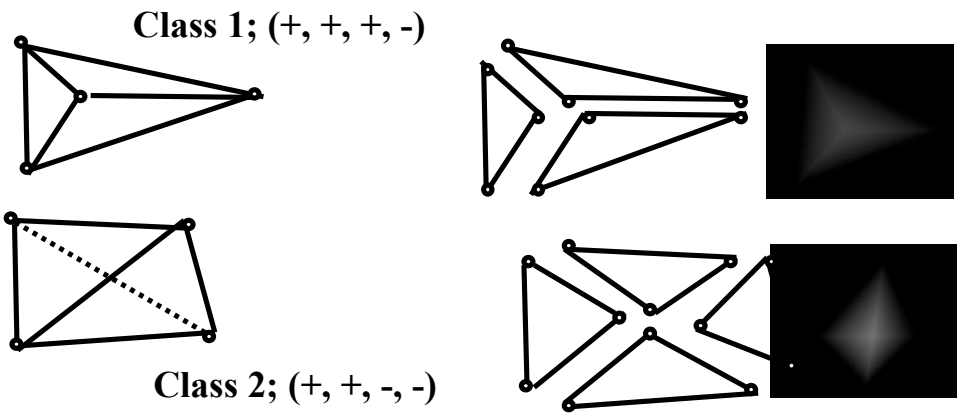
- Application
- Command
- Geometry
- Rasterization
- Texture
- Fragment
- Display



Akeley and Hanrahan, Stanford

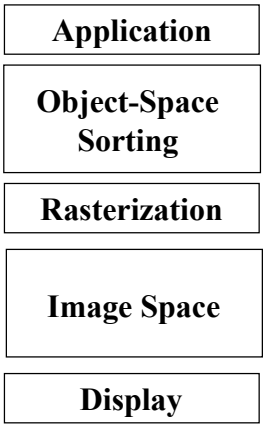
SC 2002
Tutorial M9

Sampling: Triangle-Based Approach



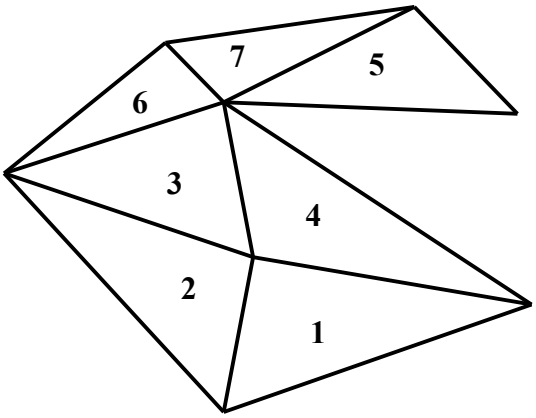
Projected Tetrahedra [Shirley-Tuchman 90]

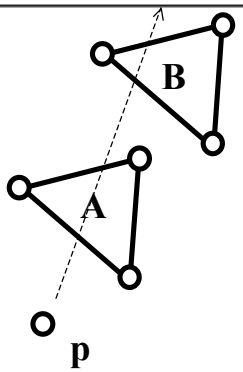
Sorting



i.e., let's sort the geometry!

Volume Rendering: Cell-Projection



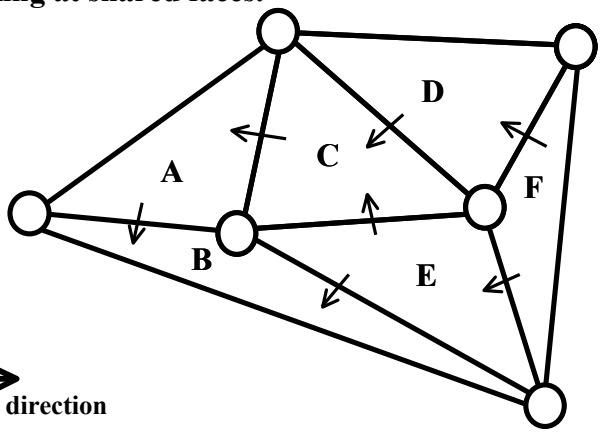


$A <_p B$

SC 2002
Tutorial M9

Object-Order Sorting Williams' MPVO

Idea: Define ordering relations by looking at shared faces.



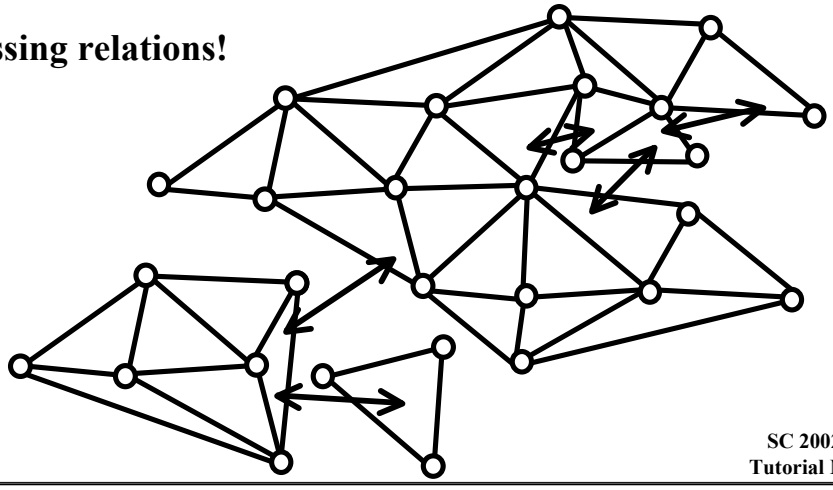
Viewing direction

B < A
A < C
B < E
C < E
C < D
E < F
D < F

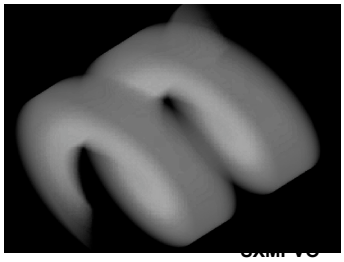
SC 2002
Tutorial M9

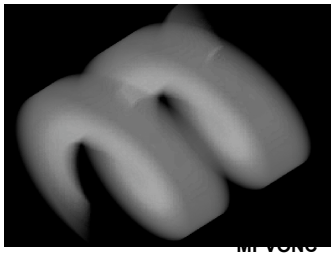
MPVO Limitations

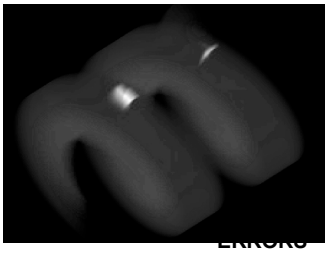
Missing relations!



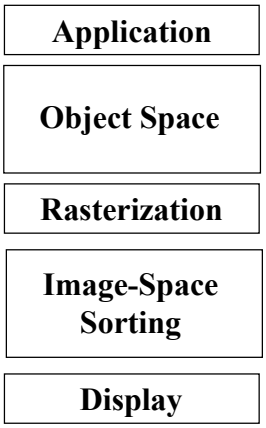
MPVONC Rendering Errors







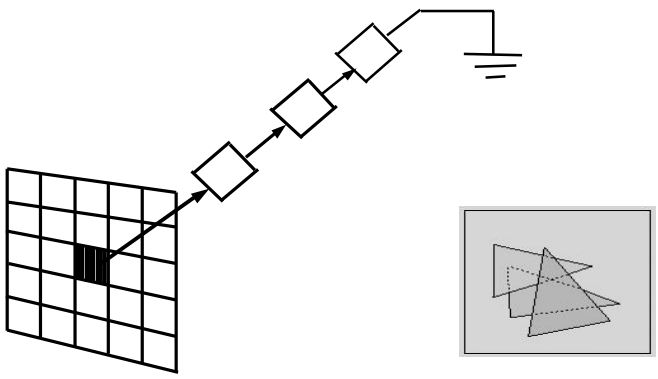
Sorting



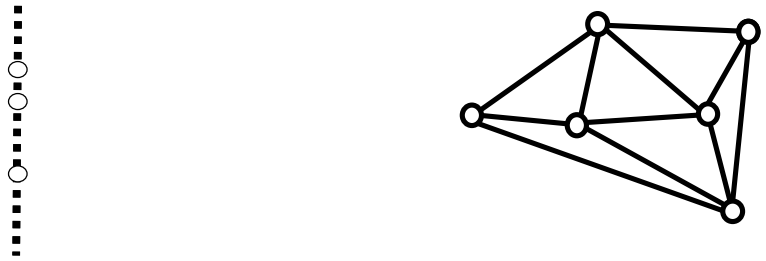
i.e., let's sort the pixes!

Image-Order Sorting: Carpenter's A-Buffer

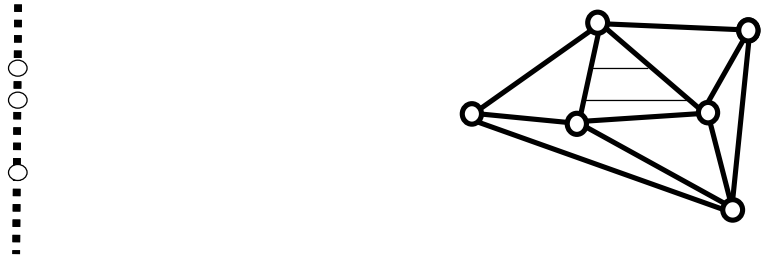
Idea: Keep a list of intersections for each pixel.



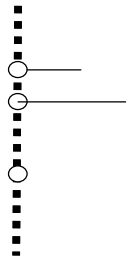
Cell-Projection With An A-Buffer

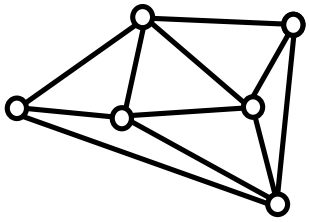


Cell-Projection With An A-Buffer

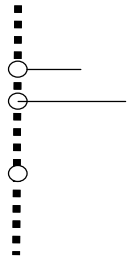


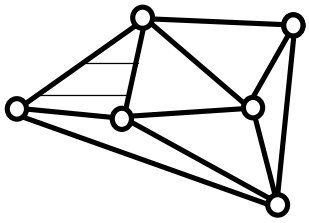
Cell-Projection With An A-Buffer





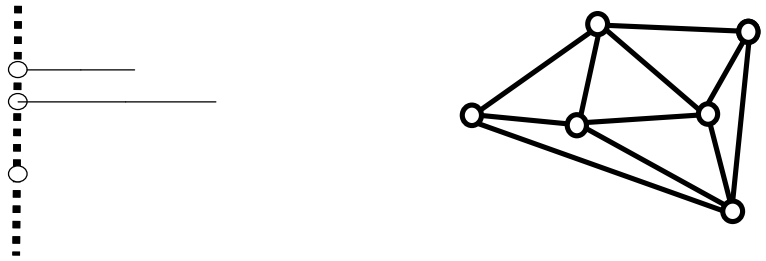
Cell-Projection With An A-Buffer



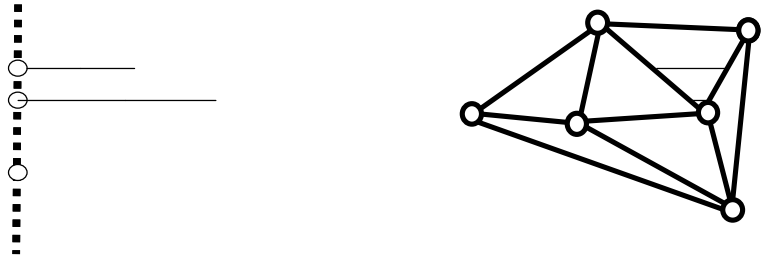


SC 2002
Tutorial M9

Cell-Projection With An A-Buffer

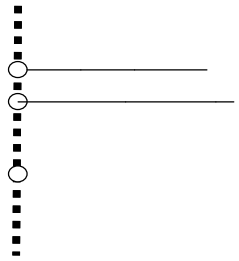


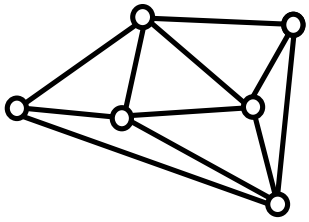
Cell-Projection With An A-Buffer



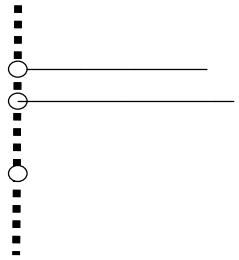
SC 2002
Tutorial M9

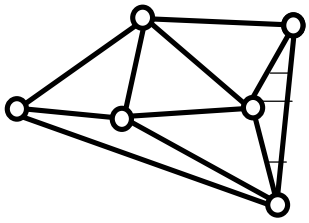
Cell-Projection With An A-Buffer





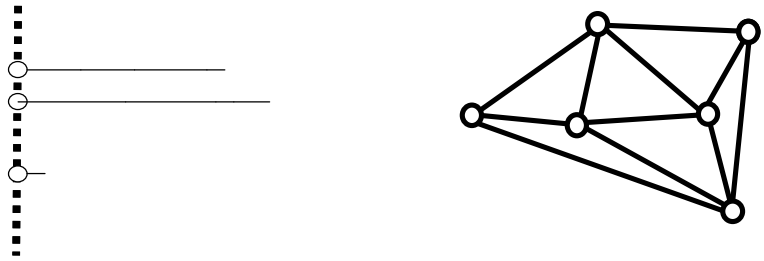
Cell-Projection With An A-Buffer



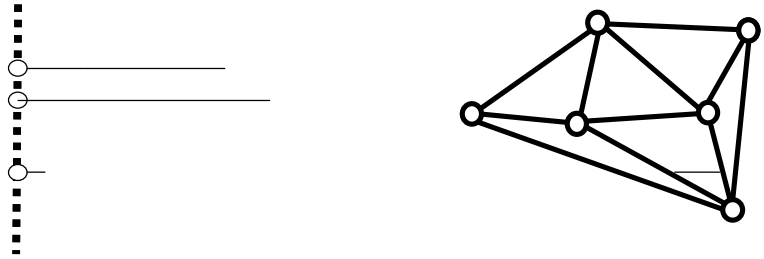


SC 2002
Tutorial M9

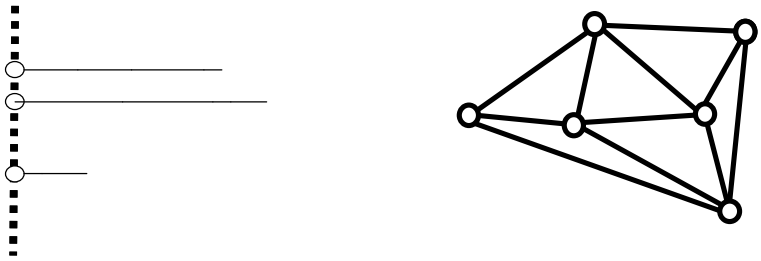
Cell-Projection With An A-Buffer



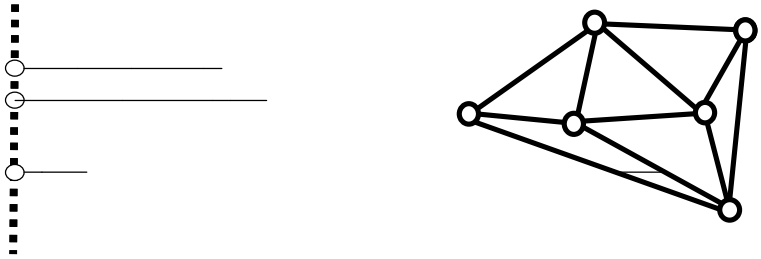
Cell-Projection With An A-Buffer



Cell-Projection With An A-Buffer

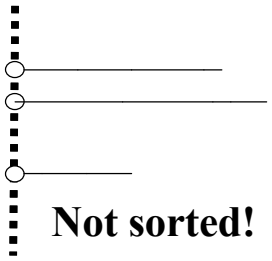


Cell-Projection With An A-Buffer

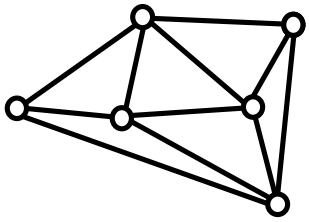


SC 2002
Tutorial M9

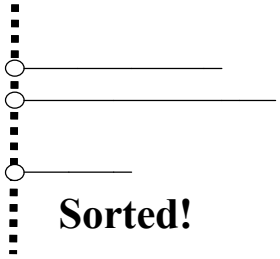
Cell-Projection With An A-Buffer



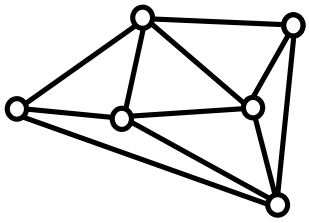
Not sorted!



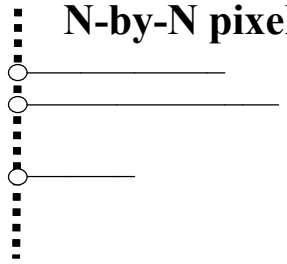
Cell-Projection With An A-Buffer

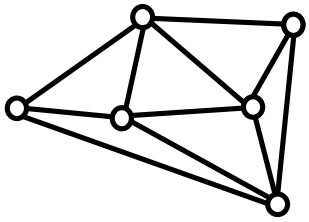


Sorted!



A-Buffer Limitations





Number of Intersections: $O(c\ N)^2$

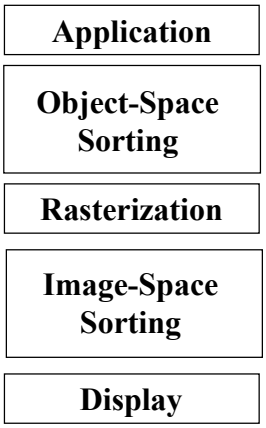
Problems:

1. **Time:** sorting intersections takes too long!
2. **Memory:** storage too high!

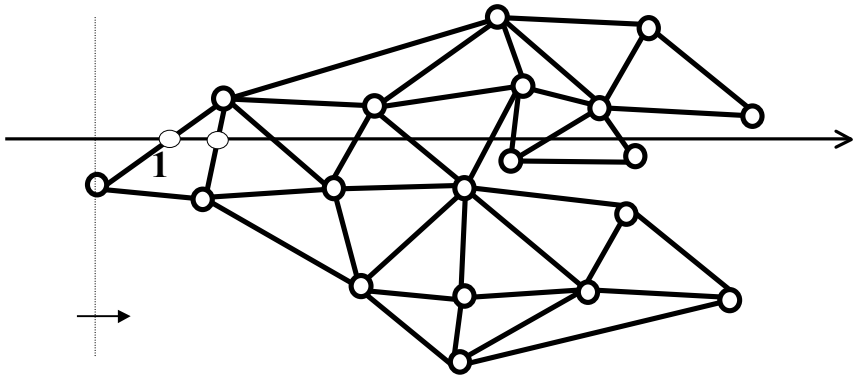
A-Buffer Hardware Designs

- **Wittenbrink's F-Buffer, GHW01**
  - → Proposes to use d^2 passes by using hardware-based *selection sort*
- **Compaq Research's Neon, GHW00**
  - → Limited to fixed depth

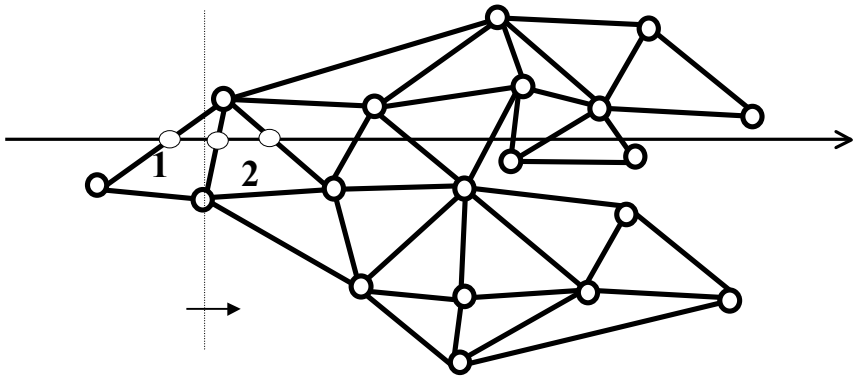
Sorting



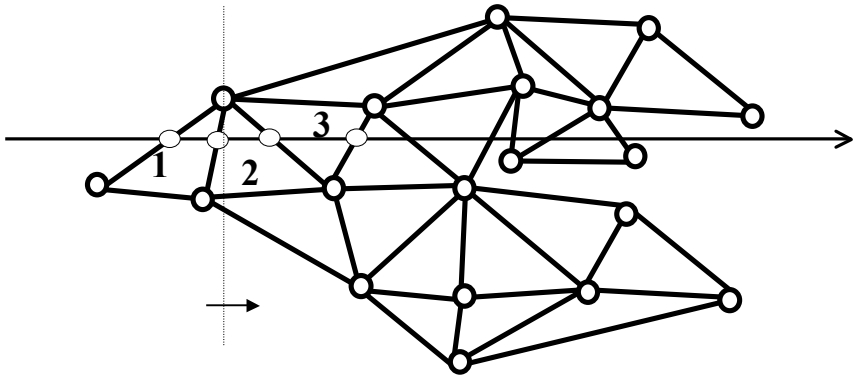
Approximate Object-Space Sorting



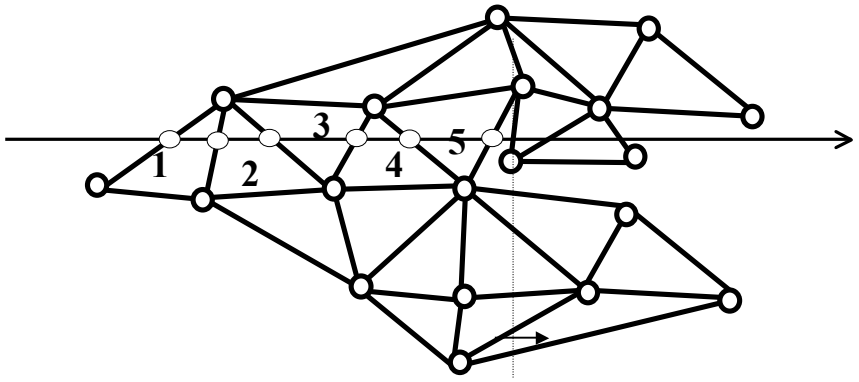
Approximate Object-Space Sorting



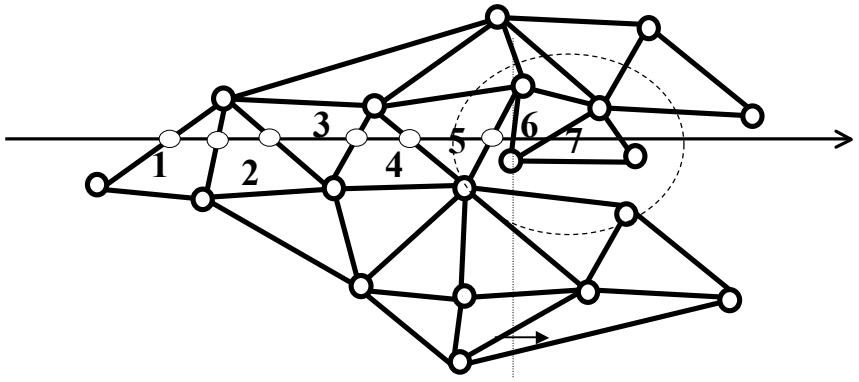
Approximate Object-Space Sorting



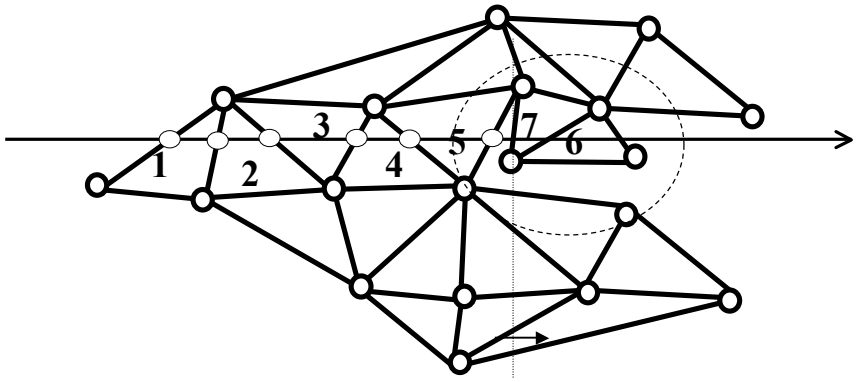
Approximate Object-Space Sorting



Approximate Object-Space Sorting

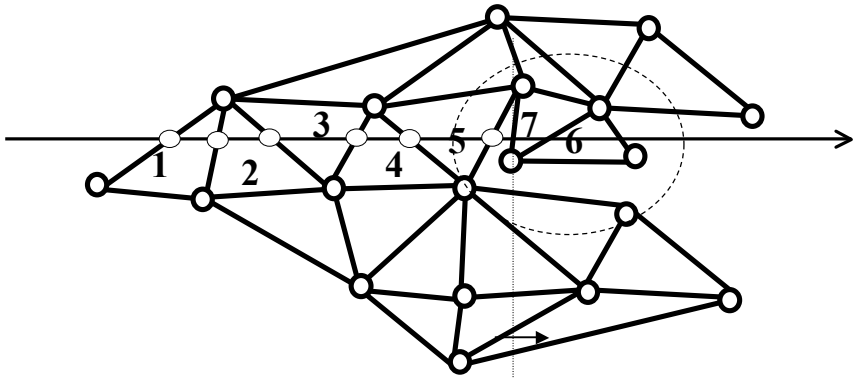


Approximate Object-Space Sorting



A Solution: Use an insertion-sort A-buffer!

Approximate Object-Space Sorting



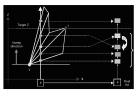
What about the space problem ?

→ Use a conservative bound on the intersections

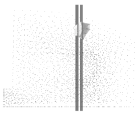
ZSWEEP:
Sort in Image and Object Space

[Farias, Mitchell, Silva 2000]

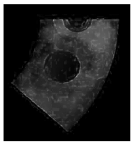
Do an approximate sorting of the cells (e.g., based on the order of the vertices), and use an insertion sort for the intersections – solves the sorting overhead



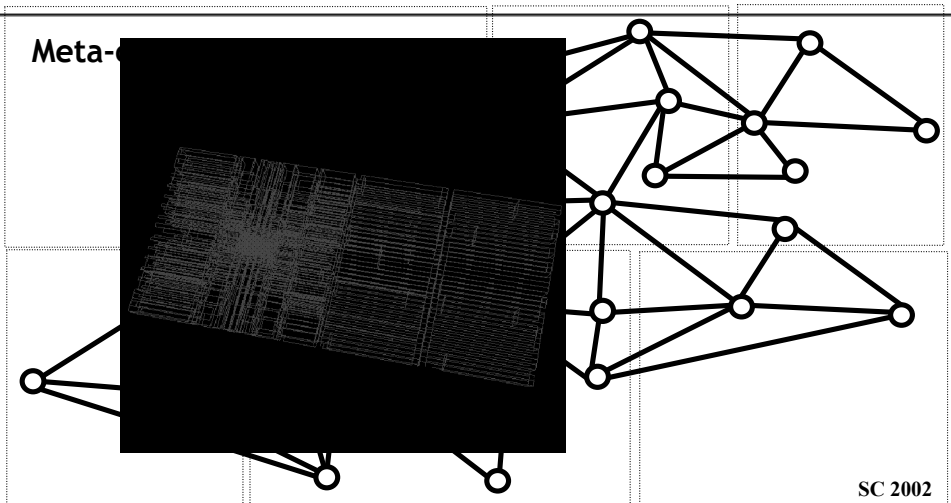
Propose a conservative technique for limiting the memory usage, by performing compositing on the intersections, anytime the lists become "too large" – solves the memory overhead



ZSWEEP - DEMO

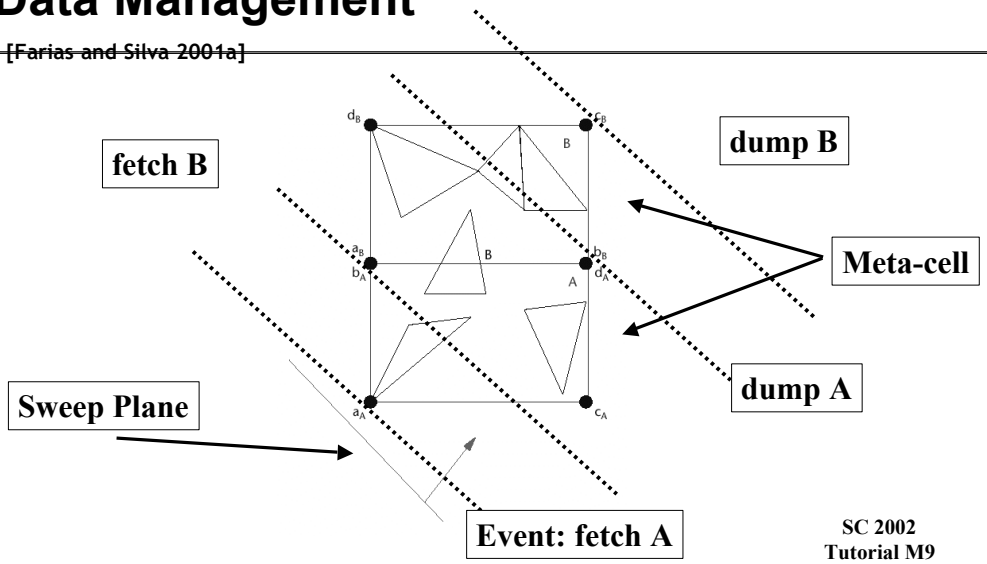


Blocking for Out-Of-Core Rendering



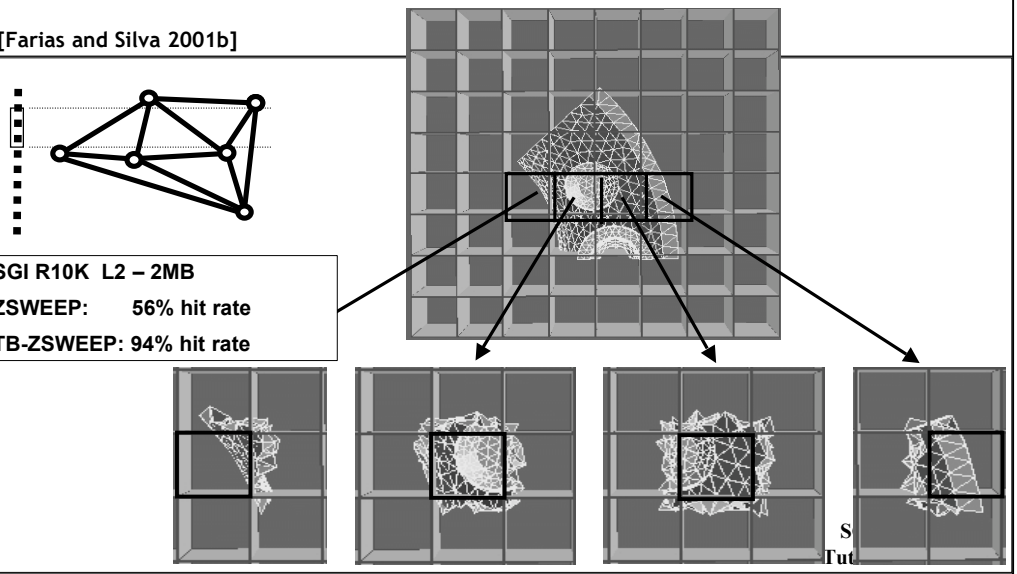
Out-Of-Core ZSWEEP: Data Management

[Farias and Silva 2001a]



Tile-Based ZSWEEP

[Farias and Silva 2001b]



SGI R10K L2 – 2MB
ZSWEEP: 56% hit rate
TB-ZSWEEP: 94% hit rate

S
Tut

Out-Of-Core ZSWEEP Results

Generating 2048x2048 Image (sec)				
	Original ZSWEEP		OOC-ZSWEEP	
	Memory	Time	Memory	Time
Blunt Fin	330	386	6.2	331
Combustion Chamber	330	407	6.2	423
Oxygen Post	350	775	13.2	667
Delta Wing	380	639	24.2	537

SC 2002
Tutorial M9

Acknowledgments

- **DOE/MICS and Sandia National Laboratory**
- **NSF**

Interactive Out-Of-Core Isosurface Extraction

Yi-Jen Chiang* Polytechnic University
Cláudio T. Silva† IBM T. J. Watson Research Center
William J. Schroeder‡ Kitware

Abstract

In this paper, we present a novel *out-of-core* technique for the interactive computation of isosurfaces from volume data. Our algorithm minimizes the main memory and disk space requirements on the visualization workstation, while speeding up isosurface extraction queries. Our overall approach is a *two-level indexing* scheme. First, by our *meta-cell* technique, we partition the original dataset into clusters of cells, called *meta-cells*. Secondly, we produce *meta-intervals* associated with the meta-cells, and build an indexing data structure on the meta-intervals. We *separate* the cell information, kept only in meta-cells in disk, from the indexing structure, which is also in disk and only contains pointers to meta-cells. Our meta-cell technique is an I/O-efficient approach for computing a k-d-tree-like partition of the dataset. Our indexing data structure, the *binary-blocked I/O interval tree*, is a new I/O-optimal data structure to perform *stabbing queries* that report from a set of meta-intervals (or intervals) those containing a query value q. Our tree is simpler to implement, and is also more space-efficient in practice than the existing structures. To perform an isosurface query, we first query the indexing structure, and then use the reported meta-cell pointers to read from disk the *active* meta-cells intersected by the isosurface. The isosurface itself can then be generated from active meta-cells. Rather than being a single-cost indexing approach, our technique exhibits a *smooth trade-off* between query time and disk space.

Keywords: Isosurface Extraction, Marching Cubes, Out-Of-Core Computation, Interval Tree, Scientific Visualization.

1 Introduction

Isosurface extraction represents one of the most effective and widely used techniques for the visualization of volume datasets. Formally, a *scalar volume dataset* consists of tuples $(\mathbf{x}, \mathcal{F}(\mathbf{x}))$, where $\mathbf{x}$ is a 3D point and $\mathcal{F}$ is a scalar function defined over 3D points. Given an isovalue q, extracting the isosurface of q is to compute the isosurface $C(q) = \{\mathbf{x} | \mathcal{F}(\mathbf{x}) = q\}$. The computation process can be divided into two phases: First, one finds the *active* cells that are intersected by the isosurface (the *search phase*), and then, one can compute the isosurface from the active cells (the *generation phase*). Most of the isosurface algorithms require the entire dataset to be kept in main memory, which is a severe limitation on their applicability, especially for large scientific applications.

In this paper, we present an isosurface technique whose main memory and disk space requirements on the visualization workstation are minimized, while speeding up the isosurface extraction procedure. In the same flavor as the methods of [10, 11], we index the dataset cells to achieve output-sensitive searches. Also, as in [10, 11], we keep both the indices (*i.e.*, intervals obtained from the cells) and the original dataset in *disk*, rather than in main memory. Moreover, during isosurface queries only a small portion of the dataset is touched and brought to main memory, by performing (using an indexing data structure) *stabbing queries* that report from a set of intervals those containing the query value q.

In [10, 11], to avoid inefficient *pointer references* in disk, the *direct cell information* is stored with its interval, in the indexing data structure. This is very inefficient in disk space, since the vertex information is duplicated many times, once for each cell sharing the vertex. Moreover, in the indexing structures [3, 18] used, each interval is stored three times in practice, increasing the duplications of vertex information by another factor of three. To eliminate this inefficiency, our indexing scheme uses a *two-level* structure. First, we partition the original dataset into clusters of cells, called *meta-cells*. Secondly, we produce *meta-intervals* associated with the meta-cells, and build our indexing data structure on the meta-intervals. We *separate* the cell information, kept only in meta-cells in disk, from the indexing structure, which is also in disk and only contains pointers to meta-cells. Isosurface queries are performed by first querying the structure, then using the reported meta-cell pointers to read from disk the *active* meta-cells intersected by the isosurface, which can then be generated from the active meta-cells.

While we need to perform *pointer references* in disk from the indexing structure to meta-cells, the *spatial coherences* of isosurfaces and of our meta-cells ensure that each meta-cell being read contains *many* active cells, so such pointer references are efficient. Also, a meta-cell is always read as a whole, hence we can use pointers *within* a meta-cell to store each meta-cell compactly. In this way, we obtain efficiencies in *both* query time and disk space. Two new techniques lie at the heart of this paper. One is the *meta-cell* technique that computes the spatially coherent meta-cells. The other is the *binary-blocked I/O interval tree*, a new I/O-optimal stabbing-query data structure that is simpler to implement and more space-efficient in practice than those in [3, 18]. We believe both techniques will find applications other than efficient out-of-core isosurface extraction.

We summarize the contributions of this work as follows.

- We present a novel out-of-core isosurface technique that improves [10, 11]. While keeping the querying time and main memory requirement small, the disk space overhead is reduced by more than one order of magnitude.
- We give a new *meta-cell* technique that partitions a volume dataset into spatially coherent meta-cells. This can be viewed as an out-of-core k-d-tree-like partition, and is efficiently carried out by performing external sorting a few times.
- We propose the *binary-blocked I/O interval tree*, a new I/O-optimal stabbing-query data structure. Previous such structures [3, 18] both have three types of secondary lists, but our tree has only two types of lists (as in the original main memory interval tree of [14]), so it has the tree size reduced by a factor of 2/3 in practice, and is also simpler to implement.

Previous Related Work

We first briefly review the work on out-of-core, or *I/O* techniques. In addition to early work on sorting and scientific computing, recently there have been I/O algorithms for graphs and for computational geometry; see [10, 11] for the references. Although most

*yjc@photon.poly.edu
†csilva@watson.ibm.com
‡william.schroeder@kitware.com

of the results are theoretical, the experiments of Chiang [8], Vengroff and Vitter [27], and Arge *et al.* [2] on some of these techniques show that they result in significant improvements over traditional algorithms in practice. Teller *et al.* [24] describe a system to compute radiosity solutions for polygonal environments larger than main memory, and Funkhouser *et al.* [15] present prefetching techniques for interactive walk-throughs in large architectural virtual environments. Very recently, Pharr *et al.* [21] give memory-coherent ray-tracing algorithms, Cox and Ellsworth [13] present application-controlled demand paging methods, and Ueng *el al.* [25] propose out-of-core streamline techniques.

As for isosurface extraction, there is a very rich literature. Here we only briefly review the results that focus on speeding up the search phase. We let N denote the number of cells in the dataset, and K the number of active cells. In Marching Cubes [20], all cells are searched for isosurface intersection, and thus $O(N)$ time is needed. Techniques avoiding exhaustive scanning include using an octree [28], identifying a collection of *seed cells* and performing contour propagation from the seed cells [4, 17, 26], NOISE [19], and other nearly optimal isosurface extraction methods [23]. The first *optimal* isosurface extraction algorithm was given by Cignoni *et al.* [12], based on the following two ideas. First, for each cell, they produce an interval $I = [\min, \max]$ where $\min$ and $\max$ are the minimum and maximum of the scalar values in the cell vertices. Then the active cells are exactly those cells whose intervals contain q. Searching active cells then amounts to performing stabbing queries. Secondly, the stabbing queries are solved by using an internal-memory interval tree [14]. After an $O(N \log N)$-time preprocessing, active cells can be found in optimal $O(\log N + K)$ time.

The first *out-of-core* isosurface technique was given by Chiang and Silva [10]. They follow the ideas of Cignoni *et al.* [12], but use the I/O-optimal interval tree of [3] to solve the stabbing queries. In their follow-up paper [11], they replaced the I/O interval tree of [3] with the metablock tree [18]. With their techniques, datasets much larger than main memory can be visualized very efficiently. The major drawback is the large overhead in disk space to hold the search structure, and the disk scratch space needed to build the structure. Another out-of-core isosurface technique, based on contour propagation from seed cells, is recently proposed in [5] (where no out-of-core implementation is reported).

2 Main Techniques

In this section we present our isosurface algorithm. There are two major techniques: the *meta-cell* technique, which is used to construct *meta-cells* from dataset cells, and the *binary-blocked I/O interval tree*, which is a new I/O-optimal stabbing-query data structure, used to serve as an *indexing* structure for the meta-cells. We show the preprocessing pipeline of our overall algorithm in Fig. 1. The main tasks are as follows:

(1) Group spatially neighboring cells into *meta-cells*. The total number of vertices in each meta-cell is roughly the same, so that during queries each meta-cell can be retrieved from disk with approximately the same I/O cost. Each cell is assigned to exactly one meta-cell.

(2) Compute and store in disk the meta-cell information for each meta-cell.

(3) Compute *meta-intervals* associated with each meta-cell. Each meta-interval is an interval $[\min, \max]$, to be defined later.

(4) Build in disk a binary-blocked I/O interval tree on meta-intervals. For each meta-interval, only its $\min$ and $\max$ values and the meta-cell ID are stored in the tree, where meta-cell ID is a pointer to the corresponding meta-cell record in disk.

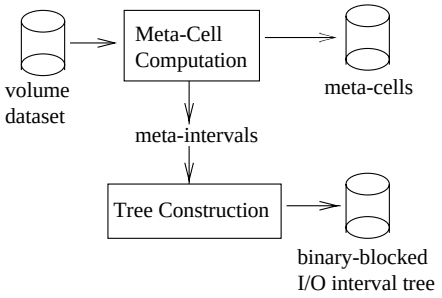


Figure 1: The preprocessing pipeline of our isosurface technique.

We describe the representation of meta-cells. Each meta-cell has a list of vertices, where each vertex entry contains its x-, y-, z- and scalar values, and a list of cells, where each cell entry contains pointers to its vertices in the vertex list. In this way, a vertex shared by many cells in the same meta-cell is stored just *once* in that meta-cell. The only duplications of vertex information occur when a vertex belongs to two cells in *different* meta-cells; in this case we let both meta-cells include that vertex in their vertex lists, so that each meta-cell has *self-contained* vertex and cell lists. We store the meta-cells, one after another, in disk.

The purpose of meta-intervals for a meta-cell is analogous to that of interval for a cell: a meta-cell is *active*, *i.e.*, intersected by the isosurface of q, if and only if one of its meta-intervals contains q. Intuitively, we could just take the minimum and maximum scalar values among the vertices to define the meta-interval (as cell intervals), but such big range would contain *gaps*[*] in which no cell interval lies. Therefore, we break such big range into pieces, each a meta-interval, by the gaps. Formally, we define the *meta-intervals* of a meta-cell as the *connected components* among the intervals of the cells in that meta-cell. With this definition, searching active meta-cells amounts to performing stabbing queries on the meta-intervals. The query pipeline of our overall algorithm is shown in Fig. 2. We have the following steps:

(1) Find all meta-intervals (and the corresponding meta-cell ID's) containing q, by querying the binary-blocked I/O interval tree in disk.

(2) (Internally) sort the reported meta-cell ID's. This makes the subsequent disk reads for active meta-cells *sequential* (except for skipping inactive meta-cells), and minimizes the disk-head movements.

(3) For each active meta-cell, read it from disk to main memory, identify active cells and compute isosurface triangles, throw away the current meta-cell from main memory and repeat the process for the next active meta-cell. At the end, patch the generated triangles and perform the remaining operations in the generation phase to generate and display the isosurface.

Now we argue that in step (3) the pointer references in disk to read meta-cells are efficient, *i.e.*, there are many active cells in an active meta-cell. Intuitively, by the way we construct the meta-cells, we can think of each meta-cell as a cube, with roughly the same number of cells in each dimension. Also, by the *spatial coherence* of an isosurface, usually there are not many meta-cells that

[*]Gaps only occur when disconnected components of cells belong to the same meta-cell.

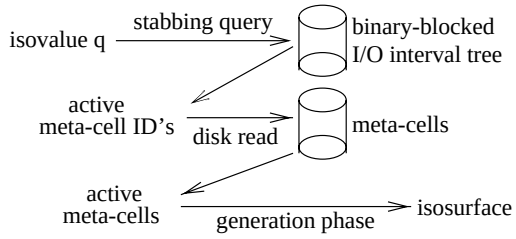


Figure 2: The query pipeline of our isosurface technique.

are cut *only through corners* by the isosurface. Thus by a dimension argument, if an active meta-cell has C cells, for most times the isosurface cuts through $C^{2/3}$ cells. This is similar to the argument that usually there are $\Theta(N^{2/3})$ active cells in an N-cell volume dataset. Then this means that we read C cells (a whole meta-cell) for every $C^{2/3}$ active cells, *i.e.*, we traverse a *thickness* of $C^{1/3}$ layers of cells, for one layer of isosurface. Therefore we read $C^{1/3} \cdot (K/B)$ disk blocks for K active cells, which is a factor of $C^{1/3}$ from optimal (B is the number of cells fitting in one disk block). Notice that when the size of meta-cells is increased, the number of duplicated vertices is decreased (less vertices in meta-cell boundaries), and the number of meta-intervals is also decreased (less meta-cells), while the number C is increased. Hence we have a *trade-off* between space and query time, by varying the meta-cell size. Since the major cost in disk reads is in *disk-head movements* (*e.g.*, reading two disk blocks takes approximately the same time as reading one block, after moving the disk head), we can increase meta-cell sizes while keeping the effect of the factor $C^{1/3}$ negligible. (We shall see the actual trade-off between disk space and query time when we present the experimental results in Section 3.)

2.1 Meta-Cell Computation

The efficient subdivision of the dataset into meta-cells lies at the heart of our overall isosurface algorithm. The computation is similar to the partition induced by a k-d-tree [6], but we do not need to compute the multiple levels. Since direct random access to vertices is very inefficient in disk, we develop a new technique that is I/O-efficient, by essentially performing external sorting a few times. We assume that the input dataset is in a general "index cell set" (ICS) format, *i.e.*, there is a list of vertices, each containing its x-, y-, z- and scalar values, and a list of cells, each containing pointers to its vertices in the vertex list. We want to partition the dataset into H^3 meta-cells, where H is a parameter we can adjust to vary the meta-cell sizes, usually several disk blocks. The final output of meta-cell computation is a single file that contains all meta-cells, one after another, each an *independent* ICS file (*i.e.*, the pointer references from cells of a meta-cell are *within* the meta-cell). We also produce meta-intervals for each meta-cell.

For simplicity, we assume that the input cell list contains cells of the same type (*e.g.*, tetrahedral cells). If this is not the case, we can first scan the cell list and put different types of cells into different cell lists. In the following, we refer to meta-cell ID's as numbers $0, 1, \cdots$ to number the meta-cells; we refer to them as *pointers* to the meta-cell positions in disk, as we previously do, only after the meta-cell computation is complete. Our meta-cell computation consists of the following steps.

1. Partition vertices into clusters of equal size. This is the *key* step in constructing meta-cells. We use each resulting cluster to define a meta-cell, whose vertices are those in the cluster, plus some *duplicated* vertices to be constructed later. Observe that meta-cells may differ dramatically in their volumes, but their numbers of vertices are roughly the same. The partitioning method is very simple. We first externally sort all vertices by the x-values, and partition them into H consecutive chunks. Then, for each such chunk, we externally sort its vertices by the y-values, and partition them into H chunks. Finally, we repeat the process for each refined chunk, except that we externally sort the vertices by the z-values. We take the final chunks as clusters. Clearly, each cluster has spatially neighboring vertices. The computing cost is bounded by three passes of external sorting. This step actually *assigns* vertices to meta-cells. We produce a *vertex-assignment* list with entries (v_{id}, m_{id}), indicating that vertex v_{id} is assigned to meta-cell m_{id}.

2. Assign cells to meta-cells and duplicate vertices. Our assignment of cells to meta-cells attempts to minimize the wasted space. The basic coverage criterion is to see how a cell's vertices have been mapped to meta-cells. A cell whose vertices all belong to the same meta-cell is assigned to that meta-cell. Otherwise, the cell is in the boundary, and a simple voting scheme is used: the meta-cell that contains the *most* vertices owns that cell, and the *missing* vertices of the cell have to be duplicated and inserted to this meta-cell. We break ties arbitrarily. In order to determine this assignment, we need to obtain for each cell, the destination meta-cells of its vertices. For in-core computation, this is easily computed by a pointer de-reference. But the out-of-core counterpart of this computation is not so simple. Our basic operation is the *join* operation (commonly used in database), using the vertex ID as the *key*, in both the cell list and the vertex-assignment list. The join operation can be performed I/O-efficiently, by externally sorting both lists by the key, and scanning through both lists to fill in the information needed [7, 9]. For example, to fill in the destination meta-cell ID of the *first* vertex in each cell, we sort the cell records in the cell list by the vertex ID's of their *first* vertices, so that the first group contains the cells whose first vertices are vertex 1, the second group contains the cells whose first vertices are vertex 2, and so on. We also sort the vertex-assignment list by vertex ID, so that we know the destination meta-cell ID's of vertex 1, of vertex 2, etc., in that sequential order. We then scan through both lists and fill in the destination meta-cell ID of the first vertex, for each cell in the cell list. We need to perform as many join operations as the degree of the cell (*i.e.*, for tetrahedra we need to perform four joins). Once all the vertex-to-meta-cell assignments have been propagated to the cell list, a single scan is enough not only to assign cells to meta-cells, but also to decide which vertices to duplicate and insert to which meta-cells. For the latter, we produce a *vertex-duplication* list with entries (v_{id}, m_{id}), indicating that vertex v_{id} has to be duplicated and inserted to meta-cell m_{id}.

3. Compute the vertex and cell lists for each meta-cell. To actually duplicate vertices and insert them to appropriate meta-cells, we first need to de-reference the vertex ID's (to obtain the *complete* vertex information) from the vertex-duplication list. We can do this by using one join operation, using vertex ID as the key, on the original input vertex list and the vertex-duplication list. Now the vertex-duplication list contains for each entry the complete vertex information, together with the ID of the meta-cell to which the vertex must be inserted. We also have a list for assigning cells to meta-cells. To finish the generation of meta-cells, we use a main join operation on these lists, using meta-cell ID as the main key. To avoid possible replications of the same vertex inside a meta-cell, we use vertex ID's as the secondary key during the sorting for the join operation. Finally, we update the vertex pointers for the cells *within* each meta-cell. This can be easily done since each meta-cell can be kept in the main memory.

4. Compute meta-intervals for each meta-cell. Since each meta-cell can fit in main memory, this step only involves in-core computation. First, we compute the interval for each cell in the meta-cell. Then we sort all interval endpoints. We scan through the endpoints, with a counter initialized to 0. A left endpoint encountered increases the counter by 1, and a right endpoint decreases the

counter by 1. A "$0 \rightarrow 1$" transition gives the beginning of a new meta-interval, and a "$1 \rightarrow 0$" transition gives the end of the current meta-interval. We can easily see that the computation is correct, and the computing time is bounded by that of internal sorting.

2.2 Binary-Blocked I/O Interval Tree

Now we present our *binary-blocked I/O interval tree*. Since it is a general stabbing-query data structure, we use the general term *interval* to refer to the underlying intervals or meta-intervals being manipulated. We use *unique cell ID's* to break a tie between endpoint values. In the case of meta-intervals and meta-cells, it is easy to see that each entry of (endpoint value, meta-cell ID) is distinct. We use N to denote the total number of intervals considered, and M and B the numbers of intervals fitting in main memory and in one disk block, respectively. One I/O operation reads or writes one disk block.

Our interval tree is I/O-optimal in space, query, and preprocessing, and is an extension of the original (main memory, binary) interval tree of [14]. Our *branching factor* Bf (*i.e.*, the maximum number of children of an internal node) is increased from 2 to $\Theta(B)$, to reduce the tree height from $O(\log_2 N)$ to $O(\log_B N)$, like B-trees. We remark that the previous I/O-optimal interval tree of [3] also increases Bf (to $\Theta(\sqrt{B})$) to make tree height $O(\log_B N)$, but an additional type of secondary lists is introduced, which potentially increases the space by a factor of 3/2 (originally the binary interval tree has two types of secondary lists). Our tree does not introduce any new type of lists, so is simpler to implement and also is more space-efficient in practice.

2.2.1 Data Structure

Before describing our binary-blocked I/O interval tree, we first review the original (main memory) interval tree of [14]. Given a set of N intervals, such interval tree T is defined recursively as follows. If there is only one interval, then the current node r is a leaf containing that interval. Otherwise, node r stores as a key the median value m that partitions the interval endpoints into two slabs, each having the same number of endpoints that are smaller (resp. larger) than m. The intervals that contain m are assigned to node r. The intervals with both endpoints smaller than m are assigned to the left slab; similarly, the intervals with both endpoints larger than m are assigned to the right slab. The left and right subtrees of r are recursively defined as the interval trees on the intervals in the left and right slabs, respectively. In addition, each internal node u of T has two secondary lists: the *left list*, which stores the intervals assigned to u, sorted in *increasing left endpoint values*, and the *right list*, which stores the same set of intervals, sorted in *decreasing right endpoint values*. It is easy to see that the tree height is $O(\log_2 N)$. Also, each interval is assigned to exactly one node, and is stored either twice (when assigned to an internal node) or once (when assigned to a leaf), and thus the overall space is $O(N)$.

In our binary-blocked I/O interval tree, $\mathcal{T}$, each node is one disk block, capable of holding B items. We want to increase the branching factor Bf so that the tree height is $O(\log_B N)$. The intuition of our method is extremely simple: we *block* a subtree of the binary interval tree T into one node of $\mathcal{T}$ (see Fig. 3). In the following, we refer to the nodes of T as *small nodes*. We take the branching factor Bf to be $\Theta(B)$. Then in an internal node of $\mathcal{T}$, there are $Bf-1$ small nodes, each having a key, a pointer to its left list and a pointer to its right list, where all left and right lists are stored in disk.

Now we give a more formal definition of tree $\mathcal{T}$. First, we sort all *left* endpoints of the N intervals in increasing order from left to right, into set E. We use (meta-)cell ID's to break ties. Set E is used to define the keys in small nodes. Then tree $\mathcal{T}$ is recursively defined as follows. If there are no more than B intervals, then the current node u is a leaf node storing all intervals. Otherwise, u is

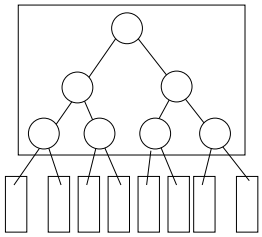


Figure 3: Intuition of binary-blocked I/O interval tree $\mathcal{T}$: each circle is a node in the binary interval tree T, and each rectangle, which blocks a subtree of T, is a node of $\mathcal{T}$.

an internal node. We take $Bf-1$ median values from E, which partition E into Bf slabs, each with the same number of endpoints. We store sorted, in non-decreasing order, these $Bf-1$ median values in node u, which serve as the keys of the $Bf-1$ small nodes in u. We *implicitly* build a subtree of T on these $Bf-1$ small nodes, by a *binary-search scheme*: the root key is the median of the $Bf-1$ sorted keys, the key of the left child of the root is the median of the lower half keys, and the right-child key is the median of the upper half keys, and so on. Now consider the intervals. The intervals that contain one or more keys of u are assigned to u. In fact, each such interval I is assigned to the *highest* small node (in the subtree in u) whose key is contained in I; we store I in the corresponding left and right lists of that small node. For the remaining intervals, each has both endpoints in the same slab and is assigned to that slab. We recursively define the Bf subtrees of node u as the binary-blocked I/O interval trees on the intervals in the Bf slabs.

Notice that with the above binary-search scheme for implicitly building a (sub)tree on the keys stored in an internal node u, Bf does not need to be a power of 2 — we can make Bf as large as possible, as long as the $Bf-1$ keys, the $2(Bf-1)$ pointers to the left and right lists, and the Bf pointers to the children, etc., can all fit into one disk block. As a comparison, in the I/O interval tree of [3], each internal node has $\Theta(Bf)$ left lists, $\Theta(Bf)$ right lists, and additional $\Theta(Bf^2)$ *multi* lists, and thus Bf is taken as $\Theta(\sqrt{B})$. Also, an interval can be stored up to three times. It is easy to see that our tree $\mathcal{T}$ has height $O(\log_B N)$, and the overall space complexity is optimal $O(N/B)$ disk blocks.

2.2.2 Query Algorithm

Our query algorithm for the binary-blocked I/O interval tree $\mathcal{T}$ is very simple and mimics the query algorithm for the binary interval tree T. Given a query point q, we perform the following recursive process starting from the root of $\mathcal{T}$. For the current node u, we read u from disk. Now consider the subtree T_u implicitly built on the small nodes in u by the binary-search scheme. Using the same binary-search scheme, we follow a root-to-leaf path in T_u. Let r be the current small node of T_u being visited, with key value m. If $q = m$, then we report all intervals in the left (or equivalently, right) list of r and stop. If $q < m$, we scan and report the intervals in the left list of r, until the first interval with left endpoint larger than q is encountered. Recall that the left lists are sorted by increasing left endpoint values. After that, we proceed to the left child of r in T_u. Similarly, if $q > m$, we scan and report the intervals in the right list of r, until the first interval with right endpoint smaller than q is encountered. Then we proceed to the right child of r in T_u. At the end, if q is not equal to any key in T_u, the binary search on the $Bf-1$ keys locates q in one of the Bf slabs. We then visit the child node of u in $\mathcal{T}$ which corresponds to that slab, and apply the same process recursively. Finally, when we reach a leaf node of $\mathcal{T}$, we check the $O(B)$ intervals stored to report those that contain q, and stop. Although the tree height is $O(\log_B N)$, in the worst-

case we might need to perform a total of $O(\log_2(N/B) + K/B)$ I/O operations for a query. We can improve this bound to optimal $O(\log_B N + K/B)$ I/O's by using the *corner structures* [18]; we omit the details here in order to stay within the page limitations.

2.2.3 Preprocessing Algorithm

We describe our preprocessing algorithm for building the tree $\mathcal{T}$. It is based on the *scan and distribute* paradigm originated from the *distribution sweep* I/O technique [8, 16]. Our algorithm follows the definition of $\mathcal{T}$ given in Section 2.2.1. In the first phase, we sort (using external sorting) all N input intervals in increasing *left* endpoint values from left to right, into a set S. We use (meta-)cell ID's to break a tie. We also copy the *left* endpoints, in the same sorted order, from S to another set E. The set E is used to define median values to partition E into slabs throughout the process.

The second phase is a recursive process. If there are no more than B intervals, then we make the current node u a leaf, store all intervals in u and stop. Otherwise, node u is an internal node. We first take the $Bf-1$ median values from E that partition E into *Bf* slabs, each containing the same number of endpoints. We store sorted in u, in non-decreasing order from left to right, these median values as the keys in the small nodes of u. We now scan all intervals (from S) to distribute them to node u or to one of the *Bf* slabs. We maintain a temporary list for node u, and also a temporary list for each of the *Bf* slabs. For each temporary list, we keep one block in the main memory as a *buffer*, and keep the remaining blocks in disk. Each time an interval is distribute to node u or to a slab, we put that interval to the corresponding buffer; when a buffer is full, it is written to the corresponding list in disk. The distribution of each interval I is carried out by the *binary-search scheme* described in Section 2.2.1, which implicitly defines a balanced binary tree T_u on the $Bf-1$ keys and the corresponding small nodes in u. We perform this binary search on these keys to find the highest small node r whose key is contained in I, in which case we assign I to small node r (and also to the current node u), by appending the small node ID of r to I and putting it to the temporary list for node u, or to find that no such small node exists and both endpoints of I lie in the same slab, in which case we distribute I to that slab by putting I to the corresponding temporary list. When all intervals in S are scanned and distributed, each temporary list has all its intervals, automatically sorted in increasing left-endpoint values. Now we sort the intervals belonging to node u by small node ID as the first key and the left-endpoint value as the second key, in increasing order, so that intervals assigned to the same small node are put together, sorted in increasing left-endpoint values. We read these intervals to set up the left lists of all small nodes in u. Then we copy each such left list to its corresponding right list, and sort the right list by decreasing right-endpoint values. The corner structure for node u, if we want to construct, can be built at this point. This completes the construction of node u. Finally, we perform the process recursively on each of the *Bf* slabs, using the intervals in the corresponding temporary list as input, to build each subtree of node u.

We remark that in the above *scan and distribute* process, instead of keeping all intervals assigned to the current node u in *one* temporary list, we could maintain $Bf-1$ temporary lists for the $Bf-1$ small nodes of u. This would eliminate the subsequent sorting by small node ID's (which is used to *re-distribute* the intervals of u into individual small nodes). But for the actual implementation, our method is used to address the system issue that a process cannot open too many files simultaneously, while avoiding a blow-up in disk scratch space. It can be shown that the overall preprocessing takes nearly optimal $O(\frac{N}{B}\log_B N)$ I/O's. We can also make the bound optimal ($O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$, as the external sorting bound [1], where M is the number of intervals fitting in main memory) by the tree-height conversion method in [10].

3 Experimental Results and Analysis

In this section, we attempt to experimentally assess the advantages and shortcomings of our new technique, in particularly as compared to our previous work [10, 11]. We consider five datasets in our study. Four of them were used in our previous papers [10, 11], and a new, larger dataset, Cyl3 with about 5.8M cells has been added to our test set. Table 1 summarize their properties.

Our experimental set-up is similar to the one we used in [10, 11]. Our benchmark machine is an off-the-shelf PC: a Pentium Pro, 200MHz with 128M of RAM, and 768M of swap space. Using Linux, we booted the machine in two different configurations, with 64M and 128M of main memory. For preprocessing, we used the machine with only 64M of main memory, and for computing the isosurfaces we varied the amount of main memory. Because of the usage of the operating system and X-windows, we estimate that only half to two thirds of main memory was actually available for computations.

Meta-cell Generation

Computing the meta-cells is a core operation of our technique, and one of the main differences between our new method and [10, 11]. Meta-cell generation is basically divided into five parts: (1) normalizing the original file, which involves separating the vertices and each type of cells into their own files, (2) mapping the vertices into meta-cells, (3) mapping the cells into meta-cells, (4) completing the meta-cell information and writing to the meta-cell file, and (5) computing the meta-intervals used for indexing. As can be seen in Tables 2 and 3, meta-cell generation can be expensive, in particular for large datasets, such as Cyl3. The main reason for this is that we do not assume any kind of pre-determined spatial coherence in our input, forcing us to perform several *external sorts* on different *keys*, over very large files.

There are several ways to make this faster. The most obvious would be to use a larger machine with enough main memory for the computation. In this case, the geometric hashing we are using becomes trivial, and clearly can be performed very efficiently. A less obvious observation is that due to the fact that we are essentially performing a global geometric hashing operation, given information about the relative positions of the vertices (basically, rough bounding boxes), the computation can be performed more efficiently. For instance, if we already have some meta-cell subdivision, we do not need to recompute another one from scratch, instead it is possible to either refine a coarser subdivision, or join multiple fine subdivisions into coarser ones. We conjecture (though have not tried yet) that we should be able to manage multi-gigabyte scientific datasets computed in distributed memory parallel machines, by running our meta-cell generation on each piece individually, since, in general, they are organized in mostly disjoint chunks of spatially coherent data.

Tables 2 and 3 give some important performance statistics. In Table 2, a global view of the performance of our technique can be seen on four different datasets. It is interesting to note that by varying the number of meta-cells, we can effectively control the disk space overhead. In general, the smaller number of cells in a meta-cell, the faster the querying and fetching, and also the more accurate the isosurface search. In Table 3 we vary the number of meta-cells used for the Delta dataset. This table shows that our algorithm scales well with increasing meta-cell sizes. The most important feature is the linear dependency of the querying accuracy versus the disk space overhead. For example, using a total of 146 meta-cells (at 7% disk overhead), for a given isosurface, we need 3.34s to find the active cells. When using 30,628 meta-cells (at 63% disk overhead), we only need 1.18s to find the correct cells. Basically, the more meta-cells, the more accurate our active-cell searchers, and the less amount of data we need to fetch from disk.

Name	# of Cells	Original Size	Binary Size
Blunt Fin	187K	5.8M	3.7M
Comb. Chamber	215K	6.8M	4.2M
Liquid Oxygen Post	513K	16.4M	10M
Delta Wing	1M	33.8M	19.4M
Cyl3	5.8M	337M	152M

Table 1: A list of the datasets used for testing. Original size is the file size as an ASCII ".scalar" or ".vtk" file.

	Blunt	Chamber	Post	Cyl3
# of meta-cells	737	1009	1870	27896
Normalization	3.1s	3.5s	8.8s	158s
Vertex Map	2.8s	3.6s	8.3s	382s
Cell Map	19s	24.1s	58.1s	783s
Meta-Cell Info	20.8s	24s	67.8s	1179s
Meta-Intervals	4.2s	4.8s	11.7s	147s
Total	50s	60s	154.8s	3652s
Original Size	3.65M	4.19M	10M	152M
Meta-Cell Size	4.39M	5M	12.2M	271M
Avg Vertex	118.1	102.1	133.2	399
Avg Cell	254.2	213.1	274.5	208
Increase	20%	21%	22%	78%
BBIO_Tree (size)	29K	28K	84K	1.7M
BBIO_Tree (time)	0.35s	0.67s	1.23s	43s

Table 2: Statistics for preprocessing isosurfaces on different datasets. First, we show the number of meta-cells used for partitioning the dataset, followed by the times for each step of the meta-cell computation and its total time. Secondly, the original dataset size and the size of the meta-cell file are shown. We also show the average numbers of vertices and of cells per meta-cell, and the overall increase in storage. Finally, we show the size (in bytes) of the BBIO tree and its construction time.

An interesting point is that the more data fetched, the more work (and main memory usage) for the isosurface generation engine. By paying the 63% disk overhead, we only need to fetch 16% of the dataset into main memory, which is clearly a substantial saving.

Figs. 4a and 5a show the bounding boxes of two meta-cell decompositions on the same dataset. The dataset used was a low resolution version of the dataset Cyl3 used in Tables 2 and 4 to avoid cluttering. One can see from the two figures that our algorithm samples the higher-resolution areas with more meta-cells, while using lower numbers of meta-cells in areas with less details.

Meta-cell Indexing

The number of meta-intervals generated is directly proportional to the number of meta-cells. The size of the interval tree (denoted by BBIO tree) increases when the dataset gets larger (*e.g.*, for the Cyl3 dataset shown in Table 2 is 1.7M), and may be well beyond the main memory size for larger dataset. This is the major reason why we need the BBIO tree, to ensure the scalability for a large number of meta-intervals being indexed. In addition, as opposed to in-core indexing structures, we need not spend the time to build/load the tree in main memory every time the process starts to run. Tables 2 and 3 also contain information related to the construction of the trees, and their respective sizes. Having the indexing data structure separated from the meta-cells is important, since in several applications multiple indexing structures can point to the same set of meta-cells. For instance, in handling time-varying datasets, one can keep a single copy of the geometric data (in the meta-cells), and have multiple BBIO trees for indexing different time steps.

Isosurface Extraction Queries

Table 3 already presents some limited querying information that demonstrates the effectiveness of the meta-cell blocking as a function of the disk space overhead. Particularly interesting are the data given in Table 3, which shows how the isosurface extraction cost changes with meta-cell sizes. As the number of meta-cells increases (and the disk space overhead also increases due to more vertex replications), the query time decreases. This shows that our technique provides a smooth trade-off between disk space overhead and querying performance. A visual representation of this effect can be seen from Figs. 4b and 5b, which show the bounding boxes of the fetched (*i.e.*, active) meta-cells during the query of the isosurface with value 0.0623775 in the Cyl dataset. Figs. 4c and 5c show the actual isosurfaces superimposed to the active meta-cells. Even for this down-sampled dataset and the coarse meta-cells, one can see the effect of more meta-cells in culling away larger portions of the dataset not containing the isosurface. Note the difference between Figs. 4b and 5b in the middle of the dataset where the cells do not get touched. As the number of meta-cells increases, the active meta-cells are refined and resemble the isosurface.

It is important to study the overall performance of the isosurface extraction query pipeline. Ideally, we would like to compare four different techniques: (1) the plain Vtk [22] pipeline; (2) an output-sensitive in-core isosurface algorithm (such as the one presented in [12]); (3) our previous work [10, 11]; (4) our new algorithm. Unfortunately, we do not have (2)[†]. With respect to the comparisons with (3) [10], we will not be able to compare for the Cyl3 dataset, since we would need over 2.4GB of disk to perform the preprocessing (and several hours).

Table 4 summarizes our benchmarks. Points worth noting:

- Our previous technique, `ioQuery` [10], performs better than both `mcQuery` (our new code) and `vtkIso` (the pure Vtk code) in all cases. This is not really a surprise, since `ioQuery` performs an exact search, only bringing active cells into main memory. Thus, it does not waste either disk bandwidth or main memory space. Unfortunately, as we pointed out before, `ioQuery` is not practical, since it uses about 8 times as much disk space as the original dataset to keep the search structure, and it needs 16 times as much disk scratch space for preprocessing.

- Our new querying code, `mcQuery`, performs better than `vtkIso` for most examples. In particular, for Cyl3, it is over 20 times faster than pure Vtk, and even in cases where there is enough main memory such as for the Delta dataset, with only 63% disk overhead, it is about five times faster than Vtk. In fact, in some cases (such as for Post and Delta), we are able to finish querying while Vtk is still reading the dataset.

One last note about the implementation. Some might be wondering how come Vtk needs so much main memory to compute isosurfaces. In fact, it might require two to three times as much main memory as the original dataset. Without further study, we can only speculate. There are several main memory overheads for isosurface calculation, besides the isosurface itself. For instance, one

[†] We believe techniques such as [12] have active cell search times at least comparable to the ones we have, but in general, these other techniques need the whole dataset to be loaded into main memory, and the preprocessing has to be done each time the dataset is loaded. Also, the indexing data structures increase the amount of main memory needed (if only by a small amount), thus making these methods less likely to be used for very large datasets.

# of meta-cells	146	361	1100	2364	3600	8400	30628
Total Time	618s	427.4s	346s	331s	331s	347s	376s
Meta-Cell Size	20.8M	21.5M	22.6M	23.7M	24.6M	26.7M	31.7M
Avg Vertex	2032.8	940.1	370.4	202.8	148.2	79.3	31.4
Avg Cell	6888.1	2785.8	914.2	425.4	279.35	119.7	32.8
Increase	7%	10%	16%	22%	26.9%	37.9%	63%
BBIO_Tree (size)	4K	16K	48K	112K	168K	640K	1.7M
BBIO_Tree (time)	0.42s	0.61s	1.51s	1.94s	3.78s	13.1s	31.9s
Query (act)	49.3K	49.3K	49.3K	49.3K	49.3K	49.3K	49.3K
Query (fetch)	704K	560K	418K	345K	320K	247K	167K
Query (mc)	87	189	414	754	1094	1996	4923
Perc. (mc)	59%	52%	37%	31%	30%	23%	16%
Query Time	3.34s	2.76s	2.09s	1.82s	1.73s	1.5s	1.18s

Table 3: Statistics for preprocessing and querying isosurfaces on the Delta dataset (original binary file size 19.4M). The entries for preprocessing are as defined in Table 2. We also show the performance of a representative isosurface query with 64M of RAM: number of active cells ("act"), number of cells fetched ("fetch"), number of fetched (*i.e.*, active) meta-cells ("mc"), the ratio between the numbers of active and overall meta-cells ("Perc. (mc)"), and finally the time for finding the active cells (the time for actual isosurface generation is not included).

is the Vtk "locator" class, which is used to avoid outputting multiple vertices for the same spatial location.

4 Conclusions

In this paper we present a new out-of-core algorithm for output-sensitive isosurface extraction. In our tests, our algorithm has shown to be both robust and effective in optimizing isosurface queries. Regardless of the size of the dataset, our techniques provide a cost-effective method to speed up isosurface extraction from volume data. The actual code can be made much faster by fine tuning the disk I/O. This is an interesting but hard and time-consuming task, and might often be non-portable across platforms, since the interplay among the operating system, the algorithms, and the disk is non-trivial to optimize. We believe that a substantial speed-up can be achieved by optimizing the external sorting and the file copying primitives.

In the process, we developed two new techniques of independent interest. First, our binary-blocked I/O interval is easier to implement, and uses less disk space than the existing external-memory stabbing-query data structures. Secondly, the technique we use to compute the meta-cells has a wider applicability in the preprocessing of general cell structures larger than main memory. For example, one could use our technique to break polyhedral surfaces larger than main memory into spatially coherent sections for simplification, or to break large volumetric grids into smaller ones for rendering purposes.

We believe this work brings efficient out-of-core isosurface techniques closer to practicality. One remaining challenge is to improve the preprocessing times for large datasets, which, even though is much lower than the ones presented in [10, 11], is still fairly costly.

Acknowledgments

Yi-Jen Chiang was supported in part by NSF Grant DMS-9312098. The work of Cláudio T. Silva was partially supported by Sandia National Labs and the Dept. of Energy Mathematics, Information, and Computer Science Office, and by NSF Grant CDA-9626370.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 1998.

[3] L. Arge and J. S. Vitter. Optimal interval management in external memory. In *Proc. IEEE Foundations of Comp. Sci.*, pages 560–569, 1996.

[4] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for improved interactivity. In *1996 Volume Visualization Symposium*, pages 39–46, October 1996.

[5] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for structured and unstructured meshes in any dimension. In *Proc. Late Breaking Hop Topics*, pages 25–28, 1997.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[7] Y.-J. Chiang. Dynamic and I/O-efficient algorithms for computational geometry and graph problems: theoretical and experimental results. Ph.D. Thesis, Technical Report CS-95-27, Dept. Computer Science, Brown University, 1995.

[8] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Computational Geometry: Theory and Applications*, 9(4):211–236, 1998.

[9] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[10] Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, pages 293–300, 1997.

[11] Y.-J. Chiang and C. T. Silva. Isosurface extraction in large scientific visualization applications using the I/O-filter technique. Technical Report, University at Stony Brook, 1997.

[12] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *1996 Volume Visualization Symposium*, pages 31–38, October 1996.

[13] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proc. IEEE Visualization*, pages 235–244, 1997.

[14] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.

[15] T. A. Funkhouser, S. Teller, C. H. Séquin, and D. Khorramabadi. Database management for models larger than main memory. *Presence: Teleoperators and Virtual Environments*, Vol.5, No.1, 1996.

	Blunt	Chamber	Post	Delta	Cyl3
# of meta-cells	737	1009	1870	30628	27896
mc disk overhead	20%	21%	22%	63%	78%
Iso value	0.67	0.30	0.10	0.21	0.062
Act. cells	20K	37K	16.4K	49K	100K
% fetched	50%	74%	38%	17%	19%
Fetched cells	93K	160K	193K	167K	1.1M
`mcQuery - 64MB`	**4.8s**	**6.97s**	**6.7s**	**16.4s**	**60s**
BBIO	0.1s	0.01s	0.1s	0.1s	0.3s
Disk I/O	2.1s	0.74s	0.9s	1.6s	11.4s
Iso comp.	2.68s	6.2s	5.8s	14.7s	48s
`mcQuery - 128MB`	**3.29s**	**6.8s**	**6.6s**	**15.5s**	**38s**
BBIO	0.1s	0.1s	0.05s	0.1s	0.1s
Disk I/O	0.53s	0.92s	0.82s	1s	11.3s
Iso comp.	2.75s	5.86s	5.78s	14.4s	26.8s
`vtkIso - 64MB`	**5.7s**	**8.1s**	**124s**	**432s**	**2032s**
Disk I/O	4.02s	4.39s	10.8s	23s	428s
Iso comp.	1.69s	3.69s	113s	409s	1604s
`vtkIso - 128MB`	**5.7s**	**8.13s**	**123s**	**425s**	**1337s**
Disk I/O	4.04s	4.41s	11.3s	21.6s	255s
Iso comp.	1.73s	3.72s	112s	403s	1082s
`ioQuery - 64MB`	**1.7s**	**2s**	**1.7s**	**11.7s**	**NA**
Disk I/O	0.5s	0.7s	0.5s	1.7s	NA
Iso comp.	1.2s	1.3s	1.2s	10s	NA
`ioQuery - 128MB`	**1.5s**	**1.5s**	**1.4s**	**11.8s**	**NA**
Disk I/O	0.3s	0.2s	0.2s	1.7s	NA
Iso comp.	1.2s	1.3s	1.2s	10s	NA

Table 4: Statistics for querying isosurfaces on different datasets using 3 different codes: `mcQuery`, `vtkIso`, and `ioQuery`, under two different main memory configurations (64M and 128M). On the top, we specify the datasets, the total number of meta-cells, the meta-cell disk space overhead, the isosurface value being queried, the number of active cells for the particular isovalue, the percentage of the dataset that was fetched during querying, and the actual number of fetched cells. We highlight in bold the overall isosurface generation time for each run. Below we break the times up into its major components. "Iso comp." is always the time to actually compute the isosurface (depending on the method, the number of cells being used for the computation varies). For `mcQuery`, BBIO is the time it takes to query the BBIO tree; Disk I/O is the time to bring the active meta-cells into main memory. For `vtkIso`, Disk I/O is the time to read the dataset from disk. For `ioQuery`, Disk I/O is the time to search (and fetch at the same time) the active cells from disk. Note that `mcQuery` reduces the disk space overhead of `ioQuery` by more than one order of magnitude.

[16] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Comp. Sci.*, pages 714–723, 1993.

[17] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, December 1995.

[18] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM Symp. on Principles of Database Sys.*, pages 233–243, 1993.

[19] Y. Livnat, H.-W. Shen, and C.R. Johnson. A near optimal isosurface extraction algorithm using span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, March 1996.

[20] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Proceedings of SIGGRAPH '87*, pages 163–169, July 1987.

[21] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Proceedings of SIGGRAPH '97*, pages 101–108, August 1997.

[22] W. Schroeder, K. Martin, and W. Lorensen. *The Visualization Toolkit*. Prentice-Hall, 1996.

[23] H.-W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *Proc. IEEE Visualization*, 1996.

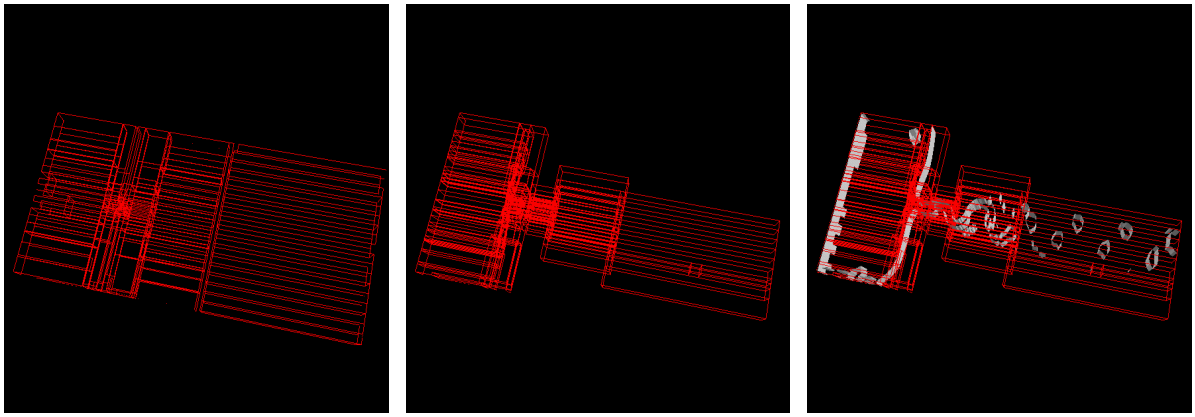
[24] S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan. Partitioning and ordering large radiosity computations. *Proceedings of SIGGRAPH '94*, pages 443–450. July 1994.

[25] S. K. Ueng, K. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, 1997.

[26] M. van Kreveld, R. van Oostrum, C. L. Bajaj, V. Pascucci, and D. R. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. ACM Symp. on Comput. Geom.*, pages 212–220, 1997.

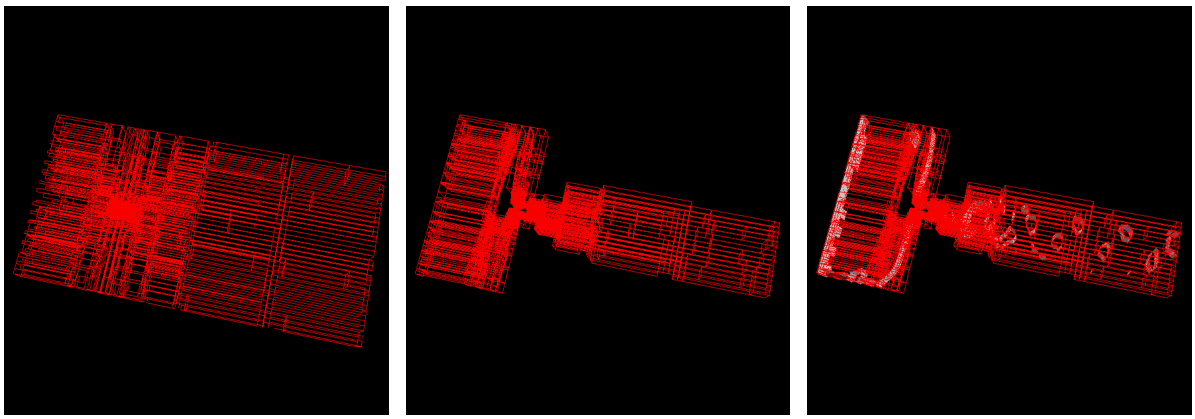
[27] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995.

[28] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 57–62, November 1990.



(a) (b) (c)

Figure 4: Illustration for the distribution of 6^3 meta-cells: (a) the bounding boxes of the meta-cells; (b) the bounding boxes of the fetched meta-cells during a query; (c) the fetched meta-cells superimposed with the isosurface.



(a) (b) (c)

Figure 5: Illustration for the distribution of 10^3 meta-cells in the same dataset as in Fig. 4.

Out-Of-Core Rendering of Large, Unstructured Grids

Ricardo Farias
State University of New York at Stony Brook

Cláudio T. Silva
AT&T

We address the problem of rendering large, unstructured volumetric grids and present a set of techniques that render arbitrarily large data sets on machines with limited memory.

The need to visualize unstructured volumetric data arises in a broad spectrum of applications including structural dynamics, structural mechanics, thermodynamics, fluid mechanics, and shock physics. One of the most powerful visualization techniques is direct volume rendering, a set of rendering techniques that avoids generating intermediary surface representations of the volume data. Direct volume rendering techniques are based on creating optical models that determine how the volume data interacts with light. By changing the modeling, it's possible to render different features of the volume.[1]

Here we address the problem of direct volume rendering of large, unstructured volumetric grids on machines with limited memory. This problem is interesting because such data sets are likely to come from computations generated on supercomputers, while visualization often happens on smaller, desktop machines. Our work also complements the recent trend of developing efficient out-of-core scientific visualization techniques. Given large, unstructured grids, currently several external memory visualization tools exist (such as isosurface computation,[2] streamline computation,[3] and surface simplification[4]) that help scientists visualize their large data sets on machines with limited memory. For instance, by coupling the techniques of Lindstrom[4] and Chiang, Silva, and Schroeder,[2] researchers can compute and simplify isosurfaces of arbitrarily large data sets, effectively visualizing such large data sets on any machine with enough disk space. Our work adds direct volume-rendering algorithms to this already powerful toolbox. (See the "Related Work" sidebar for more background information.)

We present two techniques that vary in rendering speed, disk and memory usage, ease of implementation, and preprocessing costs. The first is a memory-insensitive rendering (MIR) technique that is completely disk-based and requires a small amount of constant main memory. The second technique is based on our ZSweep algorithm. It's more involved in its preprocessing, implementation, and main-memory requirements but can be substantially faster.

Memory-insensitive rendering

In developing efficient external memory algorithms, users must know some characteristics of computer disks and their differences from the in-core main-memory system we're all accustomed to. The basic difference is that disks aren't efficient for random access to locations because "seeks" require a large amount of mechanical movement (of the heads). For sequential access, disks are fast, with a raw bandwidth within a factor of 20 of the main-memory system. Also, we can increase disk bandwidth inexpensively by using several disks in parallel. The appeal of hard drives is that the cost is much lower—on the order of 100 times cheaper than main memory. The need for sequential access when using disks has profound implications for external memory algorithms.

First, the file formats used for out-of-core algorithms must be different and generally more redundant. Indexed mesh formats are common for main-memory techniques. For instance, it's common to save a list of the vertices represented with four floats: the position (x, y, z); scalar field value; and a list of tetrahedra, referenced by four integers that refer to the vertices defining the given tetrahedron. Before we can use such data sets in our algorithm, they must be normalized—a process that dereferences the pointers to vertices. (The Chiang, Silva, and Schroeder paper thoroughly explains this process.[2])

For completeness, we'll briefly explain how to nor-

0272-1716/01/$10.00 © 2001 IEEE

Related Work

The work we describe in this article is mainly related to techniques for rendering unstructured grids and out-of-core visualization techniques. Both are active research areas in scientific visualization. In this sidebar, we briefly review each of these areas.

Unstructured-grid volume rendering

Here we consider existing unstructured-grid volume-rendering techniques from a memory-usage point of view, their applicability to render large grids, and potential extensions for out-of-core rendering. The memory usage of current techniques vary widely, and a straightforward classification of the different techniques isn't possible. Here are some of the various characteristics that generally affect the memory usage of existing techniques:

- the data set's size, in terms of its number and type of cells and vertices. (Given a mesh with t tetrahedra and n vertices, the minimum memory necessary to hold it—assuming uncompressed data and 32 bits for integers and floating-point numbers—is $16(t + n)$ bytes.)
- screen resolution and the data set's image-space depth. (In image space, the memory costs depend on the screen resolution and the data set's thickness along the z direction. Some techniques compute slices along z by intersecting discrete buffers of the same resolution as the screen with the unstructured grid. Assuming 1 byte per color channel, for computing an image of size N-by-N with s slices, we need $4sN^2$ bytes. We note that s should vary with the resolution of the data set in z. That is, if a ray that intersects the data set in s_{max} cells exists, then the closer s gets to s_{max} the more accurate the image we can obtain.)
- the use of mesh connectivity information. Some techniques explicitly use connectivity information, while others use different means of inferring it (such as discrete buffers used for determining depth information) or completely avoid using any kind of connectivity.
- the underlying data structures used for efficiency or accuracy. For instance, some techniques cache extra information per cell or per face of the data set for efficiency.

Researchers have developed several efficient algorithms for rendering irregular grids. One class of algorithms is based on adapting ray-tracing techniques for rendering unstructured grids, such as in the works of Garrity,[1] Uselton,[2] and Bunyk, Kauman, and Silva.[3] In general, these techniques require random access to the cells, connectivity information, and in some cases, extra memory to optimize the computation of intersections of rays with faces of the cell complex. Yang, Mitra, and Chiueh's paper[4] proposes an optimization for the technique in the Bunyk paper[3] that attempts to reduce the memory requirements by compositing samples as early as possible, but the proposed view-independent traversal doesn't limit the overall memory use. (The work of Hong and Kaufman,[5] although similar to that in the Bunyk paper,[3] is optimized for curvilinear grids. They used considerably less memory because their system uses the grid structure and doesn't explicitly store cell or connectivity information.)

Researchers have developed other techniques that use scan-line algorithms, which sweep the data with a plane perpendicular to the image plane.[6] Some of these techniques[7] are designed to be memory efficient but still use the mesh's connectivity. Others, such as those proposed by Giertsen[8] and Westermann and Ertl,[9] use discrete buffers to determine the compositing order and completely avoid the need for connectivity information. Using discrete buffers in z potentially lowers the accuracy of these techniques, and the buffers themselves can require a substantial amount of memory.

Some methods[6,10] employ a different kind of sweep algorithm and sweep planes in z. Yagel et al.[11] sample the irregular grid with a fixed number of planes that are later composited together. Their technique doesn't use connectivity, but the space to keep the planes can be substantial because it amounts to computing and caching many images. Farias, Mitchell, and Silva[10] developed ZSweep, which is also based on sweeping a plane in the z direction.

Another approach for rendering irregular grids is using face projection, or feed-forward, methods[12-14] in which the cells are projected onto the screen one by one. Most of these techniques exploit the graphics hardware to compute the volumetric lighting models[13] by first computing a visibility ordering[12,15,16] and incrementally accumulating their contributions to the final image. With respect to memory usage, we can separate the visibility ordering algorithms into two classes: those that use connectivity to compute the ordering[12,16] and those that use some form of power-sorting.[14] The power sorting techniques only require an extra floating-point number per cell, and they don't use connectivity information. In general, those techniques aren't guaranteed to generate correct sorting results for a wide class of grids.

One simple approach[17] is to naively compute all intersections between each ray cast with all the cells and perform a postsorting to compute the image. That is, given an N-by-N image and n cells, for each of the N^2 rays, compute the $O(n)$ intersections with cell facets in time $O(n)$ and then sort these crossing points in $O(n \log n)$ time.

continued on p. 4

continued from p. 3
However, this results in overall time $O(N^2 n \log n)$ and doesn't take advantage of coherence in the data—the sorted order of cells crossed by one ray isn't used in any way to assist in the processing of nearby rays.

Ma and Crockett[18] used this approach in the context of parallel architectures. Their technique distributes the cells among processors in a round-robin fashion. For each viewpoint, each processor independently computes the ray intersections, which are later composited in the algorithm's second phase. To avoid storing many ray intersections, Ma and Crockett cleverly schedule the computation using a *k*-d tree.

Out-of-core scientific visualization

For a general introduction to out-of-core scientific visualization theory and practice of external memory algorithms, readers should see Abello and Vitter.[19]

Cox and Ellsworth[20] propose a general framework for the systems based on application-controlled demand paging. Leutenegger and Ma[21] propose using R-trees[22] to optimize searching operations on large unstructured data sets. Ueng, Sikorski, and Ma[23] use an octree partition to restructure unstructured grids, optimizing the computation of streamlines. Shen, Chiang, and Ma[24] and Sutton and Hansen[25] have developed techniques for indexing time-varying data sets. Shen, Chiang, and Ma[24] apply their technique for volume rendering, while Sutton and Hansen[25] focus on isosurface computations.

Chiang and Silva[26] worked on I/O-optimal algorithms for isosurface generation. Their work assumes that even the preprocessing is performed completely on a machine with limited memory. Although their technique is fast in terms of actually computing the isosurfaces, the disk and preprocessing overhead of their technique is substantial. This led to further research[27] on techniques that can trade disk overhead for time in the querying for the active cells. They developed a set of useful metacell preprocessing techniques. Recently, Lindstrom[28] and El-Sana and Chiang[29] developed external memory algorithms for surface simplification. The technique in Lindstrom[30] simplifies arbitrarily large data sets on machines with just enough memory to hold the output triangle mesh.

References

1. M. Garrity, "Raytracing Irregular Volume Data," *Computer Graphics* (San Diego Workshop Volume Visualization), vol. 24, no. 5, Nov. 1990, pp. 35-40.
2. S. Uselton, *Volume Rendering for Computational Fluid Dynamics: Initial Results*, tech. report RNR-91-026, NASA Ames Research Center, Moffett Field, Calif., 1991.
3. P. Bunyk, A. Kaufman, and C. Silva, "Simple, Fast, and Robust Ray Casting of Irregular Grids," *Scientific Visualization* (Proc. Dagstuhl 97), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 30-36.
4. C.-K. Yang, T. Mitra, and T. Chiueh, "On-the-Fly Rendering of Losslessly Compressed Irregular Volume Data," *Proc. IEEE Visualization 2000*, ACM Press, New York, 2000.
5. L. Hong and A. Kaufman, "Accelerated Ray-Casting for Curvilinear Volumes," *Proc. IEEE Visualization 98*, ACM Press, New York, 1998, pp. 247-254.
6. J. Wilhelms et al., "Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids," *Proc. IEEE Visualization 96*, ACM Press, New York, 1996, pp. 57-64.
7. C. Silva and J. Mitchell, "The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 2, Apr.–Jun. 1997, pp. 104-157.
8. C. Giertsen, "Volume Visualization of Sparse Irregular Meshes," *IEEE Computer Graphics and Applications*, vol. 12, no. 2, Mar. 1992, pp. 40-48.
9. R. Westermann and T. Ertl, "The VSbuffer: Visibility Ordering of Unstructured Volume Primitives By Polygon Drawing," *Proc. IEEE Visualization 97*, ACM Press, New York, 1997, pp. 35-42.
10. R. Farias, J. Mitchell, and C. Silva, "ZSweep: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering," *Proc. 2000 Volume Visualization Symp.*, ACM Press, New York, 2000, pp. 91-99.
11. R. Yagel et al., "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *Proc. 1996 Volume Visualization Symp.*, ACM Press, New York, 1996, pp. 55-62.
12. P.L. Williams, "Visibility-Ordering Meshed Polyhedra," *ACM Trans. Graphics*, vol. 11, no. 2, Apr. 1992, pp. 103-126.
13. P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics* (San Diego Workshop Volume Visualization), vol. 24, no. 5, Nov. 1990, pp. 63-70.
14. N. Max, P. Hanrahan, and R. Crawfis, "Area and Volume for Efficient Visualization of 3D Scalar Functions," *Computer Graphics* (San Diego Workshop Volume Visualization), vol. 24, no. 5, Nov. 1990, pp. 27-33.
15. C. Stein, B. Becker, and N. Max, "Sorting and Hardware Assisted Rendering for Volume Visualization," *Proc. 1994 Symp. Volume Visualization*, ACM Press, New York, 1994, pp. 83-90.
16. J. Comba et al., "Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids," *Computer Graphics Forum*, vol. 18, no. 3, Sept. 1999, pp. 369-376.
17. C. Silva, J. Mitchell, and A. Kaufman, "Fast Rendering of Irregular Grids," *Proc. 1996 Volume Visualization Symp.*, ACM Press, New York, 1996, pp. 15-22.
18. K.-L. Ma and T.W. Crockett, "A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data," *Proc. IEEE Parallel Rendering Symp.*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 95-104.

19. J. Abello and J. Vitter, *External Memory Algorithms*, American Mathematical Soc., Providence, R.I., 1999.
20. M. Cox and D. Ellsworth, "Application-Controlled Demand Paging for Out-of-Core Visualization," *Proc. IEEE Visualization 97*, ACM Press, New York, 1997, pp. 235-244.
21. S. Leutenegger and K.-L. Ma, "Fast Retrieval of Disk-Resident Unstructured Volume Data for Visualization," *External Memory Algorithms and Visualization,* Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) Book Series, vol. 50, American Mathematical Soc., Providence, R.I., 1999.
22. A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Conf. Principles Database Systems*, ACM Press, New York, 1984, pp. 47-57.
23. S.-K. Ueng, C. Sikorski, and K.-L. Ma, "Out-of-Core Streamline Visualization on Large Unstructured Meshes," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 4, Oct.–Dec. 1997, pp. 370-380.
24. H.-W. Shen, L.-J. Chiang, and K.-L. Ma, "A Fast Volume Rendering Algorithm for Time-Varying Fields Using A Time-Space Partitioning (TSP) Tree," *Proc. IEEE Visualization 99*, ACM Press, New York, 1999, pp. 371-378.
25. P.M. Sutton and C.D. Hansen, "Accelerated Isosurface Extraction in Time-Varying Fields," *IEEE Trans. Visualization and Computer Graphics*, vol. 6, no. 2, Apr.–Jun. 2000, pp. 98-107.
26. Y.-J. Chiang and C.T. Silva, "I/O Optimal Isosurface Extraction," *IEEE Visualization 97*, ACM Press, New York, 1997, pp. 293-300.
27. Y.-J. Chiang, C.T. Silva, and W.J. Schroeder, "Interactive Out-of-Core Isosurface Extraction," *Proc. IEEE Visualization 98*, ACM Press, New York, 1998, pp. 167-174.
28. P. Lindstrom, "Out-of-Core Simplification of Large Polygonal Models," *Computer Graphics* (Proc. Siggraph 2000), ACM Press, New York, 2000, pp. 259-262.
29. J. El-Sana and Y.-J. Chiang, "External Memory View-Dependent Simplification," *Computer Graphics Forum*, vol. 19, no. 3, Aug. 2000, pp. C-139–C-150.

malize such a file, with v vertices and t tetrahedra. In an initial pass, we create two binary files: one with the list of vertices and another with the list of tetrahedra. Next, in four passes, we dereference each tetrahedral file index and replace it with the actual position and scalar field values for the vertex. To do this efficiently, we first externally sort the current version of the tetrahedra file in the index we intend to dereference. This takes time $O(t \log t)$ using an external memory merge–sort. Then, we perform a synchronous scan of both the vertex and sorted tetrahedra file, reading one record at a time and appropriately outputting the deferenced value for the vertex. Note that we can do this efficiently in time $O(v + t)$ because all the references for vertices are sorted. When we're done with all four passes, the tetrahedra file will contain t records with the value (not reference) of each of its four vertices.

In our first out-of-core rendering technique, MIR, the algorithm receives a transformation matrix, screen resolution, the normalized tetrahedron file, and associated transfer functions for lighting calculations as input.

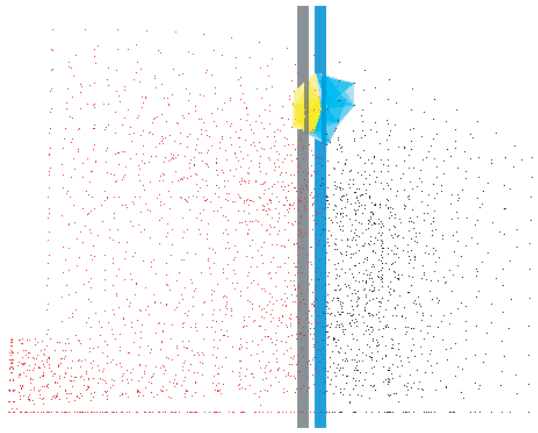
1. The first step in our algorithm is to read each cell (tetrahedron) from the normalized file, transform it with the specified transformation matrix, and compute all its ray intersections. For each pixel ρ_i, which intersects the cell in the interval (z_0, z_1), we output two records (ρ_i, z_0) and (ρ_i, z_1). For color calculations, we also save an interpolated scalar field value. This allows for fast regeneration of images with different transfer functions or (with some changes) the efficient rendering of time-varying data sets. The amount of memory necessary to perform this step is minimal; it's just enough to hold the cell's description and enough temporary storage to compute one intersection, because they're written to disk one by one as they're computed. The amount of disk space required is proportional to the number of actual ray stabbings between rays and cells.
2. The second (and generally, most time consuming) step consists of sorting the file with the ray intersections computed in the previous step, using an appropriate compare function. The compare function we use sorts primarily on the pixel identification ρ_i and secondarily on the depth of intersection z. In other words, after the file is sorted and the records for a particular pixel are together (that is, they appear sequentially in the file), the records are ordered in increasing depth.
3. The third and final step in our scheme is to traverse the ordered file generated in the previous step, use the transfer functions to light, and composite the samples, which are already in the correct order.

Our simple algorithm is essentially an external memory version of a technique previously considered by other researchers.[5,6] One group[5] discarded the technique as too inefficient because it didn't use coherency between rays. Ma and Crockett[6] used this technique for its good load-balancing characteristics. However, to make it practical, they had to optimize it to save space. No space optimizations are necessary for the out-of-core version to be useful. With this scheme, we can render an arbitrarily large image of an arbitrarily large data set if enough disk space exists to save the intersection crossings. It's also simple to implement. It doesn't use any random access to the data set, and its implementation only requires an external sort routine and code to perform ray-cell intersection.

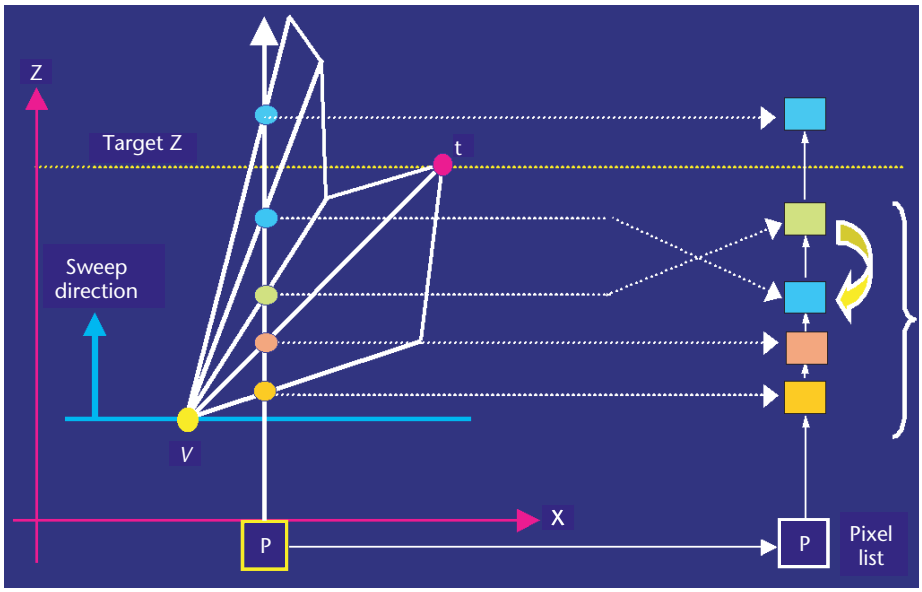
Out-of-core ZSweep

Our second technique is slightly more complex but is often a more efficient out-of-core unstructured grid renderer. It's based on our ZSweep algorithm[7] (see Figure 1, next page, for an overview).

The in-core ZSweep algorithm is based on sweeping the data with a plane parallel to the viewing plane (see the blue plane in Figure 1a) in order of increasing z, pro-



(a)



(b)

1 The in-core ZSweep algorithm. (a) 3D sweep portion of ZSweep. In blue, we show the sweep plane. The swept points are in black, and points that haven't been touched yet are in red. We highlight the tetrahedra incident on the current event point. The newly found faces (which generate new intersections) are in yellow, and the old faces are in cyan. (b) ZSweep compositing in 2D (that is, along a plane perpendicular to the viewing direction) for clarity. The current event point v is in yellow. We also show the newly found faces and the intersections along a general ray. Each intersection contributes a color and has to be composited in the correct order. The ordering computed with an insertion sort is on the right.

jecting the faces of cells that are incident to vertices as they're encountered by the sweep plane. ZSweep's face projection differs from the ones used in other projective methods.[8] During face projection, we compute the intersection of the ray emanating from each pixel and store their z-value and other auxiliary information in a sorted list of intersections for the given pixel. Our data structure for keeping the intersections is similar to an A-buffer.[9] We defer the lighting calculations[1] to a later phase (see Figure 1b). The algorithm performs compositing when it reaches the target Z plane (see the gray plane in Figure 1a). The efficiency arises because the algorithm exploits the implicit (approximate) global ordering that the vertices' z-ordering induces on the cells that are incident on them. This leads to only a few ray intersections that must be processed out of order. The efficiency also arises from using early compositing, which makes the algorithm's memory footprint quite small. The key properties for ZSweep's efficiency is that given a mesh with v vertices and c cells, the amount of sorting ZSweep does is $O(v \log v)$ in practice. Depending on the number of ray intersections, this is substantially lower than the amount necessary to sort all the intersections for each pixel.

ZSweep has two sources of main-memory usage: the pixel intersection lists and the actual data set. The data-set storage requirements represent our largest memory use. Besides the storage for the actual vertices and cells, we must also keep each vertex's use set—that is, the cells incident to each vertex.

The basic idea in our out-of-core technique is to break the data set into chunks of fixed size that we can render independently without using more than a constant amount of memory. To further limit the amount of memory necessary, we subdivide the screen into tiles, and for each tile, we render the chunks that project into it in a front-to-back order. This gives us the same optimizations as the in-core ZSweep algorithm where we've shown that image tiling leads to substantial performance improvement because of better cache coher-

ence.[10] Subdividing the screen into tiles and the data set into chunks that are rendered independently has successfully been applied to a parallelization of ZSweep.

We divided our algorithm into two parts: a view-independent preprocessing phase, which must be performed only once and generates a data file on disk that we can use for all rendering requests, and a view-dependent rendering algorithm.

Preprocessing

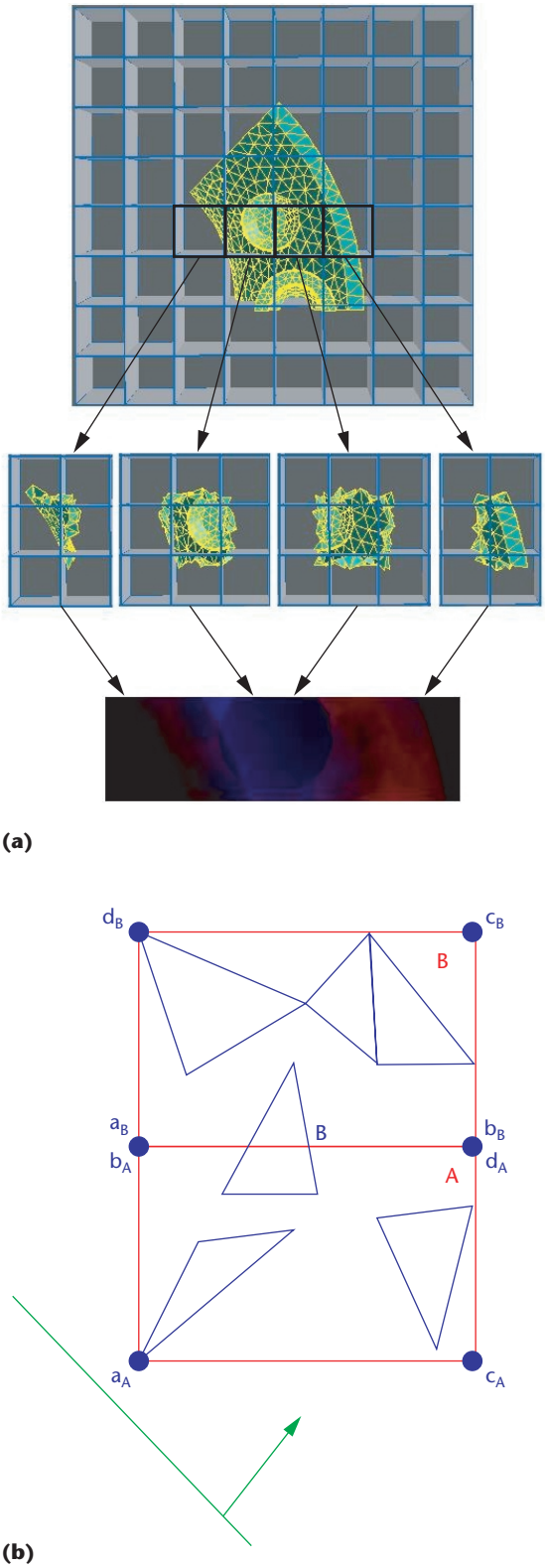
Our preprocessing is simple, and it resembles the metacell creation in the Chiang article.[2] Basically, we break the data-set file into several metacells of small, roughly fixed size. (The metacells and their construction are slightly different in Chiang,[2] because each cell belongs to a single metacell. In our case, a cell belongs to as many metacells as it spatially intersects. This isn't a substantial difference, and the normalization techniques described there still apply.) Given a target number of vertices per metacell m out of v total vertices, we first externally sort all vertices by the x-values and partition them into $\sqrt[3]{v/m}$ consecutive parts. Then, for each such chunk, we externally sort its vertices by the y-values and partition them into $\sqrt[3]{v/m}$ parts. Finally, we repeat the process for each refined part, except that we externally sort the vertices by the z-values. We take the final parts as chunks. This is the main step in constructing the chunks because it determines their shape and location in space. Chunks might differ dramatically in their volumes, but their numbers of vertices are roughly the same.

In general, the number of metacells is relatively small, so we can safely assume they fit in the memory. To render a metacell, ZSweep must have all the cells that spatially intersect that metacell and all the vertices that belong to those cells. These computations can be efficiently computed in external memory. (For full details, see the Chiang article.[2]) Our preprocessing outputs two files. The small one is a high-level description of the metacells, including their bounding box, number of vertices, number of cells, and a pointer to the start of the data for the metacell in the main data file. The larger data file is a list of the vertices and cells for each metacell. Note that several vertices and cells are repeated (possibly multiple times) in this data file, because each metacell is a self-contained unit.

Rendering algorithm

Our rendering algorithm is simple. Basically, we divide the screen into tiles and render the image tile by tile. For each tile, we compute the metacells that intersect that tile, sort the metacells in a front-to-back order, and render it using the ZSweep algorithm.

Figure 2 shows the details. For each tile, we find M, the set of the metacells that project into it. Then, we sort the vertices of the bounding boxes of M in front-to-back order by inserting them on a queue Q. The queue is used for sweeping the vertices, which have several marks. In particular, we tag vertices based on whether they're bounding-box or data-set vertices. When the sweep plane touches the first bounding-box vertex of a metacell m, we retrieve all the vertices and cells of m from disk, transform the vertices, and insert them on Q, tagging them as data-set vertices. Out-of-core ZSweep processing is essentially the same as the in-core algorithm, but it performs reading operations lazily. As it reaches vertices, it projects faces; Figure 1 shows the overall operation. As the algorithm touches bounding-box vertices, we keep track of the number of bounding-box ver-



2 The rendering portion of out-of-core ZSweep, which is performed in (a) tiles. (b) After reaching eight bounding-box vertices of a given metacell, we can safely deallocate the metacell.

Table 1. Main data sets we used for benchmarking.

Data Set	Number of Vertices (1,000)	Number of Cells (1,000)	Metacell File (Kbytes)	Metacell Data (Mbytes)	Normalized File (Mbytes)
Blunt Fin	41	187	40	26	12.7
Combustion Chamber	47	215	40	23	14.6
Oxygen Post	109	513	110	82	34
Delta Wing	212	1,005	254	205	68
SPX	2.9	13	2.6	1.2	0.8
SPX1	20	103	15	12	8
SPX2	150	830	63	110	71
SPX3	1,150	6,620	56	706	641

Table 2. Rendering times (in seconds) for our memory-insensitive irregular grid rendering algorithm.

Screen Resolution 512 x 512				
Data Set	Blunt fin	Combustion chamber	Oxygen post	Delta wing
Projection time	45	10	81	103
Time to order	213	19	386	412
Compositing time	44	6	75	79
Total time	302	35	542	594
Screen Resolution 1024 x 1024				
Data Set	Blunt fin	Combustion chamber	Oxygen post	Delta wing
Projection time	171	24	291	338
Time to order	1,030	82	1,747	1,965
Compositing time	180	26	316	322
Total time	1,381	132	2,354	2,625
Screen Resolution 2048 x 2048†				
Data Set	Blunt fin	Combustion chamber	Oxygen post	Delta wing
Projection time	254	52	435	496
Time to order	589	190	922	1,062
Compositing time	233	55	422	430
Total time	1,076	297	1,779	1,988

† We obtained the times for the 2,048 × 2,048 on a SGI R12K 400-Mhz system, with a fast SCSI disk array. Faster disks on the SGI lead to substantially improved times.

tices of a given metacell that we've seen so far. When this number reaches eight, we can safely deallocate the metacell (see Figure 2b). When we reach vertex d_a, we can free the memory from metacell a.

Experimental results

Here we report results for our two out-of-core rendering techniques and the in-core ZSweep algorithm. When not indicated, we obtained our results on a PC-class machine equipped with an AMD K7 Thunderbird 1-GHz processor, one IDE disk, and 1 Gbyte of main memory running Linux. To limit the amount of main memory available for testing purposes, we used the Linux kernel to indicate the amount of main memory to use by specifying the boot parameters directly into Linux Loader (lilo)—for example, specifying linux mem=32M at the boot prompt. (Chiang, Silva, and Schroeder use a similar methodology.[2] Simply limiting the amount of memory generally isn't enough because the operating system is likely to perform aggressive caching if enough memory is available, thus effectively transferring the data set into memory implicitly.) Table 1 has information about the data sets we used in our tests. The first four are tetrahedralized versions of the well-known NASA data sets. SPX is an unstructured grid (see Figure 1a and 2a) composed of tetrahedra. We subdivided each tetrahedron into eight for each version of the last three—that is, SPX3 is 512 times larger than SPX.

MIR

We've generated several images of the benchmark data sets using our MIR rendering algorithm. Theoretically, MIR shouldn't depend on the amount of main memory available (see Table 2). The four columns in Table 2 for each image dimension show the time it took to project the cells on the screen, the time to order the projection file, the time to compose all intersections, and the total render time.

In all our experiments, our code never used more than 5 Mbytes of main memory. It takes the normalized file as its input. Given a new point of view, it rotates the cells one by one and projects their faces on the screen with a scan conversion that's directly saved in the projection file. The projection file's size depends on the image's dimension and also on the number of segments generated for each pixel. It can get large, but the algorithm works the same. Note that the cost of the algorithm's last step, the compositing, also depends on the average

Table 3. Rendering times (in seconds) for the in-core ZSweep code running with 1 Gbyte of RAM.

Data Set	512^2	1024^2	2048^2
SPX	7	26	118
SPX1	14	46	203
SPX2	29	93	383
SPX3	107	238	834

length of segments. Depending on the data set and image size, MIR can use a lot of disk space. For example, for the Delta, the projection file has 304 Mbytes for a 512×512 image, 1.2 Gbytes for a 1024×1024, and 4.8 Gbytes for a 2048×2048.

Large images

We ran some tests with a large data set (not included in Table 1) containing roughly 1.5 million vertices and 8.5 million cells. Generating a 5000×5000 image (which takes up more than 70 Mbytes of disk) took MIR 224 seconds on a SGI Origin 3000 equipped with R12K 400-Mhz processors and a fast SCSI disk array. This is faster than our other data sets because the number of ray intersections is small. We also generated a $10{,}000 \times 10{,}000$ image from the same data set that took 824 seconds. In this case, the image occupies 300 Mbytes of disk.

Out-of-core ZSweep

Tables 3 and 4 show some results for our out-of-core ZSweep code. Out-of-core ZSweep has constant memory usage per data set, irrespective of the size of the images being generated, and can generate images that the original in-core ZSweep couldn't. For a 2048×2048 image of the Delta, the in-core ZSweep would need more than 380 Mbytes of memory, but the out-of-core ZSweep only needs about 24 Mbytes.

Our experiments show that MIR and out-of-core ZSweep are practical techniques we can use under different conditions. Out-of-core ZSweep is usually more efficient than MIR, sometimes by a factor of 10 or more, but it requires that we preprocess the files with the metacell technique before rendering. However, out-of-core ZSweep uses more memory than MIR. For generating a few high-resolution images of large data sets, MIR might be a good choice.

The MIR code is considerably slower because it performs more sorting and disk I/O. MIR might be particularly useful when trying to render a data set from the same viewpoint with a different transfer function. Because the mapping from scalar values to color (as specified in the transfer function file) is performed during compositing, we can effectively generate images with different classifications efficiently. Also, it would be efficient to render time-varying data sets because the expensive ordering doesn't need to be redone.

Conclusions

We presented two out-of-core volume techniques, which we implemented and tested against one another, and compared their rendering times and memory requirements against the in-core ZSweep algorithm.[7]

Table 4. Rendering times for the out-of-core ZSweep using 128 Mbytes of RAM. We show the time (in seconds) to generate the image and the cost per cell (in μs).

Data Set	512^2		1024^2		2048^2	
SPX	8	615	34	2,615	154	11,846
SPX1	24	233	72	699	305	2,961
SPX2	78	93	160	192	595	716
SPX3	289	43	418	63	1,157	174

The simplest technique, MIR, is useful when the amount of memory available is highly limited or only a few images of a given data set are necessary. We can also use MIR to compute several images of a given data set from the same viewpoint with different classifications (such as transfer functions). For using our out-of-core ZSweep, it would be best if the data's metacell representation is already available. Because such representations are useful for other purposes, such as isosurface generation,[2] we believe this scheme will prove beneficial.

We are currently exploring several extensions of our work. One of the simplest is using prefetching and multithreading to speedup the rendering further in out-of-core ZSweep, especially when multiple processors are available. For real-time rendering, it would be interesting to develop a time-critical version of out-of-core ZSweep,[11] which trades accuracy for speed during rendering. ■

Acknowledgments

We thank Peter Williams and Will Schroeder for interesting data sets and NASA for the Blunt Fin, Liquid Oxygen Post, and Delta Wing data sets. Ricardo Farias acknowledges partial support from CNPq-Brazil under a PhD fellowship. This work was made possible by the generous support of Sandia National Labs and the US Department of Energy Mathematics, Information, and Computer Science Office.

References

1. N. Max, "Optical Models for Direct Volume Rendering," *IEEE Trans. Visualization and Computer Graphics*, vol. 1, no. 2, June 1995, pp. 99-108.
2. Y.-J. Chiang, C.T. Silva, and W.J. Schroeder, "Interactive Out-of-Core Isosurface Extraction," *IEEE Visualization 98*, ACM Press, New York, 1998, pp. 167-174.
3. S.-K. Ueng, C. Sikorski, and K.-L. Ma, "Out-of-Core Streamline Visualization on Large Unstructured Meshes," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 4, Oct.–Dec. 1997, pp. 370-380.
4. P. Lindstrom, "Out-of-Core Simplification of Large Polygonal Models," *Computer Graphics* (Proc. Siggraph 2000), ACM Press, New York, 2000, pp. 259-262.
5. C. Silva, J.S.B. Mitchell, and A.E. Kaufman, "Fast Rendering of Irregular Grids," *1996 Volume Visualization Symp.*, ACM Press, New York, 1996, pp. 15-22.
6. K.-L. Ma and T.W. Crockett, "A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimen-

sional Unstructured Data," *Proc. IEEE Parallel Rendering Symposium*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 95-104.
7. R. Farias, J. Mitchell, and C. Silva, "ZSweep: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering," *Proc. 2000 Volume Visualization Symp.*, ACM Press, New York, 2000, pp. 91-99.
8. P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics*, vol. 24, no. 5, Nov. 1990, pp. 63-70.
9. L. Carpenter, "The A-buffer, An Antialiased Hidden Surface Method," *Computer Graphics* (Proc. Siggraph 1984), ACM Press, New York, 1984, pp. 103-108.
10. R. Farias and C. Silva, "Parallelizing the ZSweep Algorithm for Distributed-Shared Memory Architectures," to be published in *Proc. Int'l Volume Graphics Workshop*, 2001.
11. R. Farias et al., "Time-Critical Rendering of Irregular Grids," *Proc. SIBGRAPI 2000* (Brazilian Computer Graphics Conference), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 243-250.

Ricardo Farias *is a PhD student in operations research in the Applied Math Department at the State University of New York at Stony Brook. His primary research is on visualization of large volumetric data sets and high-performance computing. He has a BS in physics from Fluminense Federal University (Rio de Janeiro, Brazil) and an MS in computer vision from the Graduate School and Research in Engineering Institute (COPPE) of the Federal University of Rio de Janeiro (UFRJ).*

Cláudio Silva *is a senior member of the technical staff in the Information Visualization Research Department at AT&T Labs–Research. His main research interests are in graphics, visualization, applied computational geometry, and high-performance computing. His current research focuses on architectures and algorithms for building scalable displays, rendering techniques for large data sets, 3D scanning, and algorithms for graphics hardware. He has a BS in mathematics from the Federal University of Ceará, Brazil. He has an MS and a PhD in computer science from the State University of New York at Stony Brook. He is an ACM, IEEE, and Eurographics member.*

Readers can contact Silva at AT&T Labs–Research, 180 Park Ave., Room D265, Florham Park, NJ 07932, email csilva@research.att.com.

Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics

Cláudio T. Silva — CSE/OGI/OHSU*
Yi-Jen Chiang — Polytechnic University
Jihad El-Sana — Ben-Gurion University
Peter Lindstrom — LLNL†

Abstract

Recently, several external memory techniques have been developed for a wide variety of graphics and visualization problems, including surface simplification, volume rendering, isosurface generation, ray tracing, surface reconstruction, and so on. This work has had significant impact given that in recent years there has been a rapid increase in the raw size of datasets. Several technological trends are contributing to this, such as the development of high-resolution 3D scanners, and the need to visualize ASCI-size (Accelerated Strategic Computing Initiative) datasets. Another important push for this kind of technology is the growing speed gap between main memory and caches, such a gap penalizes algorithms which do not optimize for coherence of access. Because of these reasons, much research in computer graphics focuses on developing out-of-core (and often cache-friendly) techniques.

This paper surveys fundamental issues, current problems, and unresolved solutions, and aims to provide students and graphics researchers and professionals with an effective knowledge of current techniques, as well as the foundation to develop novel techniques on their own.

Keywords: Out-of-core algorithms, scientific visualization, computer graphics, interactive rendering, volume rendering, surface simplification.

1 INTRODUCTION

Input/Output (I/O) communication between fast internal memory and slower external memory is a major bottleneck in many large-scale applications. Algorithms specifically designed to reduce the I/O bottleneck are called *external-memory* algorithms.

This paper focusses on describing techniques for handling datasets larger than main memory in scientific visualization and computer graphics. Recently, several external memory techniques have been developed for a wide variety of graphics and visualization problems, including surface simplification, volume rendering, isosurface generation, ray tracing, surface reconstruction, and so on. This work has had significant impact given that in recent years there has been a rapid increase in the raw size of datasets. Several technological trends are contributing to this, such as the development of high-resolution 3D scanners, and the need to visualize ASCI-size (Accelerated Strategic Computing Initiative) datasets. Another important push for this kind of technology is the growing speed gap between main memory and caches, such a gap penalizes algorithms which do not optimize for coherence of access. Because of these reasons, much research in computer graphics focuses on developing out-of-core (and often cache-friendly) techniques.

*Oregon Health & Science University
†Lawrence Livermore National Laboratory

The paper reviews fundamental issues, current problems, and unresolved solutions, and presents an in-depth study of external memory algorithms developed in recent years. Its goal is to provide students and graphics professionals with an effective knowledge of current techniques, as well as the foundation to develop novel techniques on their own.

It starts with the basics of external memory algorithms in Section 2. Then in the remaining sections, it reviews the current literature in other areas. Section 4 covers surface simplification algorithms. Section 3 covers work in scientific visualization, including isosurface computation, volume rendering, and streamline computation. Section 5 discusses rendering approaches for large datasets. Finally, Section 6 talks about computing high-quality images by using global illumination techniques.

2 EXTERNAL MEMORY ALGORITHMS

The field of *external-memory algorithms* started quite early in the computer algorithms community, essentially by the paper of Aggarwal and Vitter [3] in 1988, which proposed the external-memory computational model (see below) that has been extensively used today. (External sorting algorithms were developed even earlier—though not explicitly described and analyzed under the model of [3]; see the classic book of Knuth [57] in 1973.) Early work on external-memory algorithms, including Aggarwal and Vitter [3] and other follow-up results, concentrated largely on problems such as sorting, matrix multiplication, and FFT. Later, Goodrich et al. [47] developed I/O-efficient algorithms for a collection of problems in computational geometry, and Chiang et al. [17] gave I/O-efficient techniques for a wide range of computational graph problems. These papers also proposed some fundamental paradigms for external-memory geometric and graph algorithms. Since then, developing external-memory algorithms has been an intensive focus of research, and considerable results have been obtained in computational geometry, graph problems, text string processing, and so on. We refer to Vitter [84] for an extensive and excellent survey on these results. Also, the volume [1] is entirely devoted to external-memory algorithms and visualization.

Here, we review some fundamental and general external-memory techniques that have been demonstrated to be very useful in scientific visualization and computer graphics. We begin with the computational model of Aggarwal and Vitter [3], followed by two major computational paradigms:

(1) *Batched computations*, in which no preprocessing is done and the entire data items must be processed. A common theme is to stream the data through main memory in one or more passes, while only keeping a relatively small portion of the data related to the current computation in main memory at any time.

(2) *On-line computations*, in which computation is performed for a series of query operations. A common technique is to perform a preprocessing step in advance to organize the data into a data structure *stored in disk* that is indexed to facilitate efficient searches, so that each query can be performed by searching in the data structure that examines only a very small portion of the data. Typically an even smaller portion of the data needs to be kept in main memory at any time during each query. This is in a similar spirit of performing queries in database.

We remark that the preprocessing step mentioned in (2) is actually a batched computation. Other general techniques such as *caching* and *prefetching* may be combined with the above computational paradigms to obtain further speed-ups (e.g., by reducing the necessary I/O's for blocks already in main memory and/or by overlapping I/O operations with main-memory computations), again via exploiting the particular computational properties of each individual problem as part of the algorithm design.

In Sec. 2.1, we present the computational model of [3]. In Sec. 2.2, we review three techniques in batched computations that are fundamental for out-of-core scientific visualization and graphics: external merge sort [3], out-of-core pointer de-referencing [14, 17, 18], and the meta-cell technique [20]. In Sec. 2.3, we review some important data structures for on-line computations, namely the B-tree [9, 26] and B-tree-like data structures, and show a general method of converting a main-memory, binary-tree structure into a B-tree-like data structure. In particular, we review the BBIO tree [19, 20], which is an external-memory version of the main-memory interval tree [33] and is essential for isosurface extraction, as a non-trivial example.

2.1 Computational Model

In contrast to random-access main memory, disks have extremely long access times. In order to amortize this access time over a large amount of data, a typical disk reads or writes a large block of contiguous data at once. To model the behavior of I/O systems, Aggarwal and Vitter [3] proposed the following parameters:

$$\begin{aligned} N &= \text{\# of items in the problem instance} \\ M &= \text{\# of items that can fit into main memory} \\ B &= \text{\# of items per disk block} \end{aligned}$$

where $M < N$ and $1 \ll B \leq M \ 2$[1]. Each I/O operation reads or writes one disk block, i.e., B items of data. Since I/O operations are much slower (typically two to three orders of magnitude) than main-memory accesses or CPU computations, the measure of performance for external-memory algorithms is the number of I/O operations performed; this is the standard notion of *I/O complexity* [3]. For example, reading all of the input data requires $N \ B$ I/O's. Depending on the size of the data items, typical values for workstations and file servers in production today are on the order of $M = 10^6$ to $M = 10^8$ and $B = 10^2$ to $B = 10^3$. Large-scale problem instances can be in the range $N = 10^{10}$ to $N = 10^{12}$.

We remark that sequentially scanning through the entire data takes $\Theta(\frac{N}{B})$ I/O's, which is considered as the *linear* bound, and external sorting takes $\Theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/O's [3] (see also Sec. 2.2.1), which is considered as the *sorting* bound. It is very important to observe that randomly accessing the entire data, one item at a time, takes $\Theta(N)$ I/O's in the worst case and is much more inefficient than an external sorting in practice. To see this, consider the sorting bound: since $M \ B$ is large, the term $\log_{\frac{M}{B}}\frac{N}{B}$ is much smaller than the term B, and hence the sorting bound is much smaller than $\Theta(N)$ in practice. In Sec. 2.2.2, we review a technique for a problem that greatly improves the I/O bound from $\Omega(N)$ to the sorting bound.

2.2 Batched Computations

2.2.1 External Merge Sort

Sorting is a fundamental procedure that is necessary for a wide range of computational tasks. Here we review the external merge sort [3] under the computational model [3] presented in Sec. 2.1.

The external merge sort is a k-way merge sort, where k is chosen to be $M \ B$, the maximum number of disk blocks that can fit in main memory. It will be clear later for this choice. The input is a list of N items stored in contiguous places in disk, and the output will be a sorted list of N items, again in contiguous places in disk.

[1] An additional parameter, D, denoting the number of disks, was also introduced in [3] to model parallel disks. Here we consider the standard single disk model, i.e., $D = 1$, and ignore the parameter D. It is common to do so in the literature of external-memory algorithms.

The algorithm is a recursive procedure as follows. In each recursion, if the current list L of items is small enough to fit in main memory, then we read this entire list into main memory, sort it, and write it back to disk in contiguous places. If the list L is too large to fit in main memory, then we split L into k sub-lists of equal size, sort each sub-list recursively, and then merge all sorted sub-lists into a single sorted list. The major portion of the algorithm is how to merge the k sorted sub-lists in an I/O-optimal way. Notice that each sub-list may also be too large to fit in main memory. Rather than reading one item from each sub-list for merging, we read *one block* of items from each sub-list into main memory each time. We use k blocks of main memory, each as a *1-block buffer* for a sub-list, to hold each block read from the sub-lists. Initially the first block of each sub-list is read into its buffer. We then perform merging on items in the k buffers, where each buffer is already sorted, and output sorted items, as results of merging, to disk, written in units of blocks. When some buffer is exhausted, the next block of the corresponding sub-list is read into main memory to fill up that buffer. This process continues until all k sub-lists are completely merged. It is easy to see that merging k sub-lists of total size $|L|$ takes $O(|L|\ \ B)$ I/O's, which is optimal—the same I/O bound as reading and writing all sub-lists once.

To analyze the overall I/O complexity, we note that the recursive procedure corresponds to a k-ary tree (rather than a binary tree as in the two-way merge sort). In each level of recursion, the total size of list(s) involved is N items, and hence the total number of I/O's used per level is $O(N\ \ B)$. Moreover, there are $O(\log_k \frac{N}{B})$ levels, since the initial list has $N\ \ B$ blocks and going down each level reduces the (sub-)list size by a factor of $1\ \ k$. Therefore, the overall complexity is $O(\frac{N}{B}\log_k \frac{N}{B})$ I/O's. We want to maximize k to optimize the I/O bound, and the maximum number of 1-block buffers in main memory is $M\ \ B$. By taking $k = M\ \ B$, we get the bound of $O(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B})$ I/O's, which is optimal[2] [3].

Note the technique of using a 1-block buffer in main memory for each sub-list that is larger than main memory in the above merging step. This has lead to the *distribution sweep* algorithm developed in Goodrich et al. [47] and implemented and experimented in Chiang [15] for the 2D orthogonal segment intersection problem, as well as the general *scan and distribute* paradigm developed by Chiang and Silva [18] and Chiang et al. [20] to build the *I/O interval tree* [6] used in [18] and the *binary-blocked I/O interval tree* (the *BBIO tree* for short) developed and used in [20], for out-of-core isosurface extraction. This scan and distribute paradigm enables them to perform preprocessing to build these trees (as well as the *metablock tree* [54]) in an I/O-optimal way; see Chiang and Silva [19] for a complete review of these data structures and techniques.

2.2.2 Out-of-Core Pointer De-Referencing

Typical input datasets in scientific visualization and computer graphics are given in compact *indexed* forms. For example, scalar-field irregular-grid volume datasets are usually represented as tetrahedra meshes. The input has a list of vertices, where each vertex appears exactly once and each vertex entry contains its x-, y-, z- and scalar values, and a list of tetrahedral cells, where each cell entry contains pointers/indices to its vertices in the vertex list. We refer to this as the *index cell set (ICS)* format. Similarly, in an *indexed triangle mesh*, the input has a list of vertices containing the vertex coordinates and a list of triangles containing pointers/indices to the corresponding vertex entries in the vertex list.

The very basic operation in many tasks of processing the datasets is to be able to traverse all the tetrahedral or triangular cells and obtain the vertex information of each cell. While this is trivial if the entire vertex list fits in main memory—we can just follow the vertex pointers and perform pointer de-referencing, it is far from straightforward to carry out the task efficiently in the out-of-core setting where the vertex list

[2] A matching lower bound is shown in [3].

or both lists do not fit. Observe that following the pointers results in random accesses in disk, which is very inefficient: since each I/O operation reads/writes an entire disk block, we have to read an entire disk block of B items into main memory in order to just access a single item in that block, where B is usually in the order of hundreds. Suppose the vertex and cell lists have N items in total, then this would require $\Omega(N)$ I/O's in the worst case, which is highly inefficient.

An I/O-efficient technique to perform pointer de-referencing is to replace (or augment) each vertex pointer/index of each cell with the corresponding direct vertex information (coordinates, plus the scalar value in case of volumetric data); this is the *normalization* process developed in Chiang and Silva [18], carried out I/O-efficiently in [18] by applying the technique of Chiang [14, Chapter 4] and Chiang et al. [17] as follows. In the first pass, we externally sort the cells in the cell list, using as the key for each cell the index (pointer) to the first vertex of the cell, so that the cells whose first vertices are the same are grouped together, with the first group having vertex 1 as the first vertex, the second group having vertex 2 as the first vertex, and so on. Then by scanning through the vertex list (already in the order of vertex 1, vertex 2, etc. from input) and the cell list simultaneously, we can easily fill in the direct information of the first vertex of each cell in the cell list in a sequential manner. In the second pass, we sort the cell list by the indices to the second vertices, and fill in the direct information of the second vertex of each cell in the same way. By repeating the process for each vertex of the cells, we obtain the direct vertex information for all vertices of each cell. Actually, each pass is a *joint* operation (commonly used in database), using the vertex ID's (the vertex indices) as the key on both the cell list and the vertex list. In each pass, we use $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/O's for sorting plus $O(N\ \ B)$ I/O's for scanning and filling in the information, and we perform three or four passes depending on the number of vertices per cell (a triangle or tetrahedron). The overall I/O complexity is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$, which is far more efficient than $\Omega(N)$ I/O's.

The above out-of-core pointer de-referencing has been used in [18, 20] in the context of out-of-core isosurface extraction, as well as in [34, 62] in the context of out-of-core simplification of polygonal models. We believe that this is a very fundamental and powerful technique that will be essential for many other problems in out-of-core scientific visualization and computer graphics.

2.2.3 The Meta-Cell Technique

While the above *normalization* process (replacing vertex indices with direct vertex information) enables us to process indexed input format I/O-efficiently, it is most suitable for *intermediate computations*, and not for a *final database* or *final data representation* stored in disk for on-line computations, since the disk space overhead is large—the direct vertex information is duplicated many times, once per cell sharing the vertex.

Aiming at optimizing both disk-access cost and disk-space requirement, Chiang et al. [20] developed the *meta-cell* technique, which is essentially an I/O-efficient partition scheme for irregular-grid volume datasets (partitioning regular grids is a much simpler task, and can be easily carried out by a greatly simplified version of the meta-cell technique). The resulting partition is similar to the one induced by a *k-d*-tree [10], but there is no need to compute the multiple levels. The meta-cell technique has been used in Chiang et al. [20] for out-of-core isosurface extraction, in Farias and Silva [40] for out-of-core volume rendering, and in Chiang et al. [16] for a unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids.

Now we review the meta-cell technique. Assume the input dataset is a tetrahedral mesh given in the *index cell set (ICS)* format consisting of a vertex list and a cell list as described in Sec. 2.2.2. We cluster spatially neighboring cells together to form a meta-cell. Each meta-cell is roughly of the same storage size, usually in a multiple of disk blocks and always able to fit in main memory. Each meta-cell has *self-contained* information and is always read as a whole from disk to main memory. Therefore, we can use the compact

ICS representation for each meta-cell, namely a local vertex list and a local cell list which contains pointers to the local vertex list. In this way, a vertex shared by many cells in the same meta-cell is stored just *once* in that meta-cell. The only duplications of vertex information occur when a vertex belongs to two cells in different meta-cells; in this case we let both meta-cells include that vertex in their vertex lists to make each meta-cell self-contained.

The meta-cells are constructed as follows. First, we use an external sorting to sort all vertices by their x-values, and partition them evenly into k chunks, where k is a parameter that can be adjusted. Then, for each of the k chunks, we externally sort the vertices by the y-values and again partition them evenly into k chunks. Finally, we repeat for the z-values. We now have k^3 chunks, each having about the same number of vertices. Each final chunk corresponds to a meta-cell, whose vertices are the vertices of the chunk (plus some additional vertices duplicated from other chunks; see below). A cell with all vertices in the same meta-cell is assigned to that meta-cell; if the vertices of a cell belong to different meta-cells, then a voting scheme is used to assign the cell to a single meta-cell, and the missing vertices are duplicated into the meta-cell that owns this cell. We then construct the local vertex list and the local cell list for each meta-cell. Recall that k is a parameter and we have k^3 meta-cells in the end. When k is larger, we have more meta-cell boundaries and the number of duplicated vertices is larger (due to more cells crossing the meta-cell boundaries). On the other hand, having a larger k means each meta-cell is more refined and contains less information, and thus disk read of a meta-cell is faster (fewer number of disk blocks to read). Therefore, the meta-cell technique usually leads to a trade-off between query time and disk space.

The out-of-core pointer de-referencing technique (or the *joint* operation) described in Sec. 2.2.2 is essential in various steps of the meta-cell technique. For example, to perform the voting scheme to assign cells to meta-cells, we need to know, for each cell, the destination meta-cells of its vertices. Recall that in the input cell list, each cell only has the indices (vertex ID's) to the vertex list. When we obtain k^3 chunks of vertices, we assign the vertices to these k^3 meta-cells by generating a list of tuples (v_{id}, m_{id}), meaning that vertex v_{id} is assigned to meta-cell m_{id}. Then a *joint* operation using vertex ID's as the key on this list and the cell list completes the task by replacing each vertex ID in each cell with the destination meta-cell ID of the vertex. There are other steps involving the *joint* operation; we refer to [20] for a complete description of the meta-cell technique. Overall, meta-cells can be constructed by performing a few external sortings and a few *joint* operations, and hence the total I/O complexity is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/O's.

2.3 On-Line Computations: B-Trees and B-Tree-Like Data Structures

Tree-based data structures arise naturally in the on-line setting, since data items are stored sorted and queries can typically be performed by efficient searches on the trees. The well-known balanced multiway *B-tree* [9, 26] (see also [27, Chapter 18]) is the most widely used data structure in external memory. Each tree node corresponds to one disk block, capable of holding up to B items. The *branching factor*, *Bf*, defined as the number of children of each internal node, is $\Theta(B)$ (except for the root); this guarantees that the height of a B-tree storing N items is $O(\log_B N)$ and hence searching an item takes optimal $O(\log_B N)$ I/O's. Other dynamic dictionary operations, such as insertion and deletion of an item, can be performed in optimal $O(\log_B N)$ I/O's each, and the space requirement is optimal $O(N\ \ B)$ disk blocks.

Typical trees in main memory have branching factor 2 (binary tree) or some small constant (e.g., 8 for an octree), and each node stores a small constant number of data items. If we directly map such a tree to external memory, then we get a *sub-optimal* disk layout for the tree: accessing each tree node takes one I/O, in which we read an entire block of B items just to access a constant number of items of the node in the block. Therefore, it is desirable to *externalize* the data structure, converting the tree into a *B-tree-like* data structure, namely, to increase the branching factor from 2 (or a small constant) to $\Theta(B)$ so that the height of

a balanced tree is reduced from $O(\log N)$ to $O(\log_B N)$, and also to increase the number of items stored in each node from $O(1)$ to $\Theta(B)$.

A simple and general method to externalize a tree of constant branching factor is as follows. We block a subtree of $\Theta(\log B)$ levels of the original tree into *one node* of the new tree (see Fig. 1 on page 8), so that the branching factor is increased to $\Theta(B)$ and each new tree node stores $\Theta(B)$ items, where each new tree node corresponds to one disk block. This is the basic idea of the BBIO tree of Chiang et al. [19, 20] to externalize the interval tree [33] for out-of-core isosurface extraction, and of the *meta-block tree* of El-Sana and Chiang [34] to externalize the view-dependence tree [37] for external memory view-dependent simplification and rendering. We believe that this externalization method is general and powerful enough to be applicable to a wide range of other problems in out-of-core scientific visualization and graphics.

We remark that the interval tree [33] is more difficult to externalize than the view-dependence tree [37]. When we visit a node of the view-dependence tree, we access all information stored in that node. In contrast, each internal node in the interval tree has *secondary lists* as secondary search structures, and the optimal query performance relies on the fact that searching on the secondary lists can be performed in an *output-sensitive* way—the secondary lists should not be visited entirely if not all items are reported as answers to the query. In the rest of this section, we review the BBIO tree as a non-trivial example of the externalization method.

2.3.1 The Binary-Blocked I/O Interval Tree (BBIO Tree)

The *binary-blocked I/O interval tree* (*BBIO tree*) of Chiang et al. [19,20] is an external-memory extension of the (main-memory) binary interval tree [33]. As will be seen in Sec. 3, the process of *finding active cells* in isosurface extraction can be reduced to the following problem of *interval stabbing queries* [22]: given a set of N intervals in 1D, build a data structure so that for a given query point q we can efficiently report all intervals containing q. Such interval stabbing queries can be optimally solved in main memory using the interval tree [33], with $O(N)$ space, $O(N \log N)$ preprocessing time (the same bound as sorting) and $O(\log N + K)$ query time, where K is the number of intervals reported; all bounds are optimal in terms of main-memory computation. The BBIO tree achieves the optimal performance in external-memory computation: $O(N \ B)$ blocks of disk space, $O(\log_B N + \frac{K}{B})$ I/O's for each query, and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/O's (the same bound as external sorting) for preprocessing. In addition, insertion and deletion of intervals can be supported in $O(\log_B N)$ I/O's each. All these bounds are I/O-optimal.

We remark that the *I/O interval tree* of Arge and Vitter [6] is the first external-memory version of the main-memory interval tree [33] achieving the above optimal I/O-bounds, and is used in Chiang and Silva [18] for the first work on out-of-core isosurface extraction. Comparing the BBIO tree with the I/O interval tree, the BBIO tree has only two kinds of secondary lists (the same as the original interval tree [33]) rather than three kinds, and hence the disk space is reduced by a factor of 2/3 in practice. Also, the branching factor is $\Theta(B)$ rather than $\Theta(\sqrt{B})$ and hence the tree height is halved. The tree structure is simpler; it is easier to implement, also for handling degenerate cases.

Here we only review the data structure and the query algorithm of the BBIO tree; the preprocessing is performed by the *scan and distribute* paradigm mentioned at the end of Sec. 2.2.1 and is described in [19,20]. The algorithms for insertions and deletions of intervals are detailed in [19].

2.3.1.1 Review: the Binary Interval Tree

We first review the main-memory binary interval tree [33]. Given a set of N intervals, such interval tree T is defined recursively as follows. If there is only one interval, then the current node r is a leaf containing

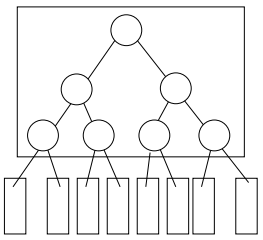


Figure 1: Intuition of a binary-blocked I/O interval tree (BBIO tree) $\mathscr{T}$: each circle is a node in the binary interval tree T, and each rectangle, which blocks a subtree of T, is a node of $\mathscr{T}$.

that interval. Otherwise, r stores as a key the median value m that partitions the interval endpoints into two slabs, each having the same number of endpoints that are smaller (resp. larger) than m. The intervals that contain m are assigned to the node r. The intervals with both endpoints smaller than m are assigned to the left slab; similarly, the intervals with both endpoints larger than m are assigned to the right slab. The left and right subtrees of r are recursively defined as the interval trees on the intervals in the left and right slabs, respectively. In addition, each internal node u of T has two secondary lists: the *left list*, which stores the intervals assigned to u, sorted in *increasing left endpoint values*, and the *right list*, which stores the same set of intervals, sorted in *decreasing right endpoint values*. It is easy to see that the tree height is $O(\log_2 N)$. Also, each interval is assigned to exactly one node, and is stored either twice (when assigned to an internal node) or once (when assigned to a leaf), and thus the overall space is $O(N)$.

To perform a query for a query point q, we apply the following recursive process starting from the root of T. For the current node u, if q lies in the left slab of u, we check the left list of u, reporting the intervals sequentially from the list until the first interval is reached whose left endpoint value is larger than q. At this point we stop checking the left list since the remaining intervals are all to the right of q and cannot contain q. We then visit the left child of u and perform the same process recursively. If q lies in the right slab of u then we check the right list in a similar way and then visit the right child of u recursively. It is easy to see that the query time is optimal $O(\log_2 N + K)$, where K is the number of intervals reported.

2.3.1.2 Data Structure

Now we review the BBIO tree, denoted by $\mathscr{T}$, and recall that the binary interval tree is denoted by T. Each node in $\mathscr{T}$ is one disk block, capable of holding B items. We want to increase the branching factor Bf so that the tree height is $O(\log_B N)$. The intuition is very simple—we *block* a subtree of the binary interval tree T into one node of $\mathscr{T}$ (see Fig. 1), as described in the general externalization method presented in the beginning of Sec. 2.3. In the following, we refer to the nodes of T as *small nodes*. We take the branching factor Bf to be $\Theta(B)$. In an internal node of $\mathscr{T}$, there are $Bf-1$ small nodes, each having a key, a pointer to its left list and a pointer to its right list, where all left and right lists are stored in disk.

Now we give a more formal definition of the tree $\mathscr{T}$. First, we sort all *left* endpoints of the N intervals in increasing order from left to right, into a set E. We use interval ID's to break ties. The set E is used to define the keys in small nodes. The BBIO tree $\mathscr{T}$ is recursively defined as follows. If there are no more than B intervals, then the current node u is a leaf node storing all intervals. Otherwise, u is an internal node. We take $Bf-1$ median values from E, which partition E into Bf slabs, each with the same number of endpoints. We store sorted, in non-decreasing order, these $Bf-1$ median values in the node u, which serve as the keys of the $Bf-1$ small nodes in u. We *implicitly* build a subtree of T on these $Bf-1$ small nodes, by a *binary-search scheme* as follows. The root key is the median of the $Bf-1$ sorted keys, the key of the left child of the

root is the median of the lower half keys, and the right-child key is the median of the upper half keys, and so on. Now consider the intervals. The intervals that contain one or more keys of u are assigned to u. In fact, each such interval I is assigned to the *highest* small node (in the subtree of T in u) whose key is contained in I; we store I in the corresponding left and right lists of that small node in u. For the remaining intervals that are not assigned to u, each has both endpoints in the same slab and is assigned to that slab; recall that there are Bf slabs induced by the $Bf-1$ keys stored in u. We recursively define the Bf subtrees of the node u as the BBIO trees on the intervals in the Bf slabs. Notice that with the above binary-search scheme for implicitly building a (sub)tree of small nodes on the keys stored in an internal node u of $\mathscr{T}$, Bf does not need to be a power of 2—we can make Bf as large as possible, as long as the $Bf-1$ keys, the $2(Bf-1)$ pointers to the left and right lists, and the Bf pointers to the children, etc., can all fit into one disk block.

It is easy to see that $\mathscr{T}$ has height $O(\log_B N)$: $\mathscr{T}$ is defined on the set E with N left endpoints, and is perfectly balanced with $Bf = \Theta(B)$. To analyze the space complexity, observe that there are no more than $N \ B$ leaves and thus $O(N \ B)$ disk blocks for the tree nodes of $\mathscr{T}$. For the secondary lists, as in the binary interval tree T, each interval is stored either once or twice. The only issue is that a left (right) list may have very few ($<< B$) intervals but still needs one disk block for storage. We observe that an internal node u has $2(Bf-1)$ left plus right lists, *i.e.*, at most $O(Bf)$ such *underfull* blocks. But u also has Bf children, and thus the number of underfull blocks is no more than a constant factor of the number of child blocks—counting only the number of tree nodes suffices to take into account also the number of underfull blocks, up to some constant factor. Therefore the overall space complexity is optimal $O(N \ B)$ disk blocks.

As we shall see in Sec. 2.3.1.3, the above data structure supports queries in non-optimal $O(\log_2 \frac{N}{B} + K \ B)$ I/O's (where K is the number of intervals reported), and we can use the *corner structures* [54] to achieve optimal $O(\log_B N + K \ B)$ I/O's while keeping the space complexity optimal.

2.3.1.3 Query Algorithm

The query algorithm for the BBIO tree $\mathscr{T}$ is very simple and mimics the query algorithm for the binary interval tree T. Given a query point q, we perform the following recursive process starting from the root of $\mathscr{T}$. For the current node u, we read u from disk. Now consider the subtree T_u implicitly built on the small nodes in u by the binary-search scheme. Using the same binary-search scheme, we follow a root-to-leaf path in T_u. Let r be the current small node of T_u being visited, with key value m. If $q = m$, then we report all intervals in the left (or equivalently, right) list of r and stop. (We can stop here for the following reasons. (1) Even some descendent of r has the same key value m, such descendent must have empty left and right lists, since if there are intervals containing m, they must be assigned to r (or some small node higher than r) before being assigned to that descendent. (2) For any non-empty descendent of r, the stored intervals are either entirely to the left or entirely to the right of $m = q$, and thus cannot contain q.) If $q < m$, we scan and report the intervals in the left list of r, until the first interval with the left endpoint larger than q is encountered. Recall that the left lists are sorted by increasing left endpoint values. After that, we proceed to the left child of r in T_u. Similarly, if $q \quad m$, we scan and report the intervals in the right list of r, until the first interval with the right endpoint smaller than q is encountered. Then we proceed to the right child of r in T_u. At the end, if q is not equal to any key in T_u, the binary search on the $Bf-1$ keys locates q in one of the Bf slabs. We then visit the child node of u in $\mathscr{T}$ which corresponds to that slab, and apply the same process recursively. Finally, when we reach a leaf node of $\mathscr{T}$, we check the $O(B)$ intervals stored to report those that contain q, and stop.

Since the height of the tree $\mathscr{T}$ is $O(\log_B N)$, we only visit $O(\log_B N)$ nodes of $\mathscr{T}$. We also visit the left and right lists for reporting intervals. Since we always report the intervals in an *output-sensitive* way, this reporting cost is roughly $O(K \ B)$, where K is the number of intervals reported. However, it is possible

that we spend one I/O to read the first block of a left/right list but only very few ($<< B$) intervals are reported. In the worst case, all left/right lists visited result in such *underfull reported blocks* and this I/O cost is $O(\log_2 \frac{N}{B})$, because we visit one left or right list per small node and the total number of small nodes visited is $O(\log_2 \frac{N}{B})$ (this is the height of the balanced binary interval tree T obtained by "concatenating" the small-node subtrees T_u's in all internal nodes u's of $\mathscr{T}$). Therefore the overall worst-case I/O cost is $O(\log_2 \frac{N}{B} + K \ B)$.

We can improve the worst-case I/O query bound. The idea is to check a left/right list of a small node from disk *only when* it is guaranteed that at least *one full block* is reported from that list; the underfull reported blocks of a node u of $\mathscr{T}$ are collectively taken care of by an additional *corner structure* [54] associated with u. A corner structure can store t intervals in optimal space of $O(t \ B)$ disk blocks, where t is restricted to be at most $O(B^2)$, so that an interval stabbing query can be answered in optimal $O(k \ B + 1)$ I/O's, where k is the number of intervals reported from the corner structure. Assuming all t intervals can fit in main memory during preprocessing, a corner structure can be built in optimal $O(t \ B)$ I/O's. We refer to [54] for a complete description of the corner structure.

We incorporate the corner structure into the BBIO tree $\mathscr{T}$ as follows. For each internal node u of $\mathscr{T}$, we remove the first block from each left and right lists of each small node in u, and collect all these removed intervals (with duplications eliminated) into a single corner structure associated with u; if a left/right list has no more than B intervals then the list becomes empty. We also store in u a "guarding value" for each left/right list of u. For a left list, this guarding value is the smallest left endpoint value among the *remaining* intervals still kept in the left list (*i.e.*, the $(B+1)$-st smallest left endpoint value in the *original* left list); for a right list, this value is the largest right endpoint value among the remaining intervals kept (*i.e.*, the $(B+1)$-st largest right endpoint value in the *original* right list). Recall that each left list is sorted by increasing left endpoint values and symmetrically for each right list. Observe that u has $2(Bf-1)$ left and right lists and $Bf = \Theta(B)$, so there are $\Theta(B)$ lists in total, each contributing at most a block of B intervals to the corner structure of u. Therefore, the corner structure of u has $O(B^2)$ intervals, satisfying the restriction of the corner structure. Also, the overall space needed is still optimal $O(N \ B)$ disk blocks.

The query algorithm is basically the same as before, with the following modification. If the current node u of $\mathscr{T}$ is an internal node, then we first query the corner structure of u. A left list of u is checked from disk only when the query value q is larger than or equal to the guarding value of that list; similarly for the right list. In this way, although a left/right list might be checked using one I/O to report very few ($<< B$) intervals, it is ensured that in this case the *original first block* of that list is also reported, from the corner structure of u. Therefore we can charge this one underfull I/O cost to the one I/O cost needed to report such first full block (i.e., reporting the first full block needs one I/O; we can multiply this one I/O cost by 2, so that the additional one I/O can be used to pay for the one I/O cost of the underfull block). This means that the overall underfull I/O cost can be charged to the $K \ B$ term of the reporting cost (with some constant factor), so that the overall query cost is optimal $O(\log_B N + K \ B)$ I/O's.

3 SCIENTIFIC VISUALIZATION

Here, we review out-of-core work done in the area of scientific visualization. In particular, we cover recent work in I/O-efficient volume rendering, isosurface computation, and streamline computation. Since 1997, this area of research has advanced considerably, although it is still an active research area. The techniques described below make it possible to perform basic visualization techniques on large datasets. Unfortunately, some of the techniques have substantial disk and time overhead. Also, often the original format of the data is not suited for visualization tasks, leading to expensive pre-processing steps, which often require some form

of data replication. Few of the techniques described below are suited for interactive use, and the development of multi-resolution approaches that would allow for scalable visualization techniques is still an elusive goal.

Cox and Ellsworth [29] propose a framework for out-of-core scientific visualization systems based on application-controlled demand paging. The paper builds on the fact that many important visualization tasks (including computing streamlines, streaklines, particle traces, and cutting planes) only need touch a small portion of large datasets at a time. Thus, the authors realize that it should be possible to *page in* the necessary data on demand. Unfortunately, as the paper shows, the operating system paging sub-system is not effective for such visualization tasks, and leads to inferior performance. Based on these premises and observations, the authors propose a set of I/O optimizations that lead to substantial improvements in computation times. The authors modified the I/O subsystem of the visualization applications to explicitly take into account the read and caching operations. Cox and Ellsworth report on several effective optimizations. First, they show that controlling the page size, in particular using page sizes smaller than those used by the operating system, leads to improved performance since using larger page sizes leads to wasted space in main memory. The second main optimization is to load data in alternative storage format (i.e., 3-dimensional data stored in sub-cubes), which more naturally captures the natural shape of underlying scientific data. Furthermore, their experiments show that the same techniques are effective for remote visualization, since less data needs to be transmitted over the network and cached at the client machine.

Ueng et al. [82] present a related approach. In their work Ueng et al. focussed on computing streamlines of large unstructured grids, and they use an approach somewhat similar to Cox and Ellsworth in that the idea is to perform on-demand loading of the data necessary to compute a given streamline. Instead of changing the way the operating system handles the I/O, these authors decide to modify the organization of the actual data on disk, and to come up with optimized code for the task at hand. They use an octree partition to restructure unstructured grids to optimize the computation of streamlines. Their approach involves a preprocessing step, which builds an octree that is used to partition the unstructured cells in disk, and an interactive step, which computes the streamlines by loading the octree cells on demand. In their work, they propose a top-down out-of-core preprocessing algorithm for building the octree. Their algorithm is based on propagating the cell (tetrahedra) insertions on the octree from the root to the leaves in phases. In each phase (recursively), octree nodes that need to be subdivided create their children and distribute the cells among them based on the fact that cells are contained inside the octree node. Each cell is replicated on the octree nodes that it would potentially intersect. For the actual streamline computation, their system allows for the concurrent computation of mutliple streamlines at the same time based on user input. It uses a multi-threaded architecture with a set of streamline objects, three scheduling queue (wait, ready, and finished), free memory pool and a table of loaded octants.

Leutenegger and Ma [59] propose to use R-trees [48] to optimize searching operations on large unstructured datasets. They arge that octrees (such as those used by Ueng et al. [82]) are not suited for storing unstructured data because of imbalance in the structure of such data making it hard to efficiently pack the octree in disk. Furthermore, they also argue that the low fan-out of octrees leads to a large number of internal nodes, which force applications into many unnecessary I/O fetches. Leutenegger and Ma use an R-tree for storing tetrahedral data, and present experiemntal results of their method on the implementation of a multi-resolution splatting-based volume renderer.

Pascucci and Frank [67] describe a scheme for defining hierarchical indices over very large regular grids that leads to efficient disk data layout. Their approach is based on the use of a space-filling curve for defining the data layout and indexing. In particular, they propose an indexing scheme for the Lebesgue Curve which can be simply and efficient computed by using bit masking, shifting, and addition. They show the usefulness of their approach in a progresive (real-time) slicing application which exploits their indexing framework for

the multi-resolution computation of arbitrary slices of very large datasets (one example in the paper has approximately one half tera nodes).

Bruckschen et al. [13] describes a technique for real-time particle traces of large time-varying datasets. They argue that it is not possible to perform this computation in real-time on demand, and propose a solution where the basic idea is to pre-compute the traces from fixed positions located on a regular grid, and to save the results for efficient disk access in a way similar to Pascucci and Frank [67]. Their system has two main components, a particle tracer and encoder, which runs as a preprocessing step, and a renderer, which interactively reads the precomputed particle traces. Their particle tracer computes traces for a whole sequence of time steps by considering the data in blocks. It works by injectign particles at grid locations and computing their new positions until the particles have left the currrent block.

Chiang and Silva [18, 20] proposed algorithms for out-of-core isosurface generation of unstructured grids. Isosurface generation can be seen as an interval stabbing problem [22] as follows: first, for each cell, produce an interval $I = [\min, \max]$ where min and max are the minimum and maximum among the scalar values of the vertices of the cell. Then a cell is active if and only if its interval contains the isovalue q. This reduces the active-cell searching problem to that of *interval search*: given a set of intervals in 1D, build a data structure to efficiently report all intervals containing a query point q. Secondly, the interval search problem is solved optimally by using the main-memory interval tree [33]. The first *out-of-core* isosurface technique was given by Chiang and Silva [18]. They follow the ideas of Cignoni et al. [22], but use the I/O-optimal interval tree [6] to solve the interval search problem. An interesting aspect of the work is that even the preprocessing is assumed to be performed completely on a machine with limited main memory. With this technique, datasets much larger than main memory can be visualized efficiently. Though this technique is quite fast in terms of actually computing the isosurfaces, the associated disk and preprocessing overhead is substantial. Later, Chiang et al. [20] further improved (i.e., reduced) the disk space overhead and the preprocessing time of Chiang and Silva [18], at the cost of slightly increasing the isosurface query time, by developing a *two-level* indexing scheme, the *meta-cell* technique, and the *BBIO* tree which is used to index the meta-cells. A meta-cell is simply a collection of contiguous cells, which is saved together for fast disk access.

Along the same lines, Sulatycke and Ghose [77] describe an extension of Cignoni et al. [22] for out-of-core isosurface extraction. Their work does not propose an optimal extension, but instead proposes a scheme that simply adapts the in-core data structure to an out-of-core setting. The authors also describe a multi-threaded implementation that aims to hide the disk I/O overhead by overlapping computation with I/O. Basically, the authors have an I/O thread that reads the active cells from disk, and several isosurface computation threads. Experimental results on relatively small regular grids are given in the paper.

Bajaj et al. [8] also proposes a technique for out-of-core isosurface extraction. Their technique is an extension of their *seed*-based technique for efficient isosurface extraction [7]. The seed-based approach works by saving a small set of seed cells, which can be used as starting points for an arbitrary isosurface by simple contour propagation. In the out-of-core adaptation, the basic idea is to separate the cells along the functional value, storing ranges of contiguous ranges together in disk. In their paper, the authors also describe how to parallelize their algorithm, and show the performance of their techniques using large regular grids.

Sutton and Hansen [78] propose the T-BON (Temporal Branch-On-Need Octree) technique for fast extraction of isosurfaces of time-varying datasets. The preprocessing phase of their technique builds a branch-on-need octree [87] for each time step and stores it to disk in two parts. One part contains the structure of the octree and does not depend at all on the specific time step. Time-step specific data is saved separately (including the extreme values). During querying, the tree is recursively traversed, taking into

account the query timestep and isovalue, and brought into memory until all the information (including all the cell data) has been read. Then, a second pass is performed to actually compute the isosurface. Their technique also uses meta-cells (or data bricking) [20] to optimize the I/O transfers.

Shen et al. [75] proposes a different encoding for time-varying datasets. Their TSP (Time-Space Partitioning) tree encodes in a single data structure the changes from one time-step to another. Each node of the octree has not only a spatial extend, but also a time interval. If the data changes in time, a node is refined by its children, which refine the changes on the data, and is annotated with its valid time range. They use the TSP tree to perform out-of-core volume rendering of large volumes. One of the nice properties is that because of its encoding, it is possible to very efficiently page the data in from disk when rendering time sequences.

Farias and Silva [40] presents a set of techniques for the direct volume rendering of arbitrarily large unstructured grids on machines with limited memory. One of the techniques described in the paper is a memory-insensitive approach which works by traversing each cell in the dataset, one at a time, sampling its ray span (in general a ray would intersect a convex cell twice) and saving two *fragment* entries per cell and pixel covered. The algorithm then performs an external sort using the pixel as the primary key, and the depth of the fragment as the secondary key, which leads to the correct ray stabbing order that exactly captures the information necessary to perform the rendering. The other technique described is more involved (but more efficient) and involves extending the ZWEEP algorithm [39] to an out-of-core setting.

The main idea of the (in-core) ZSWEEP algorithm is to sweep the data with a plane parallel to the viewing plane in order of increasing z, *projecting* the faces of cells that are incident to vertices as they are encountered by the sweep plane. ZSWEEP's face projection consists of computing the intersection of the ray emanating from each pixel, and store their z-value, and other auxiliary information, in *sorted* order in a list of intersections for the given pixel. The actual lighting calculations are deferred to a later phase. Compositing is performed as the "target Z" plane is reached. This algorithm exploits the implicit (approximate) global ordering that the z-ordering of the vertices induces on the cells that are incident on them, thus leading to only a very small number of ray intersection are done out of order; and the use of early compositing which makes the memory footprint of the algorithm quite small. There are two sources of main memory usage in ZSWEEP: the pixel intersection lists, and the actual dataset. The basic idea in the out-of-core technique is to break the dataset into chunks of fixed size (using ideas of the meta-cell work described in Chiang et al. [20]), which can be rendered independently without using more than a constant amount of memory. To further limit the amount of memory necessary, their algorithm subdivides the screen into tiles, and for each tile, which are rendered in chunks that project into it in a front-to-back order, thus enabling the exact same optimizations which can be used with the in-core ZSWEEP algorithm.

4 SURFACE SIMPLIFICATION

In this section we review recent work on out-of-core simplification. In particular, we will focus on methods for simplifying large *triangle meshes*, as these are the most common surface representation for computer graphics. As data sets have grown rapidly in recent years, out-of-core simplification has become an increasingly important tool for dealing with large data. Indeed, many conventional in-core algorithms for visualization, data analysis, geometric processing, etc., cannot operate on today's massive data sets, and are furthermore difficult to extend to work out of core. Thus, simplification is needed to reduce the (often oversampled) data set so that it fits in main memory. As we have already seen in previous sections, even though some methods have been developed for out-of-core visualization and processing, handling billion-triangle meshes, such as those produced by high resolution range scanning [60] and scientific simulations [66], is still

challenging. Therefore many methods benefit from having either a reduced (albeit still large and accurate) version of a surface, or having a multiresolution representation produced using out-of-core simplification.

It is somewhat ironic that, whereas simplification has for a long time been relied upon for dealing with complex meshes, for large enough data sets simplification itself becomes impractical, if not impossible. Typical in-core simplification techniques, which require storing the entire full-resolution mesh in main memory, can handle meshes on the order of a few million triangles on current workstations; two to three orders of magnitude smaller than many data sets available today. To address this problem, several techniques for out-of-core simplification have been proposed recently, and we will here cover most of the methods published to date.

One reason why few algorithms exist for out-of-core simplification is that the majority of previous methods for in-core simplification are ill-suited to work in the out-of-core setting. The prevailing approach to in-core simplification is to iteratively perform a sequence of local mesh coarsening operations, e.g., edge collapse, vertex removal, face clustering, etc., that locally simplify the mesh, e.g., by removing a single vertex. The order of operations performed is typically determined by their impact on the quality of the mesh, as measured using some error metric, and simplification then proceeds in a greedy fashion by always performing the operation that incurs the lowest error. Typical error metrics are based on quantities such as mesh-to-mesh distance, local curvature, triangle shape, valence, an so on. In order to evaluate (and re-evaluate) the metric and to keep track of which simplices (i.e., vertices, edges, and triangles) to eliminate in each coarsening operation, virtually all in-core simplification methods rely on direct access to information about the connectivity (i.e., adjacency information) and geometry in a neighborhood around each simplex. As a result, these methods require explicit data structures for maintaining connectivity, as well as a priority queue of coarsening operations, which amounts to $O(n)$ in-core storage for a mesh of size n. Clearly this limits the size of models that can be simplified in core. Simply offloading the mesh to disk and using virtual memory techniques for transparent access is seldom a realistic option, as the poor locality of greedy simplification results in scattered accesses to the mesh and excessive thrashing. For out-of-core simplification to be viable, such random accesses must be avoided at all costs. As a result, many out-of-core methods make use of a *triangle soup* mesh representation, where each triangle is represented independently as a triplet of vertex coordinates. In contrast, most in-core methods use some form of *indexed mesh* representation, where triangles are specified as indices into a non-redundant list of vertices.

There are many different ways to simplify surfaces. Popular coarsening operations for triangle meshes include vertex removal [73], edge collapse [53], half-edge collapse [58], triangle collapse [49], vertex pair contraction [44], and vertex clustering [71]. While these operations vary in complexity and generality, they all have one thing in common in that they partition the set of vertices from the input mesh by grouping them into clusters (Figure 2).[3] The simplified mesh is formed by choosing a single vertex to represent each cluster (either by selecting one from the input mesh or by computing a new, optimal position). For example, the edge collapse algorithm conceptually forms a forest of binary trees (the clusters) in which each edge collapse corresponds to merging two children into a single parent. Here the cluster representatives are the roots of the binary trees. In the end, it matters little what coarsening operation is used since the set partition uniquely defines the resulting connectivity, i.e., only those triangles whose vertices belong to three different clusters "survive" the simplification. In this sense, mesh simplification can be reduced to a set partioning problem, together with rules for choosing the position of each cluster's representative vertex, and we will examine how different out-of-core methods perform this partitioning. Ideally the partitioning is done so as to minimize the given error measure, although because of the complexity of this optimization problem

[3] Technically vertex removal is a generalization of half-edge collapse with optional edge flipping. Due to its ability to arbitrarily modify the connectivity, vertex removal does not produce a canonical partition of the set of vertices.

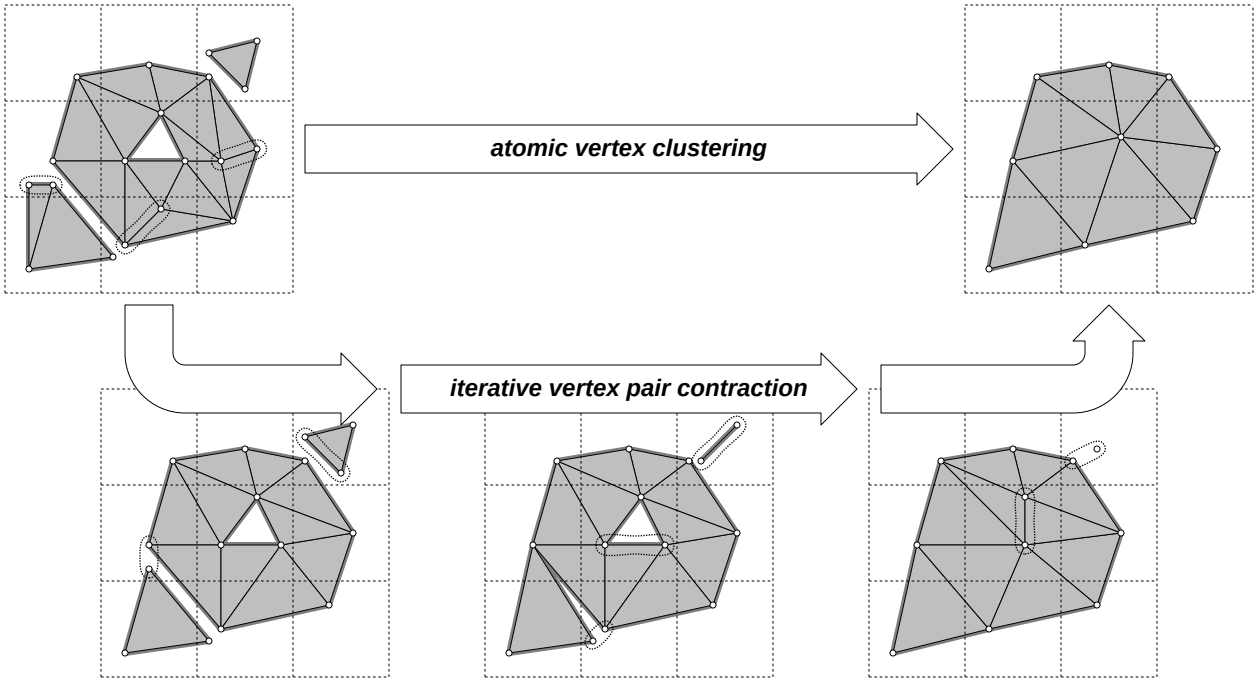


Figure 2: Vertex clustering on a 2D uniform grid as a single atomic operation (top) and as multiple pair contractions (bottom). The dashed lines represent the space-partitioning grid, while vertex pairs are indicated using dotted lines. Note that spatial clustering can lead to significant topological simplification.

heuristics are often used. There are currently two distinct approaches to out-of-core simplification, based on *spatial clustering* and *surface segmentation*. Within these two general categories, we will also distinguish between *uniform* and *adaptive* partitioning. We describe different methods within these frameworks below, and conclude with a comparison of these general techniques.

4.1 Spatial Clustering

Clustering decisions can be based on either the connectivity or the geometry of the mesh, or both. Because computing and maintaining the connectivity of a large mesh out of core can often be a difficult task in and of itself, perhaps the simplest approach to clustering vertices is based solely on *spatial partitioning*. The main idea behind this technique is to partition the space that the surface is embedded in, i.e., $\mathbb{R}^3$, into simple convex 3D regions, and to merge the vertices of the input mesh that fall in the same region. Because the mesh geometry is often specified in a Cartesian coordinate system, the most straightforward space partitioning is given by a rectilinear grid (Figure 2). Rossignac and Borrel [71] used such a grid to cluster vertices in an in-core algorithm. However, the metrics used in their algorithm rely on full connectivity information. In addition, a ranking phase is needed in which the most "important" vertex in each cluster is identified, and their method, as stated, is therefore not well suited for the out-of-core setting. Nevertheless, Rossignac and Borrel's original clustering algorithm is the basis for many of the out-of-core methods discussed below. We note that their algorithm makes use of a uniform grid to partition space, and we will discuss out-of-core methods for uniform clustering first.

4.1.1 Uniform Spatial Clustering

To extend the clustering algorithm in [71] to the out-of-core setting, Lindstrom [61] proposed using Garland and Heckbert's *quadric error metric* [44] to measure error. Lindstrom's method, called *OoCS*, works by scanning a triangle soup representation of the mesh, one triangle at a time, and computing a quadric matrix $\mathbf{Q}_t$ for each triangle t. Using an in-core sparse grid representation (e.g., a dynamic hash table), the three vertices of a triangle are quickly mapped to their respective grid cells, and $\mathbf{Q}_t$ is then distributed to these cells. This depositing of quadric matrices is done for each of the triangle's vertices whether they belong to one, two, or three different clusters. However, as mentioned earlier, only those triangles that span three different clusters survive the simplification, and the remaining degenerate ones are not output. After each input triangle has been read, what remains is a list of simplified triangles (specified as vertex indices) and a list of quadric matrices for the occupied grid cells. For each quadric matrix, an optimal vertex position is computed that minimizes the quadric error [44], and the resulting indexed mesh is then output.

Lindstrom's algorithm runs in linear time in the size of the input and expected linear time in the output. As such, the method is efficient both in theory and practice, and is able to process on the order of a quarter million triangles per second on a typical PC. While the algorithm can simplify arbitrarily large meshes, it requires enough core memory to store the simplified mesh, which limits the accuracy of the output mesh. To overcome this limitation, Lindstrom and Silva [62] proposed an extension of OoCS that performs all computations on disk, and that requires only a constant, small amount of memory. Their approach is to replace all in-core random accesses to grid cells (i.e., hash lookups and matrix updates) with coherent sequential disk accesses, by storing all information associated with a grid cell together on disk. This is accomplished by first mapping vertices to grid cells (as before) and writing partial per-cluster quadric information in the form of plane equations to disk. This step is followed by a fast external sort (Section 2.2.1) on grid cell ID of the quadric information, after which the sorted file is traversed sequentially and quadric matrices are accumulated and converted to optimal vertex coordinates. Finally, three sequential scan-and-replace steps, each involving an external sort, are performed on the list of output triangles, in which cluster IDs are replaced with indices into the list of vertices.

Because of the use of spatial partitioning and quadric errors, no explicit connectivity information is needed in [61, 62]. In spite of this, the end result is identical to what Garland and Heckbert's edge collapse algorithm [44] would produce if the same vertex set partitioning were used. On the downside, however, is that topological features such as surface boundaries and nonmanifold edges, as well as geometric features such as sharp edges are not accounted for. To address this, Lindstrom and Silva [62] suggested computing tangential errors in addition to errors normal to the surface. These tangential errors cancel out for manifold edges in flat areas, but penalize deviation from sharp edges and boundaries. As a result, boundary, nonmanifold, and sharp edges can be accounted for without requiring explicit connectivity information. Another potential drawback of connectivity oblivious simplification—and most non-iterative vertex clustering algorithms in general—is that the topology of the surface is not necessarily preserved, and nonmanifold simplices may even be introduced. On the other hand, for very large and complex surfaces, modest topology simplification may be desirable or even necessary to remove noise and unimportant features that would otherwise consume precious triangles.

4.1.2 Adaptive Spatial Clustering

The general spatial clustering algorithm discussed above does not require a rectilinear partitioning of space. In fact, the 3D space-partitioning mesh does not even have to be conforming (i.e., without cracks or T-junctions), nor do the cells themselves have to be convex or even connected (although that may be prefer-

(a) Original (b) Edge collapse [63] (c) Uniform clustering [61]

Figure 3: Base of Happy Buddha model, simplified from 1.1 million to 16 thousand triangles. Notice the jagged edges and notches in (c) caused by aliasing from using a coarse uniform grid. Most of these artifacts are due to the geometry and connectivity "filtering" being decoupled, and can be remedied by flipping edges.

able). Because the amount of detail often varies over a surface, it may be desirable to adapt the grid cells to the surface shape, such that a larger number of smaller cells are used to partition detailed regions of the surface, while relatively larger cells can be used to cluster vertices in flat regions.

The advantage of producing an *adaptive partition* was first demonstrated by Shaffer and Garland in [74]. Their method makes two passes instead of one over the input mesh. The first pass is similar to the OoCS algorithm [61], but in which a uniform grid is used to accumulate both primal (distance-to-face) and dual (distance-to-vertex) quadric information. Based on this quadric information, a principal component analysis (PCA) is performed that introduces split planes that better partition the vertex clusters than the uniform grid. These split planes, which are organized hierarchically in a binary space partitioning (BSP) tree, are then used to cluster vertices in a second pass over the input data. In addition to superior qualitative results over [61], Shaffer and Garland report a 20% average reduction in error. These improvements come at the expense of higher memory requirements and slower simplification speed.

In addition to storing the BSP-tree, a priority queue, and both primal and dual quadrics, Shaffer and Garland's method also requires a denser uniform grid (than [61]) in order to capture detailed enough information to construct good split planes. This memory overhead can largely be avoided by refining the grid adaptively via multiple passes over the input, as suggested by Fei et al. [41]. They propose uniform clustering as a first step, after which the resulting quadric matrices are analyzed to determine the locations of sharp creases and other surface details. This step is similar to the PCA step in [74]. In each cell where higher resolution is needed, an additional split plane is inserted, and another pass over the input is made (processing only triangles in refined cells). Finally, an edge collapse pass is made to further coarsen smooth regions by making use of the already computed quadric matrices. A similar two-phase hybrid clustering and edge collapse method has recently been proposed by Garland and Shaffer [45].

The uniform grid partitioning scheme is somewhat sensitive to translation and rotation of the grid, and for coarse grids aliasing artifacts are common (see Figure 3). Inspired by work in image and digital signal processing, Fei et al. [42] propose using two interleaved grids, offset by half a grid cell in each direction, to combat such aliasing. They point out that detailed surface regions are relatively more sensitive to grid translation, and by clustering the input on both grids and measuring the local similarity of the two resulting meshes (again using quadric errors) they estimate the amount of detail in each cell. Where there is a large amount of detail, simplified parts from both meshes are merged in a retriangulation step. Contrary to [41, 45, 74], this semi-adaptive technique requires only a single pass over the input.

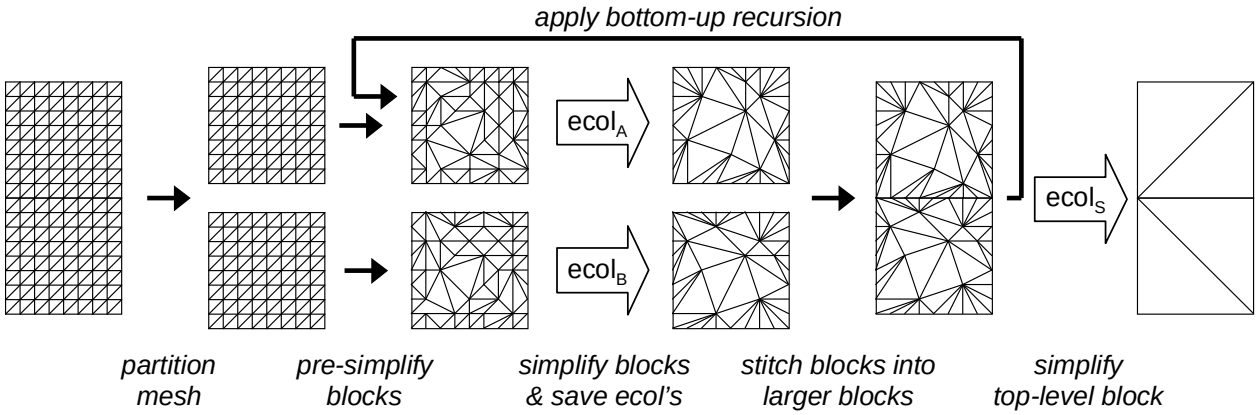


Figure 4: Height field simplification based on hierarchical, uniform surface segmentation and edge collapse (figure courtesy of Hugues Hoppe).

4.2 Surface Segmentation

Spatial clustering fundamentally works by finely partitioning the space that the surface lies in. The method of *surface segmentation*, on the other hand, partitions the *surface* itself into pieces small enough that they can be further processed independently in core. Each surface patch can thus be simplified in core to a given level of error using a high quality simplification technique, such as edge collapse, and the coarsened patches are then "stitched" back together. From a vertex set partition standpoint, surface segmentation provides a coarse division of the vertices into smaller sets, which are then further refined in core and ultimately collapsed. As in spatial clustering, surface segmentation can be uniform, e.g., by partitioning the surface over a uniform grid, or adaptive, e.g., by cutting the surface along feature lines. We begin by discussing uniform segmentation techniques.

4.2.1 Uniform Surface Segmentation

Hoppe [52] described one of the first out-of-core simplification techniques based on surface segmentation for the special case of height fields. His method performs a 2D spatial division of a regularly gridded terrain into several rectangular blocks, which are simplified in core one at a time using edge collapse until a given error tolerance is exceeded. By disallowing any modifications to block boundaries, adjacent blocks can then be quickly stitched together in a hierarchical fashion to form larger blocks, which are then considered for further simplification. This allows seams between sibling blocks to be coarsened higher up in in the hierarchy, and by increasing the error tolerance on subsequent levels a progressively coarser approximation is obtained (see Figure 4). Hoppe's method was later extended to general triangle meshes by Prince [69], who uses a 3D uniform grid to partition space and segment the surface. Both of these methods have the advantage of producing not a single static approximation but a multiresolution representation of the mesh—a *progressive mesh* [50]—which supports adaptive refinement, e.g., for view-dependent rendering. While being significantly slower (by about two orders of magnitude) and requiring more (although possibly controllable) memory than most spatial clustering techniques, the improvement in quality afforded by error-driven edge collapse can be substantial.

Bernardini et al. [11] developed a strategy similar to [52, 69]. Rather than constructing a multiresolution hierarchy, however, a single level of detail is produced. Seams between patches are coarsened by shifting the

single-resolution grid (somewhat akin to the approach in [42]) after all patches in the current grid have been simplified. In between the fine granularity provided by a progressive mesh [69] and the single-resolution meshes created in [11], Erikson et al. [38] proposed using a static, discrete level of detail for each node in the spatial hierarchy. As in [11, 52, 69], a rectilinear grid is used for segmentation in Erikson's method.

One of the downsides of the surface segmentation techniques described above is the requirement that patch boundaries be left intact, which necessitates additional passes to coarsen the patch seams. This requirement can be avoided using the *OEMM* mesh data structure proposed by Cignoni et al. [23]. As in [69], an octree subdivision of space is made. However, when processing the surface patch for a node in this tree, adjacent nodes are loaded as well, allowing edges to be collapsed across node boundaries. Some extra bookkeeping is done to determine which vertices are available for consideration of an edge collapse, as well as which vertices can be modified or removed. This general data structure supports not only out-of-core simplification but also editing, visualization, and other types of processing. Other improvements over [69] include the ability to adapt the octree hierarchy, such that child nodes are collapsed only when the parent cell contains a sufficiently small number of triangles.

4.2.2 Adaptive Surface Segmentation

Some of the surface segmentation methods discussed so far already support adapting the vertex set partition to the shape of the surface (recall that this partitioning is the goal of triangle mesh simplification). For example, simplification *within* a patch is generally adaptive, and in [23] the octree space partition adapts to the local complexity of the surface. Still, using the methods above, the surface is always segmented by a small set of predefined cut planes, typically defined by an axis-aligned rectilinear grid. In contrast, the out-of-core algorithm described by El-Sana and Chiang [34] segments the surface solely based on its shape. Their technique is a true out-of-core implementation of the general error-driven edge collapse algorithm. Like in other methods based on surface segmentation, edge collapses are done in batches by coarsening patches, called *sub-meshes*, up to a specified error tolerance, while making sure patch boundaries are left intact. In contrast to previous methods, however, the patch boundaries are not defined via spatial partitioning, but by a set of edges whose collapse costs exceed a given error threshold. Thus patch boundaries are not artificially constrained from being coarsened, but rather delineate important features in the mesh that are to be preserved. Rather than finding such boundaries explicitly, El-Sana and Chiang sort all edges by error and load as many of the lowest-cost edges, called *spanning edges*, and their incident triangles as can fit in main memory. The highest-cost spanning edge then sets the error threshold for the current iteration of in-core simplification. In this process, patches of triangles around spanning edges are loaded and merged whenever possible, creating sub-meshes that can be simplified independently by collapsing their spanning edges. When the sub-meshes have been simplified, their edges are re-inserted into the priority queue, and another iteration of sub-mesh construction and simplification is performed. As in [69], the final output of the algorithm is a multiresolution mesh.

To support incidence queries and an on-disk priority queue, El-Sana and Chiang make use of efficient search data structures such as B-trees (see also Section 2). Their method is surprisingly fast and works well when sufficient memory is available to keep I/O traffic to a minimum and enough disk space exists for storing connectivity information and an edge collapse queue for the entire input mesh.

Iterative simplification of extremely large meshes can be a time consuming task, especially when the desired approximation is very coarse. This is because the number of coarsening operations required is roughly proportional to the size of the *input*. In such situations, it may be preferable to perform the inverse of simplification, *refinement*, by starting from a coarse representation and adding only a modest amount detail, in which case the number of refinement operations is determined by the size of the *output*. Choudhury and

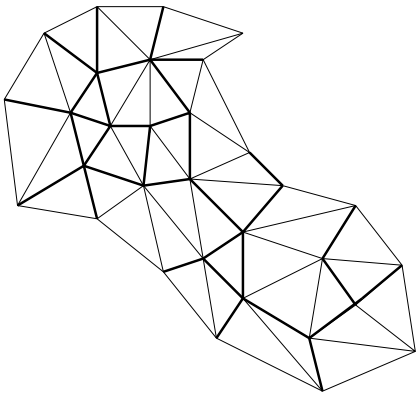


Figure 5: Submesh for a set of spanning edges (shown as thick lines) used in El-Sana and Chiang's simplification method.

Watson [21] describe an out-of-core version of the *RSimp* refinement method [12] that they call *VMRSimp* (for *Virtual Memory RSimp*). Rather than using sophisticated out-of-core data structures, VMRSimp relies on the virtual memory mechanism of the operating system to handle data paging. As in the original method, VMRSimp works by incrementally refining a partition of the set of input triangles. These triangle sets are connected and constitute an adaptive segmentation of the surface. The choice of which triangle patch to refine is based on the amount of normal variation within it (essentially a measure of curvature defined over a region of the surface). VMRSimp improves upon RSimp by storing the vertices and triangles in each patch contiguously in virtual memory. When a patch is split in two, its constituent primitives are reorganized to preserve this locality. As patches are refined and become smaller, the locality of reference is effectively increased, which makes the virtual memory approach viable. Finally, when the desired level of complexity is reached, a representative vertex is computed for each patch by minimizing a quadric error function. Even though the simplification happens in "reverse," this method, like all others discussed here, is yet another instance of vertex set partitioning and collapse.

4.3 Summary of Simplification Techniques

In this section we have seen a variety of different out-of-core surface simplification techniques, and we discussed how these methods partition and later collapse the set of mesh vertices. All of these methods have shown to be effective for simplifying surfaces that are too large to fit in main memory. The methods do have their own strengths and weaknesses, however, and we would like to conclude with some suggestions for when to a certain technique may be preferable over another.

Whereas spatial clustering is generally the fastest method for simplification, it often produces lower quality approximations than the methods based on surface segmentation. This is because surface segmentation allows partitioning the vertex set based directly on error (except possibly along seams) using an iterative selection of fine-grained coarsening operations. Spatial clustering, on the other hand, typically groups a large number of vertices, based on little or no error information, in a single atomic operation. The incremental nature of surface segmentation also allows constructing a multiresolution representation of the mesh that supports fine-grained adaptive refinement, which is important for view-dependent rendering. Indeed, several of the surface segmentation methods discussed above, including [23, 34, 38, 52, 69], were designed explicitly for view-dependent rendering. Finally, because surface segmentation methods generally maintain connectivity, they support topology-preserving simplification.

category	*method*	*peak mem. usage (MB)*		*peak disk usage (GB)*		*speed*
		theoretical	*example*	*theoretical*	*example*	*(Ktri/s)*
spatial clustering	[61]	$144V_{\text{out}}$	257	0	0	100–250[a]
	[41]	$\simeq 200V_{\text{out}}$	$\simeq 356$	0	0	65–150
	[74]	$\geq 464V_{\text{out}}$	≥ 827	0	0	45–90
	[62]	$O(1)$	8	$120–370V_{\text{in}}$	21–64	30–70
surface segmentation	[23]	$O(1)$	80	$\simeq 68V_{\text{in}}$	12	6–13[b]
	[34]	$O(1)$	128	$\Omega(V_{\text{in}})$	?	5–7
	[21]	$O(1)$[c]	1024	$\geq 86V_{\text{in}}$	≥ 15	4–5
	[69]	$O(1)$	512	$\Omega(V_{\text{in}})$	?	0.8–1

[a] Speed of author's current implementation on an 800 MHz Pentium III.

[b] When the one-time OEMM construction step is included in the simplification time, the effective speed drops to 4–6 Ktri/s.

[c] This method is based on virtual memory. Thus all available memory is generally used during simplification.

Table 1: Simplification results for various methods. These results were obtained or estimated from the original publications, and are only approximate. When estimating memory usage, here expressed in number of input (V_{in}) and output (V_{out}) vertices, we assume that the meshes have twice as many triangles as vertices. The results in the "example" columns correspond to simplifying the St. Matthew data set [60], consisting of 186,810,938 vertices, to 1% of its original size. For methods with constant ($O(1)$) memory usage, we list the memory configuation from the original publication. The disk usage corresponds to the amount of *temporary* space used, and does not include the space needed for the input and output mesh. For the clustering techniques, the speed is measured as the size of the input over the simplification time. Because the surface segmentation methods work incrementally, the size of the output greatly affects the speed. Thus, for these methods the speed is measured in terms of the change in triangle count.

Based on the observations above, it may seem that surface segmentation is always to be favored over spatial clustering. However, the price to pay for higher quality and flexibility is longer simplification times, often higher resource requirements, and less straightforward implementations. To get a better idea of the resource usage (RAM, disk, CPU) for the various methods, we have compiled a table (Table 1) based on numerical data from the authors' own published results. Note that the purpose of this table is not to allow accurate quantitative comparisons between the methods; clearly factors such as quality of implementation, hardware characteristics, amount of resource contention, assumptions made, data sets used, and, most important, what precisely is being measured have a large impact on the results. However, with this in mind, the table gives at least a rough idea of how the methods compare. For example, Table 1 indicates that the spatial clustering methods are on average one to two orders of magnitude faster than the surface segmentation methods. Using the 372-million-triangle St. Matthew model from the Digital Michelangelo Project [60] as a running example, the table suggests a difference between a simplification time of half an hour using [61] and about a week using [69]. For semi-interactive tasks that require periodic user feedback, such as large-isosurface visualization where on-demand surface extraction and simplification are needed, a week's worth of simplification time clearly isn't practical.

In terms of resource usage, most surface segmentation methods make use of as much RAM as possible, while requiring a large amount of disk space for storing the partially simplified mesh, possibly including full connectivity information and a priority queue. For example, all surface segmentation methods discussed above have input-sensitive disk requirements. Finally, certain types of data sets, such as isosurfaces from scientific simulations [66] and medical data [2], can have a very complicated topological structure, resulting

either from noise or intrinsic fine-scale features in the data. In such cases, simplification of the topology is not only desirable but is necessary in order to reduce the complexity of the data set to an acceptable level. By their very nature, spatial clustering based methods are ideally suited for removing such small features and joining spatially close pieces of a surface.

To conclude, we suggest using spatial clustering for very large surfaces when time and space are at a premium. If quality is the prime concern, surface segmentation methods perform favorably, and should be chosen if the goal is to produce a multiresolution or topology-equivalent mesh. For the best of both worlds, we envision that hybrid techniques, such as [41, 45], that combine fast spatial clustering with high-quality iterative simplification, will play an increasingly important role for practical out-of-core simplification.

5 INTERACTIVE RENDERING

The advances in the tools for 3D modeling, simulation, and shape acquisition have led to the generation of very large 3D models. Rendering these models at interactive frame rates has applications in many areas, including entertainment, computer-aided design (CAD), training, simulation, and urban planning.

In this section, we review the out-of-core techniques to render large models at interactive frame rates using machines with small memory. The basic idea is to keep the model on disk, load on demand the parts of the model that the user sees, an display each visible part at a level of detail proportional to its contribution to the image being rendered. The following subsections describe the algorithms to efficiently implement this idea.

5.1 General Issues

Building an Out-Of-Core Representation for a Model At preprocessing time, an out-of-core rendering system builds a representation for the model on disk. The most common representations are bounding volume hierarchies (such as bounding spheres [72]) and space partitioning hierarchies (such as and k-D trees [43, 80], BSP trees [85], and octrees [4, 23, 28, 82, 83]).

Some approaches assume that this preprocessing step is performed on a machine with enough memory to hold the entire model [43, 83]. Others make sure that the preprocessing step can be performed on a machine with small memory [23, 28, 85].

Precomputing Visibility Information One of the key computations at runtime is determining the visible set — the parts of the model the user sees. Some systems precompute from-region visibility, i.e., they split the model into cells, and for each cell they precompute what the user would see from any point within the cell [4, 43, 80]. This approach allows the system to reuse the same visible set for several viewpoints, but it requires long preprocessing times, and may cause bursts of disk activity when the user crosses cell boundaries.

Other systems use from-point visibility, i.e., they determine on-the-fly what the user sees from the current viewpoint [28, 85]. Typically, the only preprocessing required by this approach is the construction of a hierarchical spatial decomposition for the model. Although this approach needs to compute the visible set for every frame, it requires very little preprocessing time, and reduces the risk of bursts of disk activity, because the changes in visibility from viewpoint to viewpoint tend to be much smaller than the changes in visibility from cell to cell.

View-Frustum Culling Since the user typically has a limited field of view, a very simple technique, that perhaps all rendering systems use, is to cull away the parts of the model outside the user's view frustum. If a hierarchical spatial decomposition or a hierarchy of bounding volumes is available, view-frustum culling can be optimized by recursively traversing the hierarchy from the root down to the leaves, and culling away entire subtrees that are outside the user's view frustum [24].

Occlusion Culling Another technique to minimize the geometry that needs to be loaded from disk and sent to the graphics hardware is to cull away geometry that is occluded by other geometry. This is a hard problem to solve exactly [56, 80, 85], but fast and accurate approximations exist [35, 55].

Level-of-Detail Management (or Contribution Culling) The amount of geometry that fits in main memory typically exceeds the interactive rendering capability of the graphics hardware. One approach to alleviate this problem is to reduce the complexity of the geometry sent to the graphics hardware. The idea is to display a certain part of the model using a level of detail (LOD) that is proportional to the part's contribution to the image being rendered. Thus, LOD management is sometimes referred to as contribution culling.

Several level of detail approaches have been used in the various walkthrough systems. Some systems use several static levels of detail, usually 3-10, which are constructed off-line using various simplification techniques [25, 36, 44, 70, 81]. Then at real-time, an appropriate level of detail is selected based on various criteria such as distance from the user's viewpoint and screen space projection. Other systems use a multiresolution hierarchy, which encodes all the levels of detail, and is constructed off-line [31, 37, 50, 51, 64, 89]. Then at real-time, the mesh adapts to the appropriate level of detail in a continuous, coherent manner.

Overlapping Concurrent Tasks Another technique to improve the frame rates at runtime is to perform independent operations in parallel. Many systems use multi-processor machines to overlap visibility computation, rendering, and disk operations [4, 43, 46, 83, 88]. Corrêa et al. [28] show that these operations can also be performed in parallel on single-processor machines by using multiple threads.

Geometry Caching The viewing parameters tend to change smoothly from frame to frame, specially in walkthrough applications. Thus, the changes in the visible set from frame to frame tend to be small. To exploit this coherence, most systems keep in main memory a geometry cache, and update the cache as the viewing parameters change [4, 28, 43, 83, 85]. Typically these systems use a least recently used (LRU) replacement policy, keeping in memory the parts of the model most recently seen.

Speculative Prefetching Although changes in the visible set from frame to frame tend to be small, they are occasionally large, because even small changes in the viewing parameters can cause large visibility changes. Thus, although most frames require to perform few or no disk operations, a common behavior of out-of-core rendering systems is that some frames require to perform more disk operations that can be done during the time to render a frame. The technique to alleviate these bursts of disk activity is to predict (or speculate) what parts of the model are likely to become visible in the next few frames and prefetch them from disk ahead of time. Prefetching amortizes the cost of the bursts of disk operations over the frames that require few or no disk operations, and produces much smoother frame rates [4, 28, 43, 83]. Traditionally, prefetching strategies have relied on from-region visibility algorithms. Recently, Corrêa et al. [28] showed that prefetching can be based on from-point visibility algorithms.

Replacing Geometry with Imagery Image-based rendering techniques such as texture-mapped impostors can be used to accelerate the rendering process [5,30,65,76]. These texture-mapped impostors are generated either in a preprocessing step or at runtime (but not every frame). These techniques are suitable for outdoor models. Textured depth meshes [4, 30, 76] can also be used to replace far geometry. Textured depth meshes are an improvement over texture-mapped impostors, because textured depth meshes provide perspective correction.

5.2 Detailed Discussions

Funkhouser [43] has developed an interactive display system that allows interactive walkthrough large buildings. In an off-line stage a display database is constructed for the given architectural model. The display database stores the building model as a set of objects which are represented at multiple level of detail. It is include a space partition constructed by subdividing the space into cells along the major axis-aligned polygons of the building model. The display database also stores visibility information for each cell. The precomputed visibility determines the set of cells (cell-to-cell visibility) and objects (cell-to-object) which are visible form any cell. The visibility computation is based on the algorithm of Teller and Sequin [80].

In real time the system relies on the precomputed display database to allow interactive walkthrough large building models. For each change in the viewpoint or view direction system performs the following steps.

- It computes the set of potentially visible objects to render. Such set is always a proper subset of the cell-to-object set.
- For each potentially visible object it selects the appropriate level-detail representation for rendering. The screen space projection is used to select the LOD range, and then an optimization algorithm is used to reduce the LOD range to maintain bounded frame rates.
- The potentially visible objects, each in its appropriate level of detail, are sent to the graphics hardware for rendering.

To support the above rendering scheme for large building models, the system manages the display database in an out-of-core manner. The system uses prediction to estimate the observer viewpoint to prefetch objects which are likely to become visible in the upcoming future. The system uses the observer viewpoint, movement, and rotation to determine the observer rang that includes the observer viewpoints possible in the next n frames. The observer range is weighted based on the direction of travel and the solid behavior of the walls.

The cell-to-cell and cell-to-object are used to predict a superset of the objects potentially visible from the observer range. For each frame they computer the set of range cells that include the observer range by performing shortest path search of the cell adjacency graph. Then they add the objects in the cell-to-object visibly of each newly discovered cell to the lookahead set. After adding an object to the lookahead set the system claims all its LODs. The rendering process select the appropriate static level of detail for each object based on precomputed information and the observer position.

Aliaga et al. [4] have presented a system, which renders large complex model at interactive rates. As preprocessing, the input models space is portioned into virtual cells that do not need to coincide with wall or other large occluders. A cull box is placed around each virtual cell. The cull box partitions the space into near (inside the box) and far (outside the box) geometry. Instead of rendering the far geometry, the system generates textured depth mesh for the inside faces of the cell. Then the outside of the box as viewed from the cell center-point. For the near geometry, they compute four level of detail for each object, and select

potential occluders in the preprocess stage. At run time, they cull to the view frustum, cull back facing, and select the appropriate level of detail for the potentially visible objects. To balance the quality of the near and far geometry they have used a pair of error metrics for each cell-a cull box size and the LOD error threshold. The system stores the model in a scene graph hierarchy. They use the model's grouping as the upper layer, and below that they maintain an octree-like bounding volumes. They store the geometry in the leaf nodes in a triangle strips form. Since large fraction of the model database is stored in an external media. The prefetch performed based on the potentially visible near geometry for each cell, which is computed in the preprocessing. At run time, the system maintains a list of cell the user may visit. The prediction algorithm takes into account the user's motion speed, velocity, and view direction.

Correa et al. [28] have developed the iWalk, which allows users to walkthrough large models at interactive rates using typical PC. iWalk has presented a complete out-of-core process that include an out-of-core preprocessing and out-of-core real-time multi-threaded rendering approach. The out-of-core preprocessing algorithm creates a disk hierarchy representation of the input model. The algorithm first breaks the model into sections that fit in main memory, and then incrementally builds the octree on disk. The preprocessing algorithm also generates hierarchical structure file that include information concern the spatial relationship of the nodes in the hierarchy. Hierarchical structure is a small footprint of the models and for that reason they have assumed that it fits in local memory. At run time the algorithm utilizes the created hierarchy in an out-of-core manner for multi-threaded rendering. It uses PLP [55] to determine the set of nodes which are visible form user's viewpoint. For each newly discovered node the system sends a fetch request, which is processed by the fetch thread by loading the node from the disk into a memory cache. They system used least recently used policy for node replacement. To minimize the I/O overhead, a look-ahead thread is used to utilizes the user motion to predict the future user's viewpoint, use the PLP [55] to determine the set of potentially visible nodes, and send fetch request for these nodes.

Varadhan and Manocha [83] has developed an algorithm to render large geometric models at interactive rates. Their algorithm assumes that the scene graph has been constructed and the space was portioned appropriately. However, the algorithm precomputed static levels of detail for each object and associate them with the leaf nodes. At run time the algorithm traverses the scene graph from the root node at each frame. For each visited node it performs a culling tests to determine whether it needs to recursively scan the visited node or not. These culling tests include view frustum culling, simplification culling, and occlusion culling. Upon the completion of the traversal the algorithm computes the list of object representations that need to be rendering in the current frame. They refer to the list of object as the front. As the user changes its view position and direction objects representation in the front list may changes its level of detail or visibility status.

The traversal of the scene graph in an out-of-core manner is achieved by maintaining a scene graph skeleton that includes the nodes, connectivity information, bounding boxes, and error metrics. The resulting skeleton typically include small fraction of the scene graph. At run time, they use two processes, one for rendering and the other manages the disk I/O. During the rendering of one frame the I/O process goes into three stages- continue prefetching for the previous frame, fetching, and prefetching for the current frame. The goal of the prefetching is to increase the hit rate during the fetch stage. The prefetching takes into account the speed and the direction of the users' motion to estimate the appropriate representation for each object and potentially visible objects in the next frames. To further optimize the prefetching process by prioritizing the different object representations in the front the prefetch selects them based on their priority. The least recently used policy is used to remove object representation from the cache.

6 GLOBAL ILLUMINATION

Teller et al. [79] describe a system for computing radiosity solutions of large environments. Their system is based on partitioning the dataset into small pieces, and ordering the radiosity computation in such a way as to minimize the amount of data that needs to be in memory at any given size. They exploit the fact that when computing radiosity computation for a given patch only requires information about other parts of the model that can be seen from that patch, which often is only a small subset of the whole dataset.

Pharr et al. [68] describe techniques for ray tracing very complex scenes. Their work is based on the caching and reodering computations. Their approach uses three different types of caches: ray, geometry, and texture caches. In order to optimize the use of the cache, they developed a specialized scheduler that reorders the way the rendering computations are performed in order to minimize I/O operations by exploiting computational decomposition, ray grouping, and voxel scheduling.

Wald et al. [85, 86] present a real-time ray tracing system for very-large models. Their system is similar in some respects to Pharr et al., and it is also based on the reordering of ray computations (ray grouping) and voxel caching. By exploting parallelism both at the microprocessor level (with MMX/SSE instructions) and at the machine level (PC clusters), Wald et al. are able to compute high-quality renderings of complex scenes in real time (although the images are relatively low resolution).

ACKNOWLEDGEMENTS

We would like to thank Wagner Corrêa (Princeton University) for writing Section 5.1.

References

[1] J. Abello and J. Vitter. *External Memory Algorithms and Visualization*, vol. 50 of *DIMACS Series*. American Mathematical Society, 1999.

[2] M. J. Ackerman. The Visible Human Project. *Proceedings of the IEEE*, 86(3):504–511, Mar. 1998. URL http://www.nlm.nih.gov/research/visible.

[3] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[4] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. H. III, T. Hudson, W. Sturzlinger, R. Bastos, M. Whitton, F. B. Jr., and D. Manocha. MMR: an interactive massive model rendering system using geometric and image-based acceleration. *Symposium on Interactive 3D Graphics*, 199–206. 1999.

[5] D. G. Aliaga and A. A. Lastra. Architectural Walkthroughs Using Portal Textures. *IEEE Visualization '97*, 355–362. 1997.

[6] L. Arge and J. S. Vitter. Optimal Interval Management in External Memory. *Proc. 37th Annu. IEEE Sympos. Found. Comput. Sci.*, 560–569. 1996.

[7] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast Isocontouring for Improved Interactivity. *1996 Volume Visualization Symposium*, 39–46. 1996.

[8] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel Accelerated Isocontouring for Out-of-Core Visualization. *Symposium on Parallel Visualization and Graphics*, 97–104. 1999.

[9] R. Bayer and McCreight. Organization of large ordered indexes. *Acta Inform.*, 1:173–189, 1972.

[10] J. L. Bentley. Multidimensional binary search trees used for associative search ing. *Commun. ACM*, 18(9):509–517, Sep. 1975.

[11] F. Bernardini, J. Mittleman, and H. Rushmeier. Case Study: Scanning Michelangelo's Florentine Pietà. SIGGRAPH 99 Course #8, Aug. 1999. URL http://www.research.ibm.com/pieta.

[12] D. Brodsky and B. Watson. Model Simplification Through Refinement. *Graphics Interface 2000*, 221–228. May 2000.

[13] R. Bruckschen, F. Kuester, B. Hamann, and K. I. Joy. Real-time out-of-core visualization of particle traces. *IEEE Parallel and Large-Data Visualization and Graphics Symposium 2001*, 45–50. 2001.

[14] Y.-J. Chiang. Dynamic and I/O-efficient algorithms for computational geometry and graph problems: theoretical and experimental results. *Ph.D. Thesis, Technical Report CS-95-27*, Dept. Computer Science, Brown University, 1995. Also available at http://cis.poly.edu/chiang/thesis.html.

[15] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Computational Geometry: Theory and Applications*, 9(4):211–236, 1998.

[16] Y.-J. Chiang, R. Farias, C. Silva, and B. Wei. A Unified Infrastructure for Parallel Out-Of-Core Isosurface Extraction and Volume Rendering of Unstructured Grids'. *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 59–66. 2001.

[17] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-Memory Graph Algorithms. *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 139–149. 1995.

[18] Y.-J. Chiang and C. T. Silva. I/O Optimal Isosurface Extraction. *IEEE Visualization 97*, 293–300, Nov. 1997.

[19] Y.-J. Chiang and C. T. Silva. External Memory Techniques for Isosurface Extraction in Scientific Visualization. *External Memory Algorithms and Visualization, DIMACS Series*, 50:247–277, 1999.

[20] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive Out-Of-Core Isosurface Extraction. *IEEE Visualization 98*, 167–174, Oct. 1998.

[21] P. Choudhury and B. Watson. Completely Adaptive Simplification of Massive Meshes. *Tech. Rep. CS-02-09*, Northwestern University, Mar. 2002. URL http://www.cs.northwestern.edu/~watsonb/school/docs/vmrsimp.t%r.pdf.

[22] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, April - June 1997.

[23] P. Cignoni, C. Rocchini, C. Montani, and R. Scopigno. External Memory Management and Simplification of Huge Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 2002. To appear.

[24] J. H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19(10):547–554, Oct. 1976.

[25] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks, Jr., and W. V. Wright. Simplification Envelopes. *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, 119 – 128. August 1996.

[26] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11:121–137, 1979.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd Ed.*. MIT Press, 2001.

[28] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. iWalk: Interactive Out-Of-Core Rendering of Large Models. *Technical Report TR-653-02*, Princeton University, 2002.

[29] M. B. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. *IEEE Visualization 97*, 235–244, Nov. 1997.

[30] L. Darsa, B. Costa, and A. Varshney. Navigating Static Environments Using Image-Space Simplification and Morphing. *Proceedings, 1997 Symposium on Interactive 3D Graphics*, 28 – 30. 1997.

[31] L. De Floriani, P. Magillo, and E. Puppo. Efficient Implementation of Multi-Triangulation. *Proceedings Visualization '98*, 43–50. October 1998.

[32] D. Ebert, H. Hagen, and H. Rushmeier, eds. *IEEE Visualization '98*. IEEE, Oct. 1998.

[33] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.

[34] J. El-Sana and Y.-J. Chiang. External Memory View-Dependent Simplification. *Computer Graphics Forum*, 19(3):139–150, Aug. 2000.

[35] J. El-Sana, N. Sokolovsky, and C. T. Silva. Integrating occlusion culling with view-dependent rendering. *IEEE Visualization 2001*, 371–378. Oct. 2001.

[36] J. El-Sana and A. Varshney. Topology Simplification for Polygonal Virtual Environments. *IEEE Transactions on Visualization and Computer Graphics*, 4, No. 2:133–144, 1998.

[37] J. El-Sana and A. Varshney. Generalized View-Dependent Simplification. *Computer Graphics Forum*, 18(3):83–94, Aug. 1999.

[38] C. Erikson, D. Manocha, and W. V. Baxter III. HLODs for Faster Display of Large Static and Dynamic Environments. *2001 ACM Symposium on Interactive 3D Graphics*, 111–120. Mar. 2001.

[39] R. Farias, J. Mitchell, and C. Silva. ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering. *Volume Visualization Symposium*, 91–99. 2000.

[40] R. Farias and C. T. Silva. Out-Of-Core Rendering of Large, Unstructured Grids. *IEEE Computer Graphics & Applications*, 21(4):42–51, July / August 2001.

[41] G. Fei, K. Cai, B. Guo, and E. Wu. An Adaptive Sampling Scheme for Out-of-Core Simplification. *Computer Graphics Forum*, 21(2):111–119, Jun. 2002.

[42] G. Fei, N. Magnenat-Thalmann, K. Cai, and E. Wu. Detail Calibration for Out-of-Core Model Simplification through Interlaced Sampling. *SIGGRAPH 2002 Conference Abstracts and Applications*, 166. Jul. 2002.

[43] T. A. Funkhouser. Database Management for Interactive Display of Large Architectural Models. *Graphics Interface '96*, 1–8. 1996.

[44] M. Garland and P. S. Heckbert. Surface Simplification Using Quadric Error Metrics. *Proceedings of SIGGRAPH 97*, 209–216. Aug. 1997.

[45] M. Garland and E. Shaffer. A Multiphase Approach to Efficient Surface Simplification. *IEEE Visualization 2002*. Oct. 2002. To appear.

[46] B. Garlick, D. R. Baum, and J. M. Winget. Interactive Viewing of Large Geometric Databases Using Multiprocessor Graphics Workstations. *SIGGRAPH 90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation)*, 239–245. ACM SIGGRAPH, 1990.

[47] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-Memory Computational Geometry. *IEEE Foundations of Comp. Sci.*, 714–723. 1993.

[48] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *Proc. ACM SIGMOD Conf. Principles Database Systems*, 47–57. 1984.

[49] B. Hamann. A Data Reduction Scheme for Triangulated Surfaces. *Computer Aided Geometric Design*, 11(2):197–214, 1994.

[50] H. Hoppe. Progressive Meshes. *Proceedings of SIGGRAPH 96*, 99–108. Aug. 1996.

[51] H. Hoppe. View-Dependent Refinement of Progressive Meshes. *Proceedings of SIGGRAPH '97*, 189 – 197. August 1997.

[52] H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. Ebert et al. [32], 35–42.

[53] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh Optimization. *Proceedings of SIGGRAPH 93*, 19–26. Aug. 1993.

[54] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for Data Models with Constraints and Classes. *Journal of Computer and System Sciences*, 52(3):589–612, 1996.

[55] J. T. Klosowski and C. T. Silva. The Prioritized-Layered Projection Algorithm for Visible Set Estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):108–123, Apr. 2000.

[56] J. T. Klosowski and C. T. Silva. Efficient Conservative Visibility Culling Using the Prioritized-Layered Projection Algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):365–379, Oct. 2001.

[57] D. Knuth. *The Art of Computer Programming Vol. 3: Sorting and Searching*. Addison-Wesley, 1973.

[58] L. Kobbelt, S. Campagna, and H.-P. Seidel. A General Framework for Mesh Decimation. *Graphics Interface '98*, 43–50. Jun. 1998.

[59] S. Leutenegger and K.-L. Ma. *External Memory Algorithms and Visualization*, vol. 50 of *DIMACS Book Series*, chap. Fast Retrieval of Disk-Resident Unstructured Volume Data for Visualization. American Mathematical Society, 1999.

[60] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. *Proceedings of SIGGRAPH 2000*, 131–144. Jul. 2000. URL http://graphics.stanford.edu/projects/mich.

[61] P. Lindstrom. Out-of-Core Simplification of Large Polygonal Models. *Proceedings of SIGGRAPH 2000*, 259–262. 2000.

[62] P. Lindstrom and C. T. Silva. A memory insensitive technique for large model simplification. *IEEE Visualization 2001*, 121–126. Oct. 2001.

[63] P. Lindstrom and G. Turk. Fast and Memory Efficient Polygonal Simplification. Ebert et al. [32], 279–286.

[64] D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. *Proceedings, 1995 Symposium on Interactive 3D Graphics*, 105 – 106. 1995.

[65] P. W. C. Maciel and P. Shirley. Visual Navigation of Large Environments Using Textured Clusters. *Symposium on Interactive 3D Graphics*, 95–102, 211. 1995. URL citeseer.nj.nec.com/cardosomaciel95visual.html.

[66] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimitis, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very High Resolution Simulation of Compressible Turbulenceon the IBM-SP System. *Proceedings of Supercomputing 99*. Nov. 1999.

[67] V. Pascucci and R. Frank. Global Static Indexing for Real-time Exploration of Very Large Regular Grids. *Proc. SC 2001, High Performance Networking and Computing*. 2001.

[68] M. Pharr, C. Kolb, R. Gershbein, and P. M. Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Proceedings of SIGGRAPH 97*, 101–108. August 1997.

[69] C. Prince. *Progressive Meshes for Large Models of Arbitrary Topology*. Master's thesis, University of Washington, 2000.

[70] J. Rossignac and P. Borrel. Multi-Resolution 3D Approximations for Rendering. *Modeling in Computer Graphics*, 455–465. June–July 1993.

[71] J. Rossignac and P. Borrel. Multi-Resolution 3D Approximations for Rendering Complex Scenes. *Modeling in Computer Graphics*, 455–465. Springer-Verlag, 1993.

[72] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. *Proceedings of SIGGRAPH 2000*, 343–352. 2000.

[73] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of Triangle Meshes. *Computer Graphics (Proceedings of SIGGRAPH 92)*, vol. 26, 65–70. Jul. 1992.

[74] E. Shaffer and M. Garland. Efficient Adaptive Simplification of Massive Meshes. *IEEE Visualization 2001*, 127–134. Oct. 2001.

[75] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree. *IEEE Visualization 99*, 371–378, Oct. 1999.

[76] F. Sillion, G. Drettakis, and B. Bodelet. Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery. *Computer Graphics Forum*, 16(3):C207–C218, 1997.

[77] P. Sulatycke and K. Ghose. A fast multithreaded out-of-core visualization technique. *Proc. 13th International Parallel Processing Symposium*, 569–575. 1999.

[78] P. M. Sutton and C. D. Hansen. Accelerated Isosurface Extraction in Time-Varying Fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):98–107, Apr. 2000.

[79] S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan. Partitioning and Ordering Large Radiosity Computations. *Proceedings of SIGGRAPH 94*, 443–450. July 1994.

[80] S. Teller and C. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics*, 25(4):61–68, 1991.

[81] G. Turk. Re-tiling polygonal surfaces. *Computer Graphics: Proceedings SIGGRAPH '92*, vol. 26, No. 2, 55–64. 1992.

[82] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-Core Streamline Visualization on Large Unstructured Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, Oct. 1997.

[83] G. Varadhan and D. Manocha. Out-of-Core Rendering of Massive Geometric Environments. *IEEE Visualization 2002*. 2002. To appear.

[84] J. S. Vitter. External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, 2001.

[85] I. Wald, P. Slusallek, and C. Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. *Rendering Techniques 2001*, 277–288, 2001.

[86] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.

[87] J. Wilhelms and A. V. Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[88] P. Wonka, M. Wimmer, and F. Sillion. Instant Visibility. *Computer Graphics Forum*, 20(3):411–421, 2001.

[89] J. Xia, J. El-Sana, and A. Varshney. Adaptive Real-Time Level-of-detail-based Rendering for Polygonal Models. *IEEE Transactions on Visualization and Computer Graphics*, 171 – 183, June 1997.