SIGGRAPH 2001
Course 41: Advanced Issues in Level of Detail

Presenters:

David Luebke, University of Virginia

Amitabh Varshney, University of Maryland

Jon Cohen, Johns Hopkins University

Martin Reddy, SRI International

Ben Watson, Northwestern University

Organizer:

David Luebke

Slides:

Luebke (55 pages)	1
Varshney (53)	56
Cohen (22)	109
Reddy (28)	131
Watson (29)	159

Supplemental Material:

Luebke: *"Perceptually Driven Interactive Rendering"* (10)	188
Luebke: "*Robust View-Dependent Simplification …*" (35)	198
Luebke: *"A Developer's Survey of Polygonal Simplification…"* (12)	233
Varshney: *"Skip Strips…"* (8)	245
Varshney: *"Adaptive Real-Time Level-of-Detail …"* (23)	253
Varshney: *"Variable-Precision Rendering"* (11)	276
Cohen: *"Concepts and Algorithms for Polygonal Simplification"* (34)	287
Cohen: *"Appearance-Preserving Simplification"* (8)	321
Cohen: *"Simplifying Polygonal Models Using Successive Mappings"* (9)	329
Cohen: *"Simplification Envelopes"* (11)	338

Advanced Issues in Level of Detail

David Luebke	University of Virginia
Jon Cohen	Johns Hopkins University
Amitabh Varshney	University of Maryland
Martin Reddy	SRI International
Ben Watson	Northwestern University

SIGGRAPH 2001

Course Introduction

- *Level of detail* or *LOD* methods provide a powerful means for managing scene complexity
- Now a standard tool for the graphics developer to control rendering speed
- This course will adress advanced issues in using and developing LOD algorithms

SIGGRAPH 2001

Course Prerequisites

- We assume:
  - Knowledge of the basic LOD concept
  - Experience with interactive graphics
- Target audience
  - Developers wishing to become sophisticated LOD users
  - Researchers wishing to broaden their knowledge of the field

SIGGRAPH 2001

Course Topics

- Underpinnings (*Cohen*, *Reddy*, *Watson*):
  - Classifying LOD error metrics
  - Perceptual analysis of degradation of detail
  - Balancing fidelity and performance

SIGGRAPH 2001

Course Topics

- Topics in LOD creation (*Cohen*, *Varshney*):
  - Appearance-preserving simplification
  - Controlled topology reduction
  - Variable-precision rendering

SIGGRAPH 2001

Course Topics

- LOD frameworks (*Luebke*, *Varshney*):
  - Static LOD
  - Dynamic LOD
  - View-dependent LOD

SIGGRAPH 2001

Course Schedule

8:30	Welcome, Introductions	Luebke
8:50	Frameworks	Luebke
	Static versus view-dependent LOD	
10	Break	
10:15	Algorithms	Varshney
	Algorithms and approaches for simplification	
11:30	Fidelity	Cohen
	Measuring error and preserving appearance	
12	Lunch	

SIGGRAPH 2001

Course Schedule

1:30	Fidelity Continued	Cohen
2	Perception	Reddy
	Understanding and applying visual perception	
3	Break	
3:15	Tradeoffs	Watson
	Balancing fidelity and performance	
4:30	Panel	All
	Issues, trends and important questions in LOD	
4:55	Conclusion	Luebke

SIGGRAPH 2001

Frameworks: Static Vs. View-Dependent LOD

- Questions I will address:
  - *What is view-dependent simplification?*
  - *Why is it better than traditional simplification? When is it worse?*
  - *How is it implemented efficiently?*
- Questions I will leave for the others:
  - *How can we control visual fidelity?*
  - *How much simplification is appropriate?*

SIGGRAPH 2001

Motivation: Preaching To The Choir

- Interactive rendering of large-scale geometric datasets is important
  - Scientific and medical visualization
  - Architectural and industrial CAD
  - Training (military and otherwise)
  - Entertainment

SIGGRAPH 2001

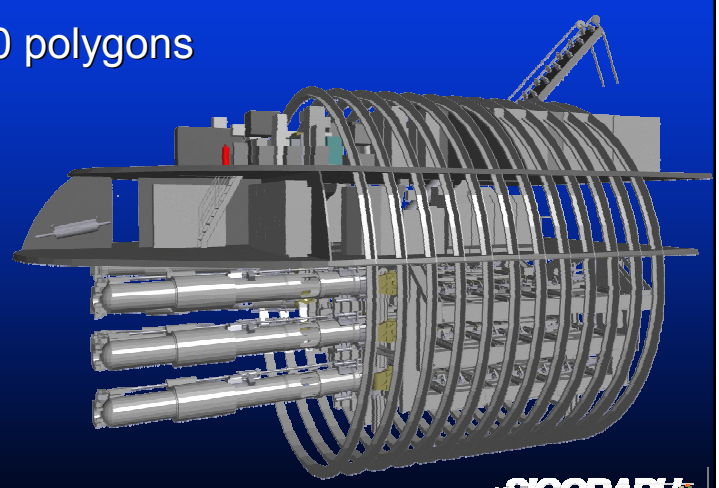
Motivation:
Big Models

- The problem:
  - Polygonal models are often too complex to render at interactive rates
- Even worse:
  - Incredibly, models are getting bigger as fast as hardware is getting faster…

SIGGRAPH 2001

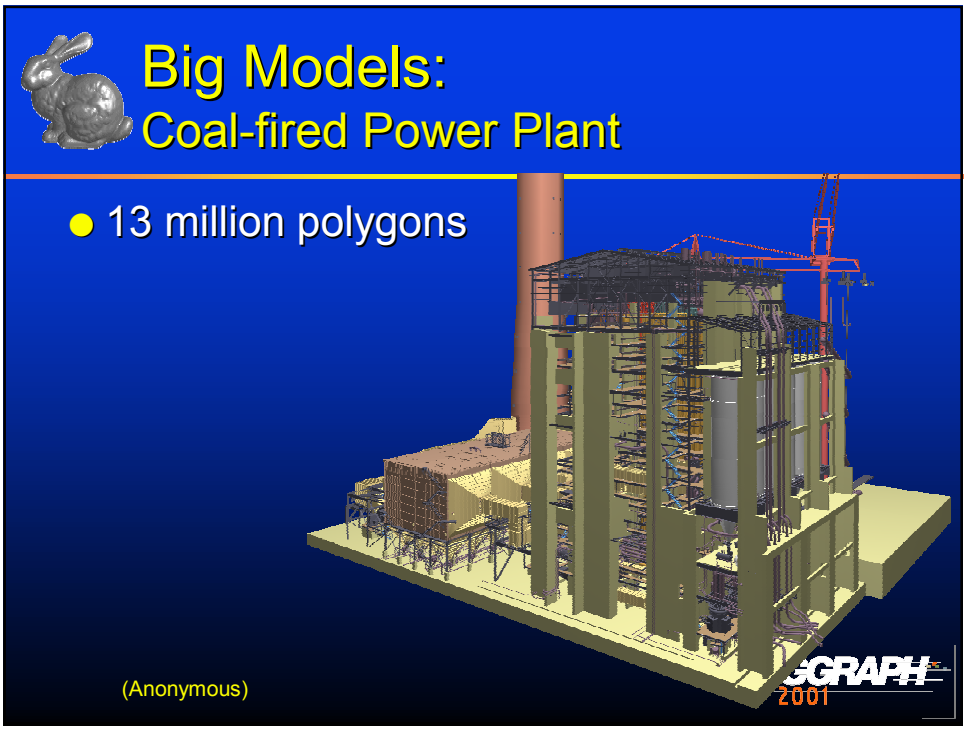
Big Models:
Submarine Torpedo Room

- 700,000 polygons



Courtesy General Dynamics, Electric Boat Div.

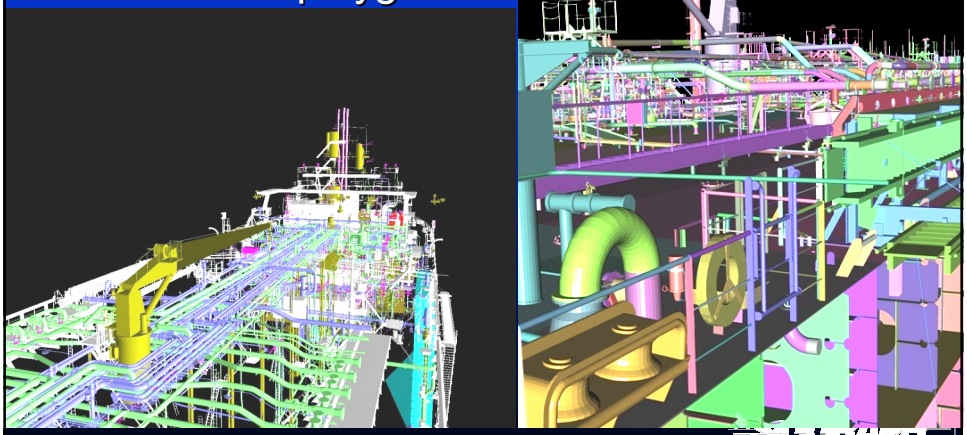
SIGGRAPH 2001


Big Models:
Coal-fired Power Plant
13 million polygons
(Anonymous)
SIGGRAPH
2001

Big Models:
Plant Ecosystem Simulation
16.7 million polygons (sort of)
Deussen et al: Realistic Modeling of Plant Ecosystems

Big Models:
Double Eagle Container Ship

- 82 million polygons



Courtesy Newport News Shipbuilding

Big Models:
The Digital Michelangelo Project

- David:
  56,230,343 polygons
- St. Matthew:
  372,422,615 polygons

Courtesy Digital Michelangelo Project

Level of Detail:
The Basic Idea

- One solution:
  - Simplify the polygonal geometry of small or distant objects
  - Known as *Level of Detail* or *LOD*
    - A.k.a. polygonal simplification, geometric simplification, mesh reduction, multiresolution modeling, …

SIGGRAPH 2001

Level of Detail:
Traditional Approach

Create *levels of detail* (LODs) of objects:

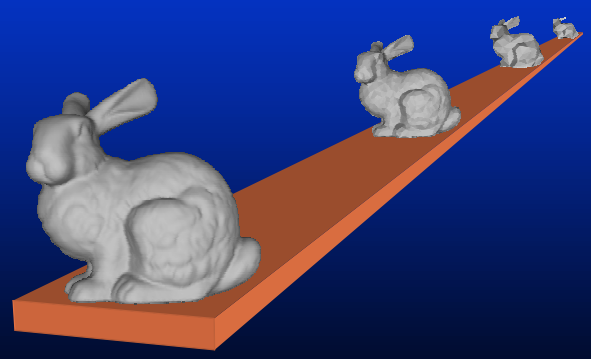
69,451 polys | 2,502 polys | 251 polys | 76 polys

Courtesy Stanford 3D Scanning Repository

SIGGRAPH 2001

Level of Detail: Traditional Approach

- Distant objects use coarser LODs:



SIGGRAPH 2001

Traditional Approach: Static Level of Detail

- Traditional LOD in a nutshell:
  - Create LODs for each object separately in a preprocess
  - At run-time, pick each object's LOD according to the object's distance (or similar criterion)
- Since LODs are created offline at fixed resolutions, I refer to this as *Static LOD*

SIGGRAPH 2001

Advantages of Static LOD

- Simplest programming model; decouples simplification and rendering
  - LOD creation need not address real-time rendering constraints
  - Run-time rendering need only pick LODs

SIGGRAPH 2001

Advantages of Static LOD

- Fits modern graphics hardware well
  - Easy to compile each LOD into triangle strips, display lists, vertex arrays, …
  - These render *much* faster than unorganized polygons on today's hardware (3-5 x)

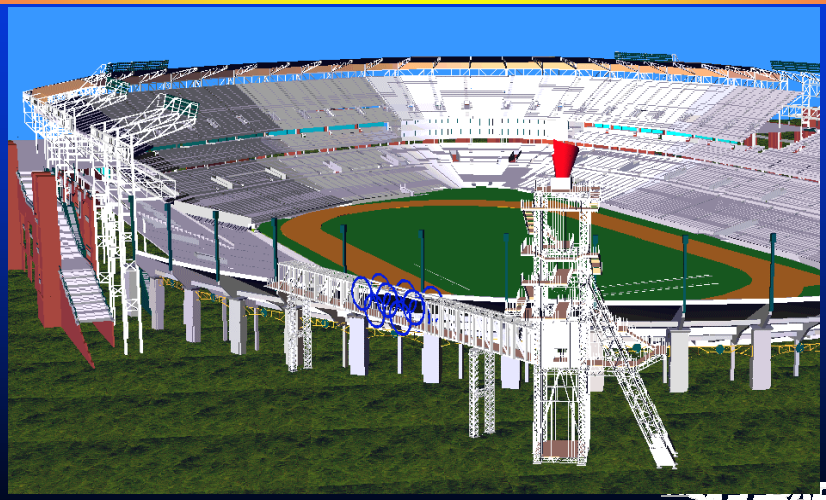
SIGGRAPH 2001

Disadvantages of Static LOD

- So why use anything but static LOD?
- Answer: sometimes static LOD not suited for *drastic simplification*
- Some problem cases:
  - Terrain flyovers
  - Volumetric isosurfaces
  - Super-detailed range scans
  - Massive CAD models

SIGGRAPH 2001

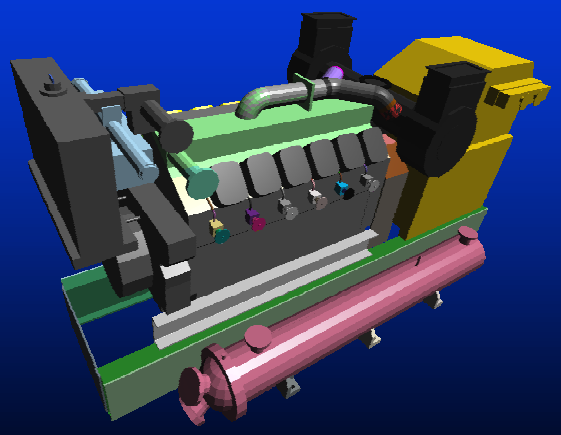
Drastic Simplification:
The Problem With Large Objects



Courtesy IBM and ACOG

SIGGRAPH 2001

Drastic Simplification:
The Problem With Small Objects



Courtesy Electric Boat

SIGGRAPH 2001

Drastic Simplification

- For drastic simplification:
  - Large objects must be subdivided
  - Small objects must be combined
- Difficult or impossible with static LOD
- *So what can we do?*

SIGGRAPH 2001

Dynamic Level of Detail

- A departure from the traditional static approach:
  - Static LOD: create individual LODs in a preprocess
  - Dynamic LOD: create data structure from which a desired level of detail can be extracted *at run time*.

SIGGRAPH 2001

Dynamic LOD: Advantages

- Better granularity → better fidelity
  - LOD is specified exactly, not chosen from a few pre-created options
  - Thus objects use no more polygons than necessary, which frees up polygons for other objects
  - Net result: better resource utilization, leading to better overall fidelity/polygon

SIGGRAPH 2001

Dynamic LOD: Advantages

- Better granularity → smoother transitions
  - Switching between traditional LODs can introduce visual "popping" effect
  - Dynamic LOD can adjust detail gradually and incrementally, reducing visual pops
    - Can even *geomorph* the fine-grained simplification operations over several frames to eliminate pops [Hoppe 96, 98]

SIGGRAPH 2001

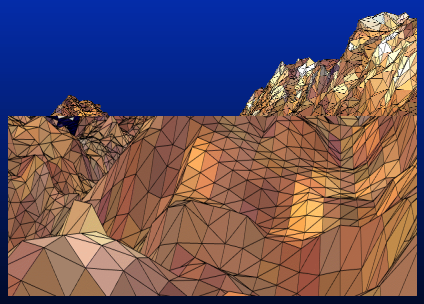
Dynamic LOD: Advantages

- Supports progressive transmission
  - *Progressive Meshes [Hoppe 97]*
  - *Progressive Forest Split Compression [Taubin 98]*
- Supports *view-dependent LOD*
  - Use current view parameters to select best representation *for the current view*
  - Single objects may thus span several levels of detail

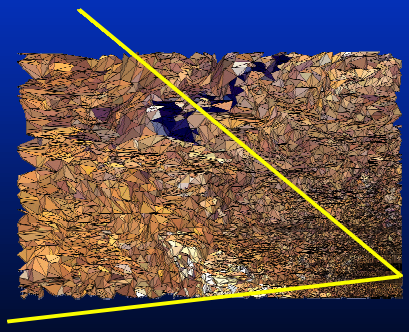
SIGGRAPH 2001

View-Dependent LOD:
Examples

- Show nearby portions of object at higher resolution than distant portions



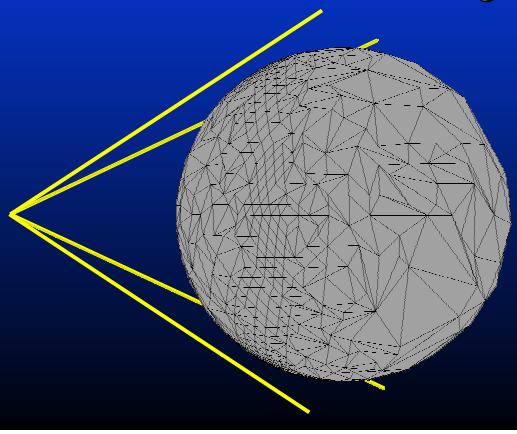
View from eyepoint



Birds-eye view

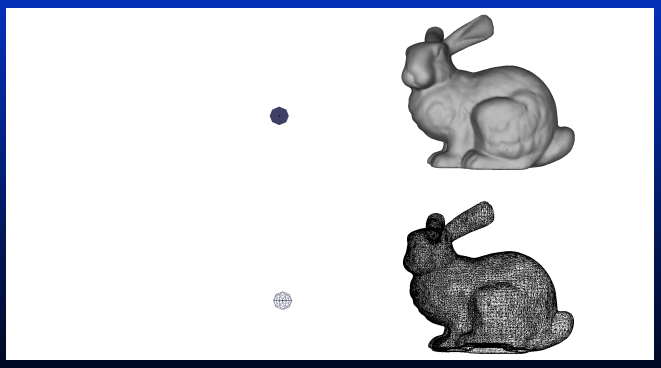
View-Dependent LOD:
Examples

- Show silhouette regions of object at higher resolution than interior regions



View-Dependent LOD: Examples

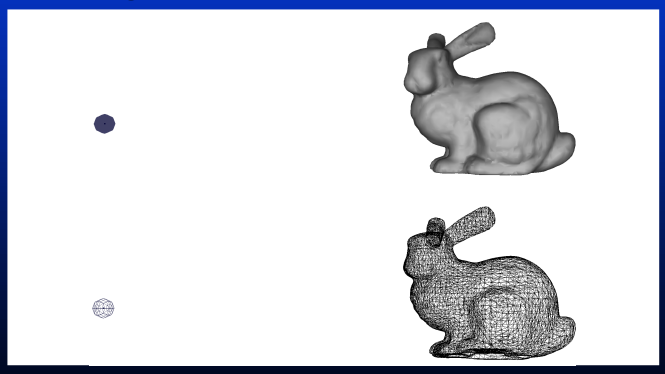
- Show more detail where the user is looking than in their peripheral vision:



34,321 triangles

View-Dependent LOD: Examples

- Show more detail where the user is looking than in their peripheral vision:



11,726 triangles

View-Dependent LOD: Advantages

- Even better granularity
  - Allocates polygons where they are most needed, within as well as among objects
  - Enables even better overall fidelity
- Enables drastic simplification of very large objects
  - Example: stadium model
  - Example: terrain flyover

SIGGRAPH 2001

An Aside: Hierarchical LOD

- View-dependent LOD solves the Problem With Large Objects
- *Hierarchical LOD* can solve the Problem With Small Objects
  - Merge objects into assemblies
  - At sufficient distances, simplify assemblies, not individual objects
- Note that hierarchical LOD implies a topology-modifying algorithm

SIGGRAPH 2001

An Aside: Hierarchical LOD

- Hierarchical LOD dovetails nicely with view-dependent LOD
  - Treat the *entire scene* as a single object to be simplified in view-dependent fashion
- Hierarchical LOD can also sit atop traditional static LOD schemes
  - *Imposters [Maciel 95]*
  - *HLODs [Erikson 01]*

View-Dependent LOD: Algorithms

- Many good published algorithms:
  - *Progressive Meshes* by Hoppe [SIGGRAPH 96, SIGGRAPH 97, …]
  - *Merge Trees* by Xia & Varshney [Visualization 96]
  - *Hierarchical Dynamic Simplification* by Luebke & Erikson [SIGGRAPH 97]
  - *Multitriangulation* by DeFloriani et al
  - Others…

Overview:
The VDS Algorithm

- I'll mostly describe my own work
  - Algorithm: *VDS* Implementation: *VDSlib*
  - Similar in concept to most other algorithms

SIGGRAPH 2001

Overview:
The VDS Algorithm

- Overview of the VDS algorithm:
  - A preprocess builds the *vertex tree*, a hierarchical clustering of vertices
  - At run time, clusters appear to grow and shrink as the viewpoint moves
  - Clusters that become too small are collapsed, filtering out some triangles

SIGGRAPH 2001

Data Structures

- The *vertex tree*
  - Represents the entire model
  - Hierarchy of all vertices in model
  - Queried each frame for updated scene
- The *active triangle list*
  - Represents the current simplification
  - List of triangles to be displayed
  - Triangles added and deleted by operations on vertex tree

SIGGRAPH 2001

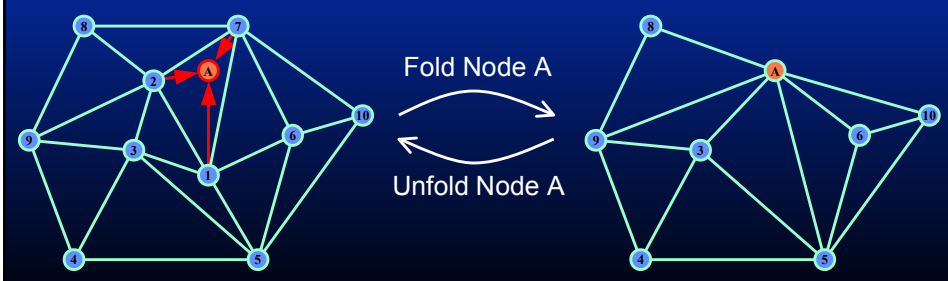
The Vertex Tree

- Each vertex tree node *supports* a subset of the model vertices
  - Leaf nodes support a single vertex from the original full-resoluton model
  - The root node supports all vertices
- For each node we also assign a representative vertex or *proxy*

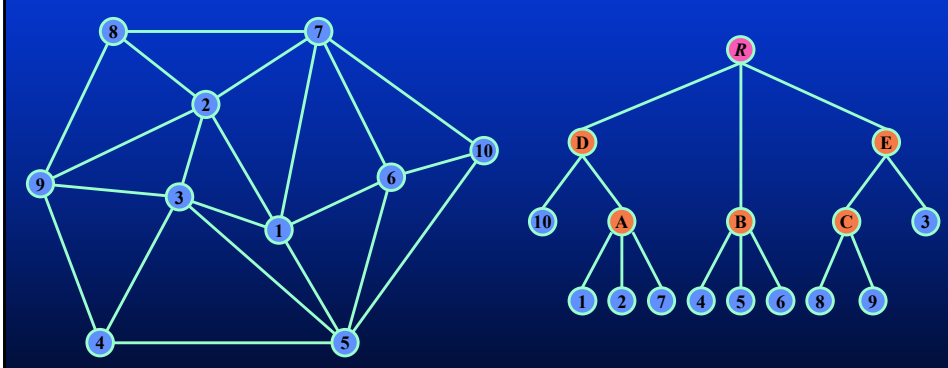
SIGGRAPH 2001

The Vertex Tree:
Folding And Unfolding

- *Folding* a node collapses its vertices to the proxy
- *Unfolding* the node splits the proxy back into vertices

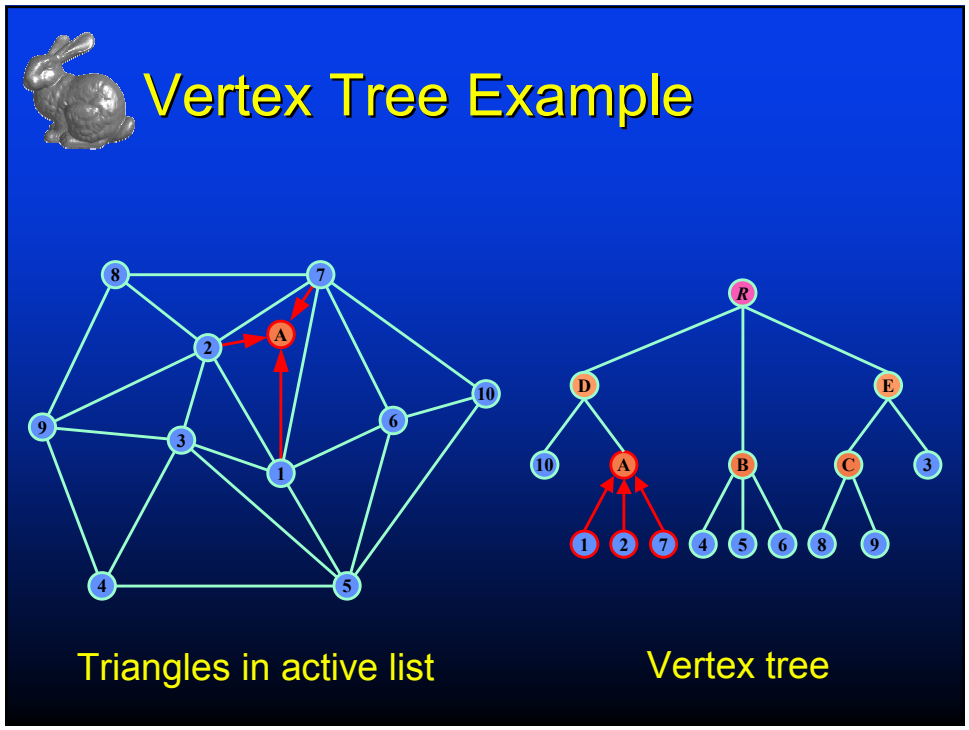


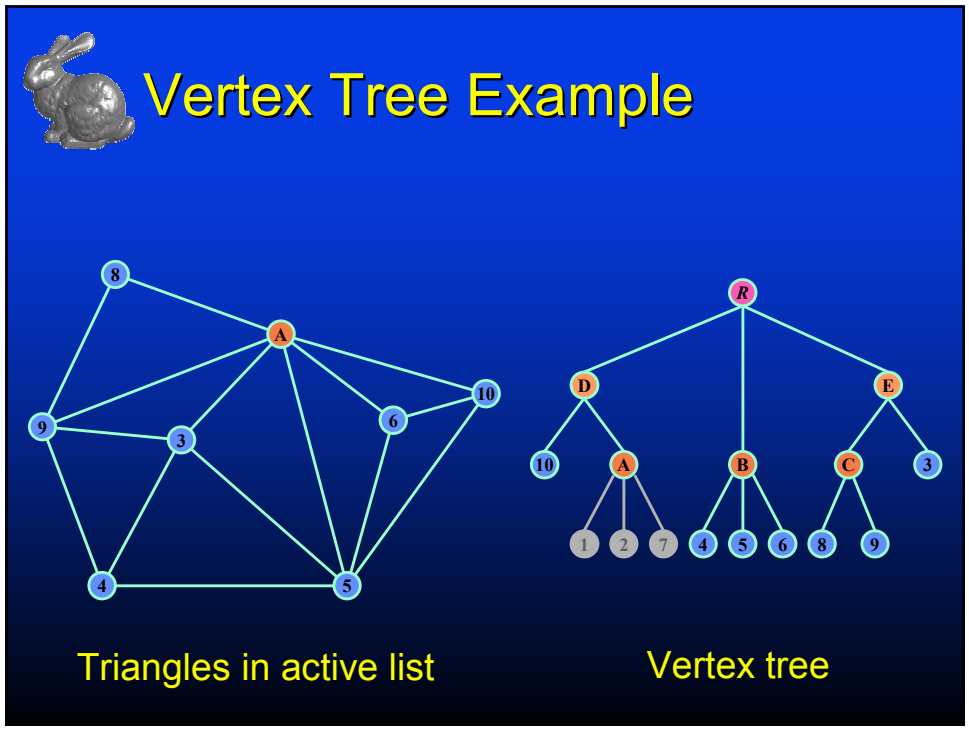
Vertex Tree Example

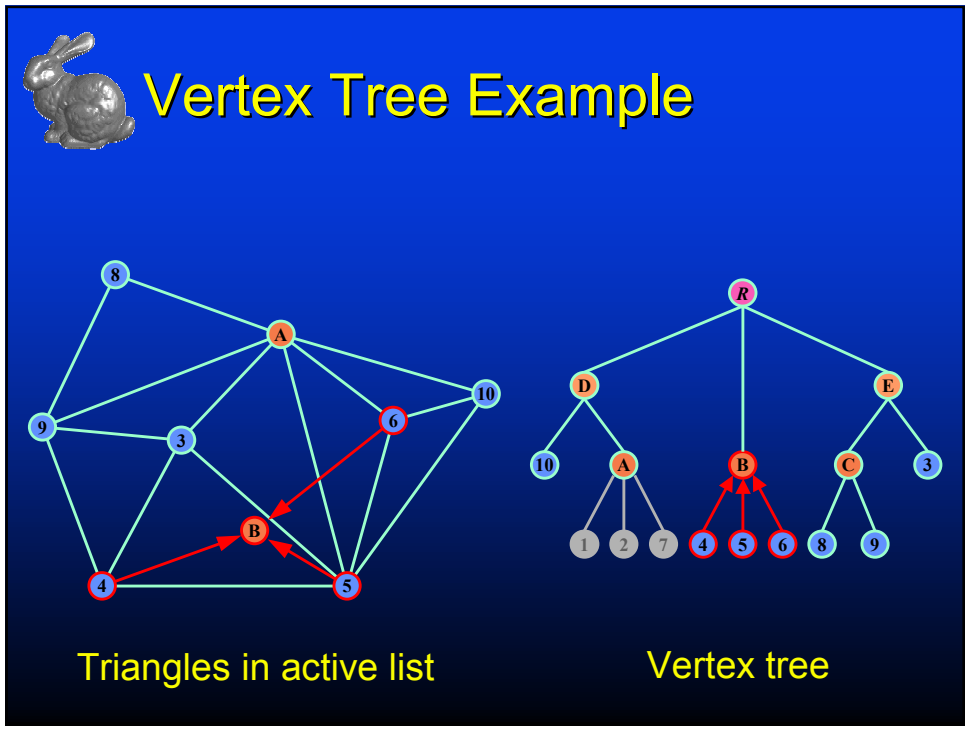


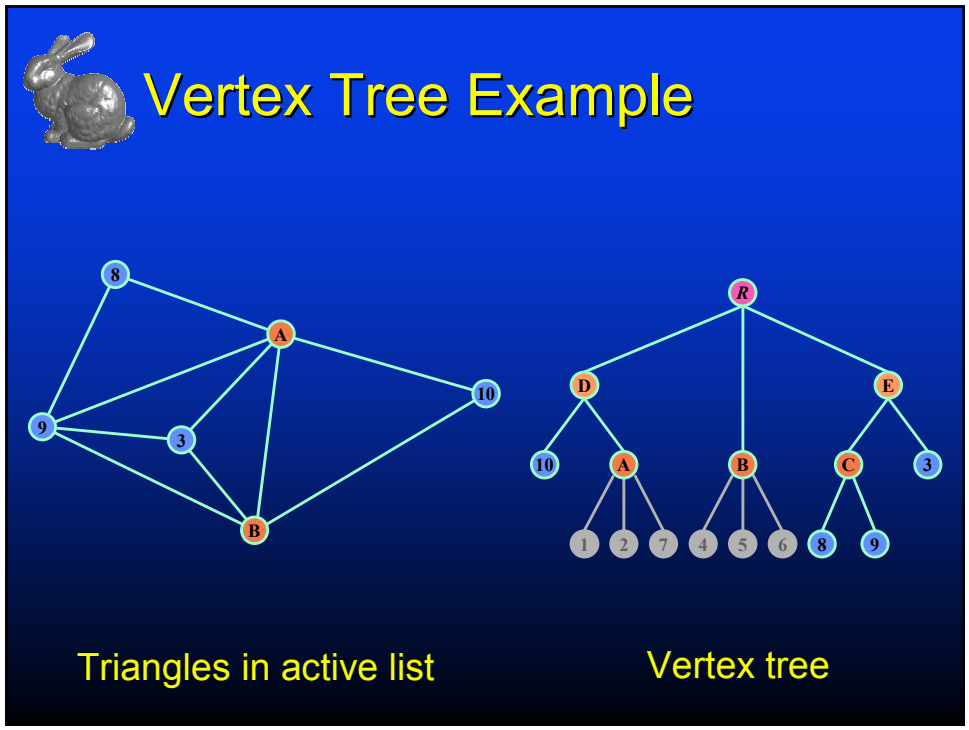
Triangles in active list

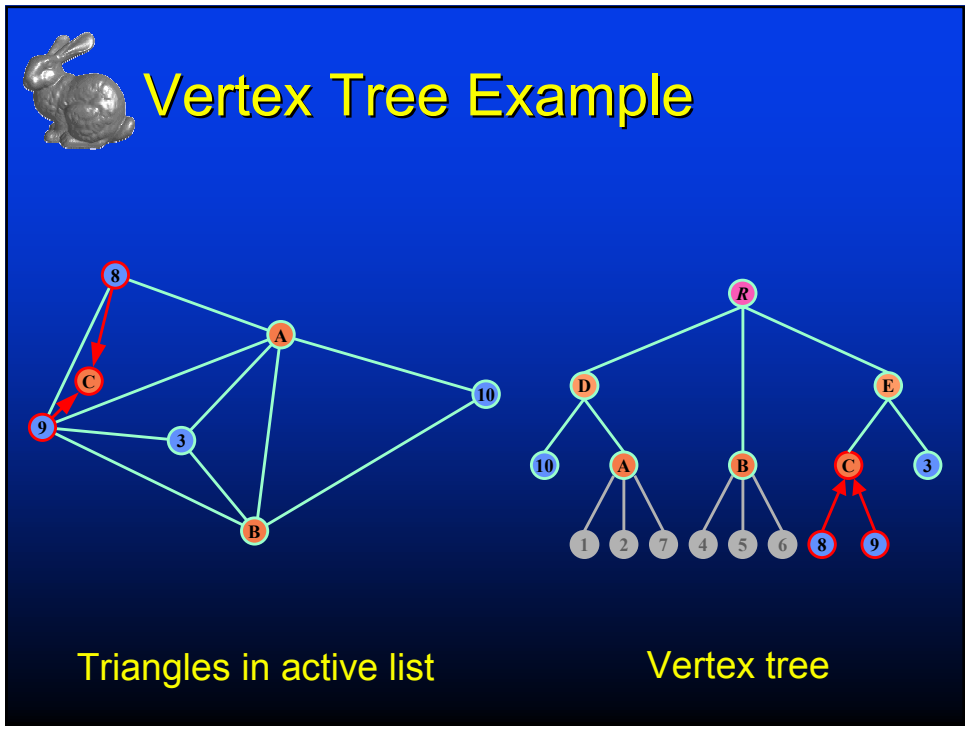
Vertex tree

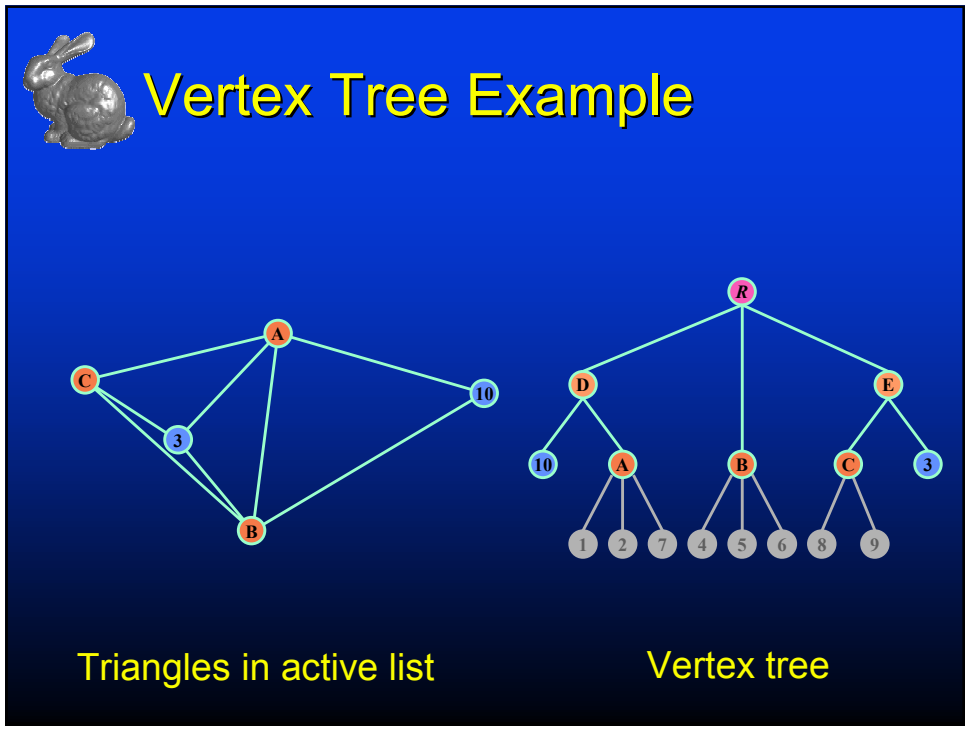

Vertex Tree Example
8
7
2
A
10
9
3
6
1
4
5
R
D
E
10
A
B
C
3
1
2
7
4
5
6
8
9
Triangles in active list
Vertex tree

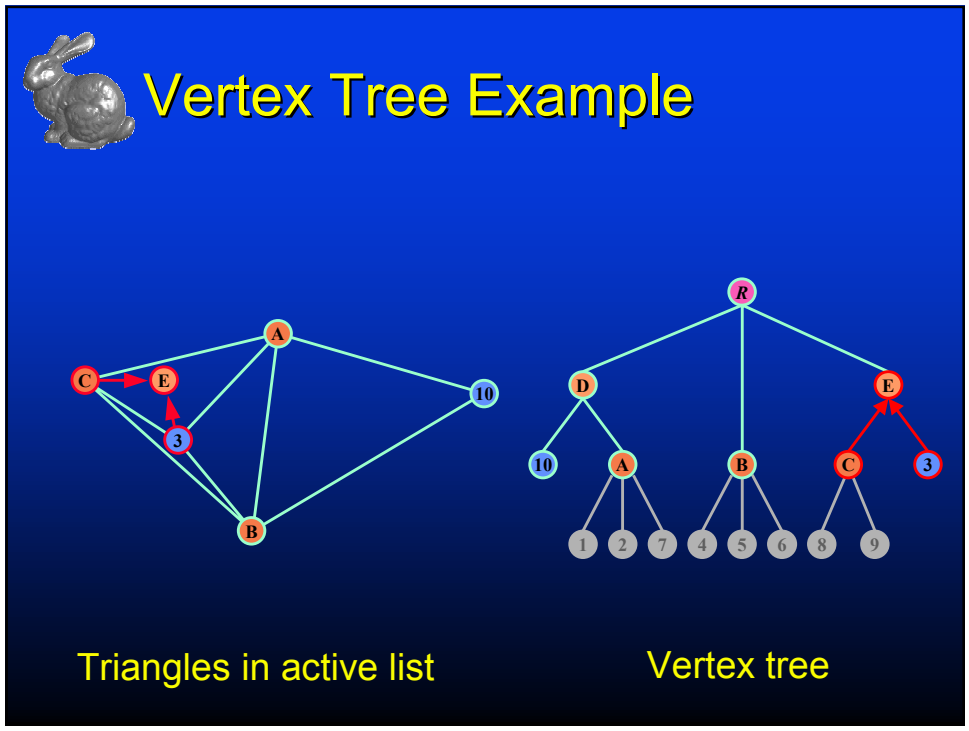

Vertex Tree Example
8
A
10
9
3
6
4
5
R
D
E
10
A
B
C
3
1
2
7
4
5
6
8
9
Triangles in active list
Vertex tree

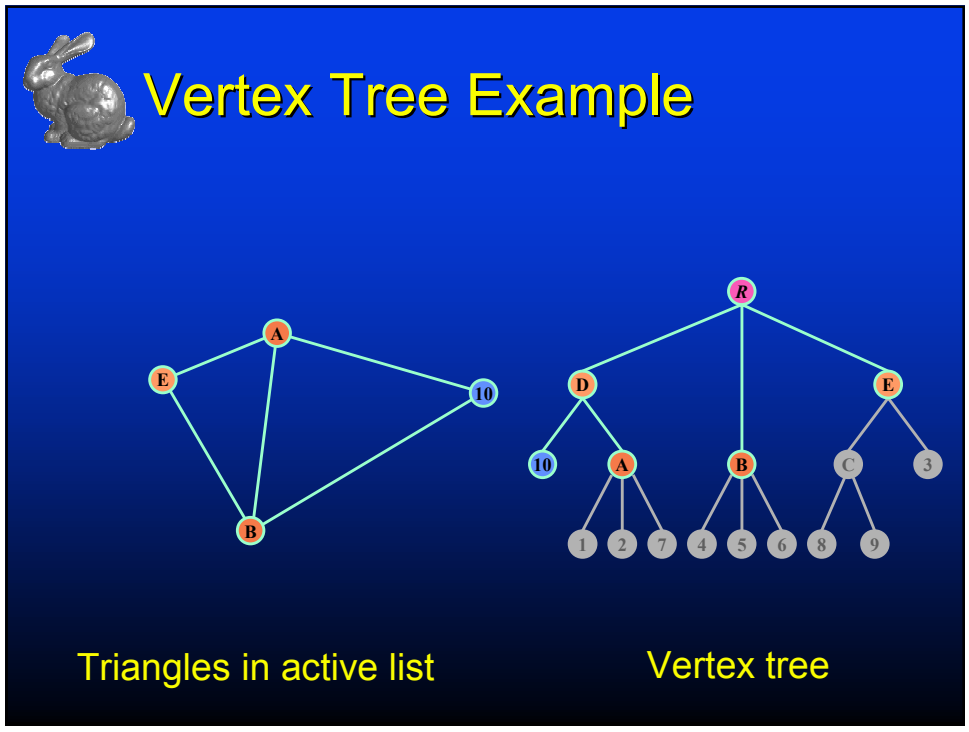

Vertex Tree Example
8
A
10
9
3
6
B
4
5
R
D
E
10
A
B
C
3
1
2
7
4
5
6
8
9
Triangles in active list
Vertex tree


Vertex Tree Example
8
A
10
9
3
B
R
D
E
10
A
B
C
3
1
2
7
4
5
6
8
9
Triangles in active list
Vertex tree

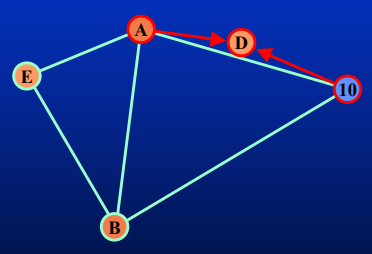

Vertex Tree Example
8
A
C
10
9
3
B
R
D
E
10
A
B
C
3
1
2
7
4
5
6
8
9
Triangles in active list
Vertex tree

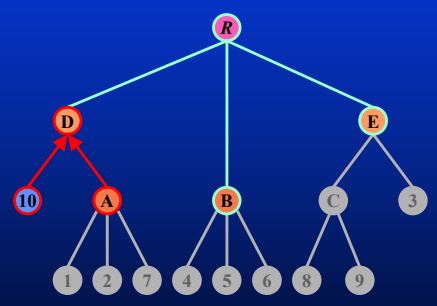

Vertex Tree Example
A
C
10
3
B
R
D
E
10
A
B
C
3
1
2
7
4
5
6
8
9
Triangles in active list
Vertex tree


Vertex Tree Example
A
C
E
10
3
B
R
D
E
10
A
B
C
3
1
2
7
4
5
6
8
9
Triangles in active list
Vertex tree


Vertex Tree Example
A
E
10
B
R
D
E
10
A
B
C
3
1
2
7
4
5
6
8
9
Triangles in active list
Vertex tree

Vertex Tree Example

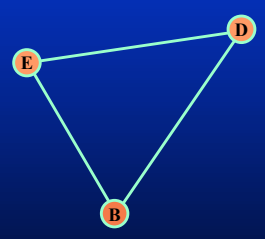


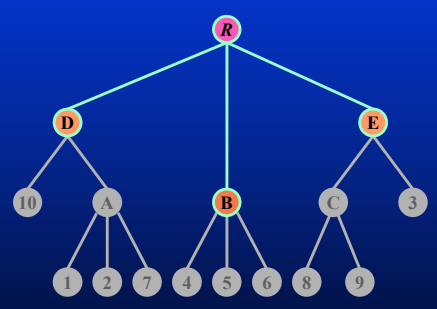


Triangles in active list

Vertex tree

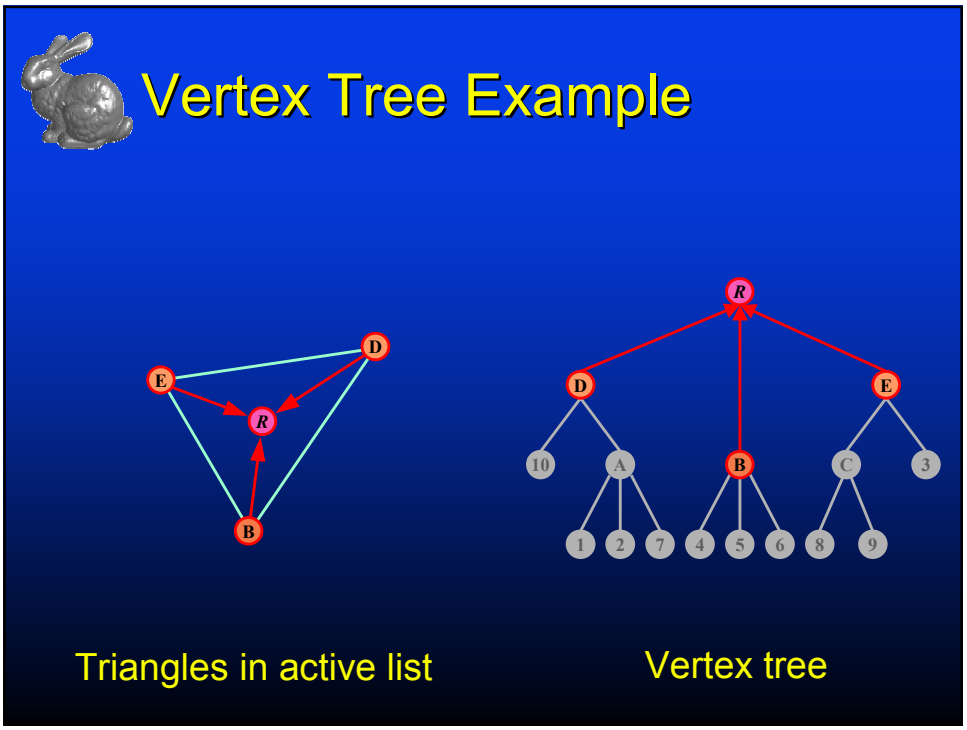
Vertex Tree Example

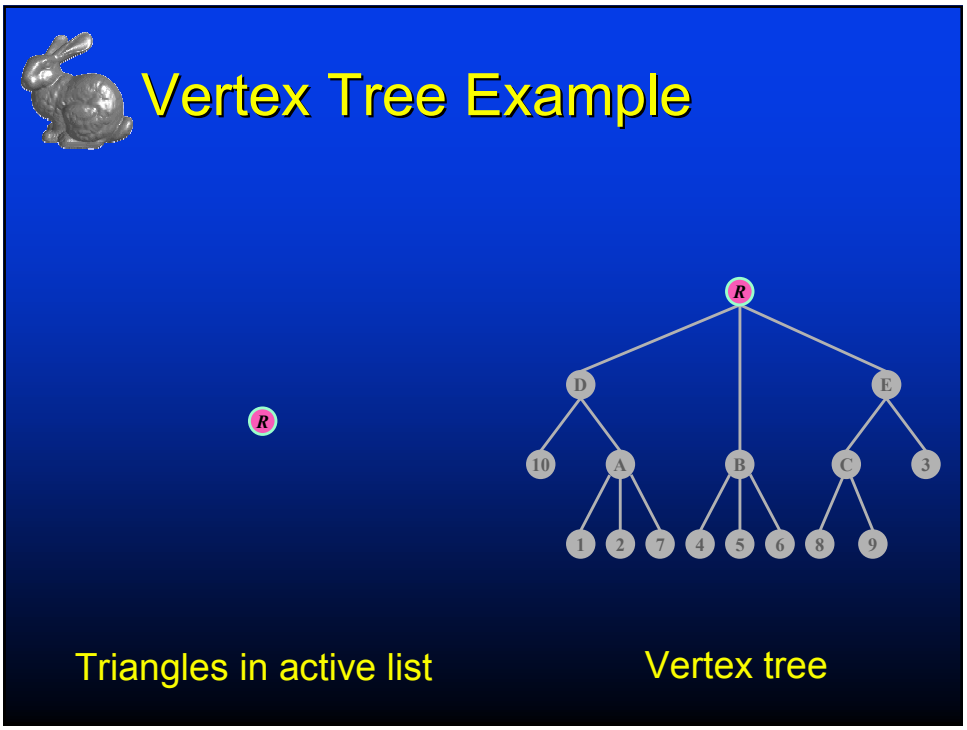




Triangles in active list

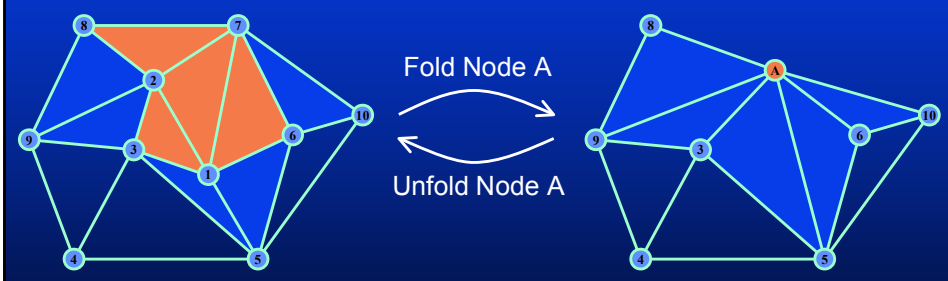
Vertex tree


Vertex Tree Example
E
D
R
B
R
D
E
10
A
B
C
3
1
2
7
4
5
6
8
9
Triangles in active list
Vertex tree


Vertex Tree Example
R
R
D
E
10
A
B
C
3
1
2
7
4
5
6
8
9
Triangles in active list
Vertex tree

The Vertex Tree:
Livetris and Subtris

- Two categories of triangles affected:



`Node->Subtris`: triangles that disappear upon folding
`Node->Livetris`: triangles that just change shape

The Vertex Tree:
Livetris and Subtris

- The *key observation*:
  - Each node's subtris can be computed offline to be accessed quickly at run time
  - Each node's livetris can be maintained at run time, or lazily evaluated upon rendering

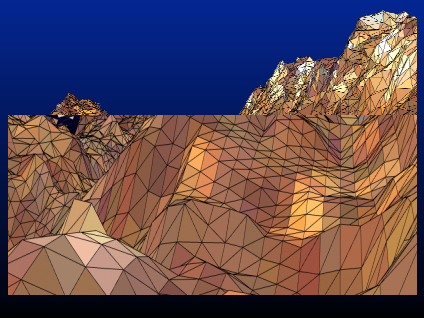
View-Dependent Simplification

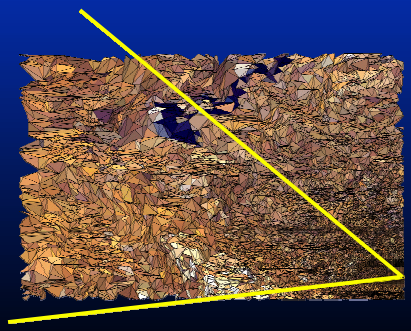
- Any run-time criterion for folding and unfolding nodes may be used
- Examples of view-dependent simplification criteria:
  - Screenspace error threshold
  - Silhouette preservation
  - Triangle budget simplification
  - Gaze-directed perceptual simplification (discussed by Martin later)

SIGGRAPH 2001

Screenspace Error Threshold

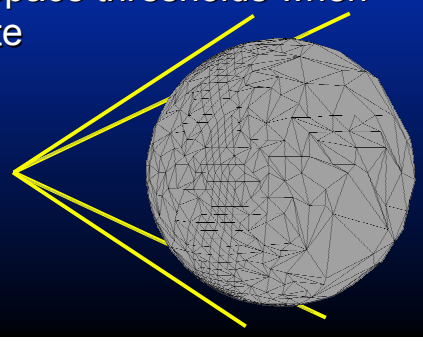
- Nodes chosen by projected area
  - User sets screenspace size threshold
  - Nodes which grow larger than threshold are unfolded





Silhouette Preservation

- Retain more detail near silhouettes
  - A *silhouette node* supports triangles on the visual contour
  - Use tighter screenspace thresholds when examining silhouette nodes



Triangle Budget Simplification

- Minimize error within specified number of triangles
  - Sort nodes by screenspace error
  - Unfold node with greatest error, putting children into sorted list
  - Repeat until budget is reached

SIGGRAPH 2001

View-Dependent Criteria:
Other Possibilities

- *Specular highlights*: Xia describes a fast test to unfold likely nodes
- *Surface deviation*: Hoppe uses an elegant surface deviation metric that combines silhouette preservation and screenspace error threshold

SIGGRAPH 2001

View-Dependent Criteria:
Other Possibilities

- *Sophisticated surface deviation metrics:* See Jon's talk!
- *Sophisticated perceptual criteria*: See Martin's talk!
- *Sophisticated temporal criteria*: See Ben's talk!

SIGGRAPH 2001

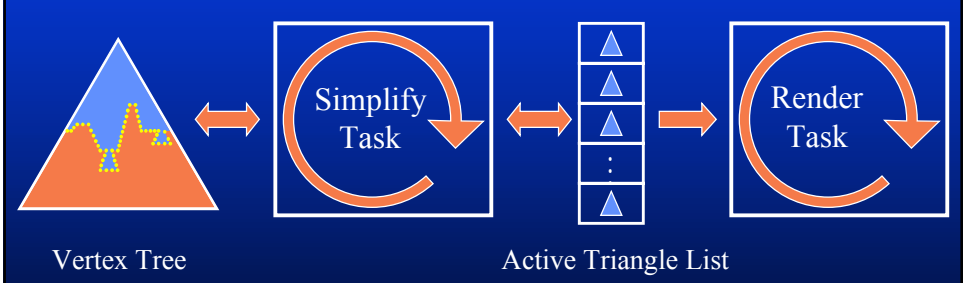
Implementing VDS: Optimizations

- Asynchronous simplification
  - Parallelize the algorithm
- Exploiting temporal coherence
  - Scene changes slowly over time
- Maintain memory coherent geometry
  - Optimize for rendering
  - Support for out-of-core rendering

SIGGRAPH 2001

Asynchronous Simplification

- Algorithm partitions into two tasks:



- Run them in separate processes

SIGGRAPH 2001

Asynchronous Simplification

- If S = time to simplify, R = time to render:
  - Single process = $(S + R)$
  - Pipelined = **max**(S, R)
  - Asynchronous = R

SIGGRAPH 2001

Temporal Coherence

- Exploit the fact that frame-to-frame changes are small
- Two examples:
  - Active triangle list
  - Vertex tree

SIGGRAPH 2001

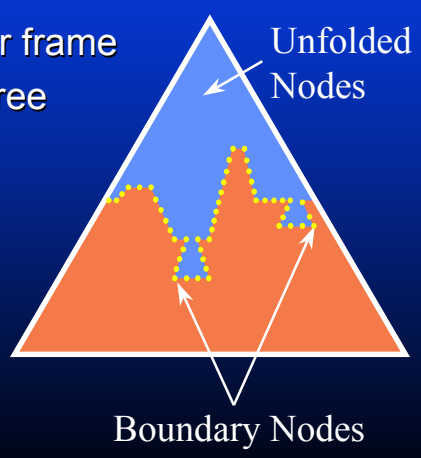
Exploiting Temporal Coherence

- Active triangle list
  - Could calculate active triangles every frame
  - But…few triangles are added or deleted each frame
  - Idea: make only incremental changes to an *active triangle list*
    - Simple approach: doubly-linked list of triangles
    - Better: maintain coherent arrays with swapping

SIGGRAPH 2001

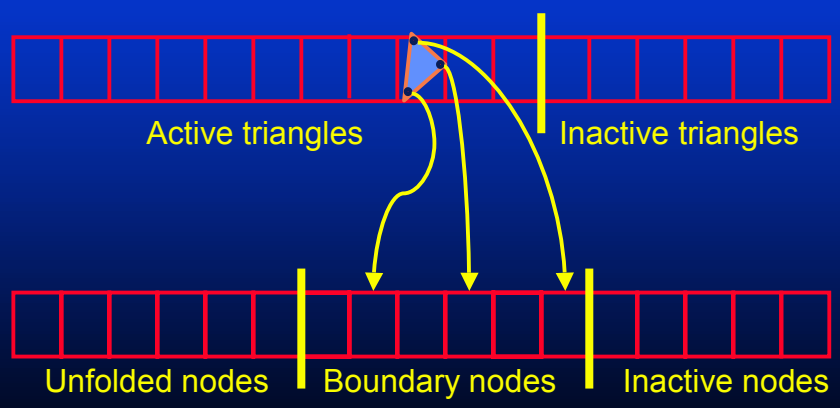
Exploiting Temporal Coherence

- Vertex Tree
  - Few nodes change per frame
  - Don't traverse whole tree
  - Do local updates only at *boundary nodes*



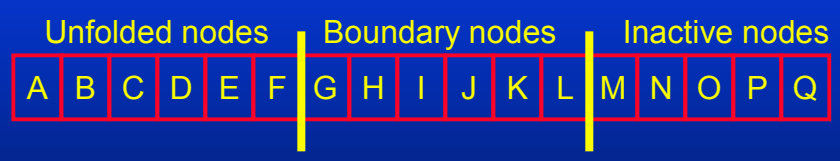
Optimizing For Rendering

- Idea: maintain geometry in coherent arrays



Optimizing For Rendering

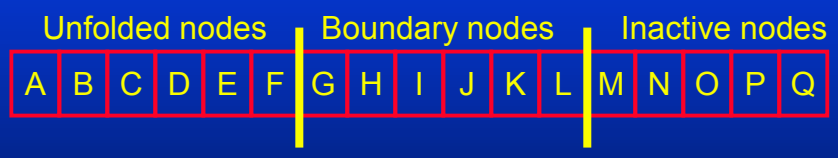
- Idea: use swaps to maintain coherence



Fold node D:

Optimizing For Rendering

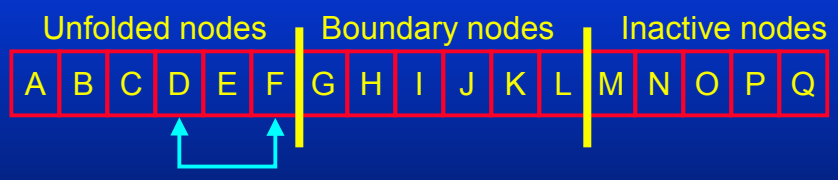
- Idea: use swaps to maintain coherence



Fold node D:
Swap D with F

Optimizing For Rendering

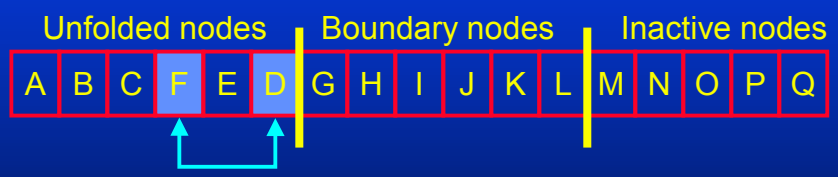
- Idea: use swaps to maintain coherence



Fold node D:
Swap D with F

Optimizing For Rendering

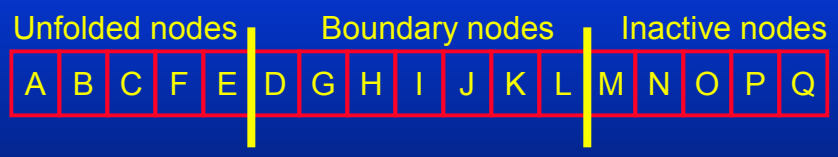
- Idea: use swaps to maintain coherence



Fold node D:
Swap D with F

Optimizing For Rendering

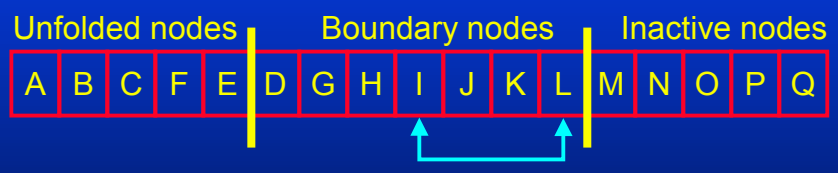
- Idea: use swaps to maintain coherence



Fold node D:
Move Unfolded/Boundary Marker

Optimizing For Rendering

- Idea: use swaps to maintain coherence

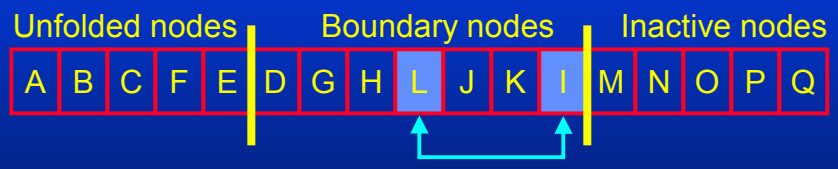


Fold node D:

Deactivate D's children (swap w/ last boundary node)

Optimizing For Rendering

- Idea: use swaps to maintain coherence

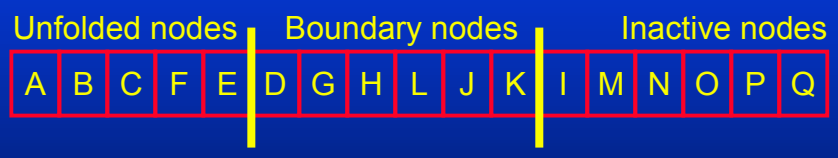


Fold node D:

Deactivate D's children (swap w/ last boundary node)

Optimizing For Rendering

- Idea: use swaps to maintain coherence

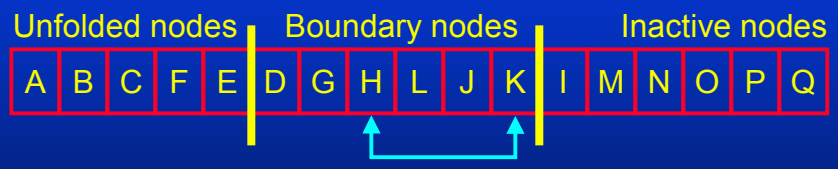


Fold node D:

Deactivate D's children (swap w/ last boundary node)

Optimizing For Rendering

- Idea: use swaps to maintain coherence

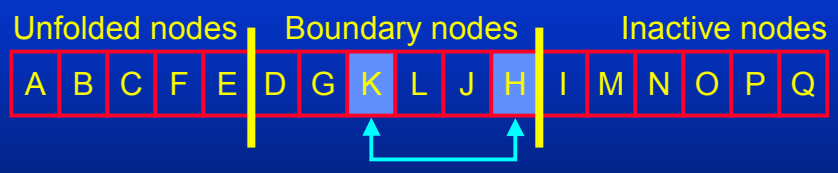


Fold node D:

Deactivate D's children (swap w/ last boundary node)

Optimizing For Rendering

- Idea: use swaps to maintain coherence

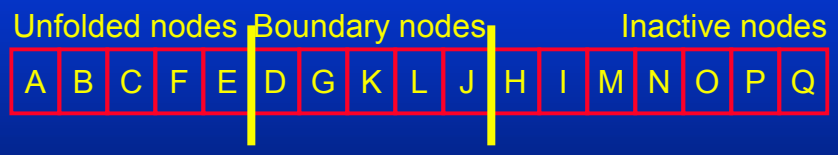


Fold node D:

Deactivate D's children (swap w/ last boundary node)

Optimizing For Rendering

- Idea: use swaps to maintain coherence

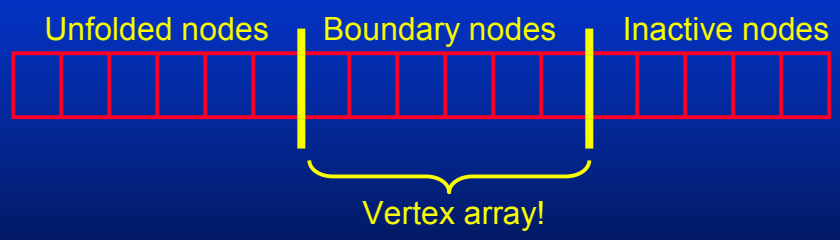


Fold node D:

Deactivate D's children (swap w/ last boundary node)

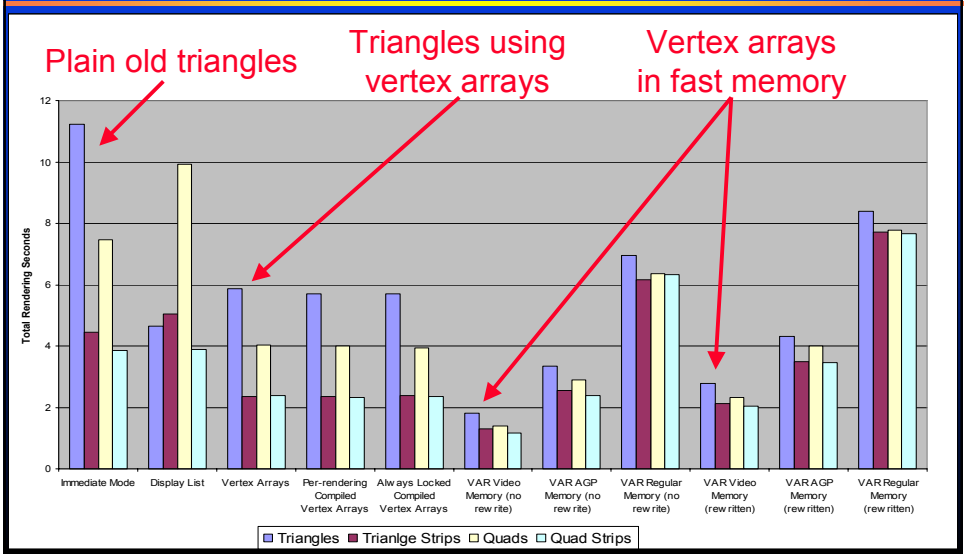
Optimizing For Rendering:
Vertex Arrays

- Biggest win: vertex arrays



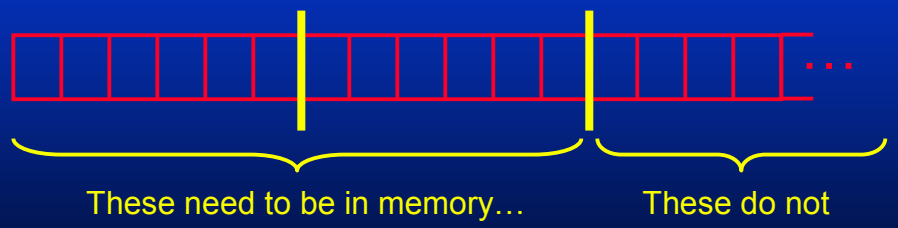
  – Actually, keep separate parallel arrays for rendering data (coords, colors, etc)

Optimizing For Rendering:
Vertex Arrays on GeForce2



Out-of-core Rendering

- Coherent arrays lend themselves to out-of-core simplification and rendering:



SIGGRAPH 2001

Out-of-core Rendering

- Coherent arrays lend themselves to out-of-core simplification and rendering:
  - Only need active portions of triangle and node arrays
  - Implement arrays as memory-mapped files
    - Let virtual memory system manage paging
    - A prefetch thread walks boundary nodes, bringing their children into memory to avoid glitches

SIGGRAPH 2001

Summary:
VDS Pros

- *Supports drastic simplification!*
  - View-dependent; handles the Problem With Large Objects
  - Hierarchical; handles the Problem With Small Objects
  - Robust; does not require (or preserve) mesh topology

SIGGRAPH 2001

Summary:
VDS Pros

- Rendering can be implemented efficiently using vertex arrays
- Supports rendering of models much larger than main memory

SIGGRAPH 2001

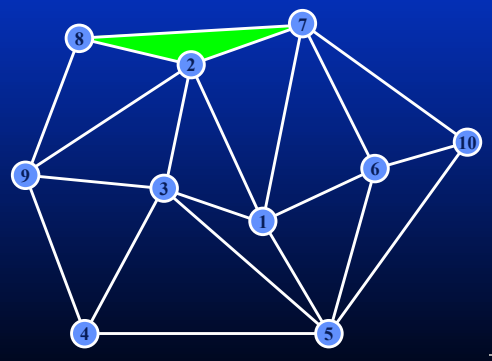
Summary:
VDS Cons

- Increases CPU load
- Fastest rendering mode currently restricted to 65K vertices (April 2001)

SIGGRAPH 2001

Summary:
VDS Cons

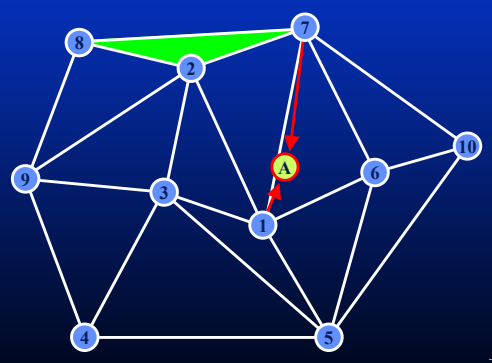
- Currently nothing to prevent *mesh foldovers:*



SIGGRAPH 2001

Summary:
VDS Cons

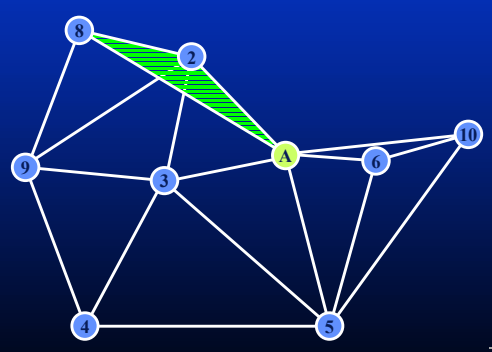
- Currently nothing to prevent *mesh foldovers:*



SIGGRAPH 2001

Summary:
VDS Cons

- Currently nothing to prevent *mesh foldovers:*



SIGGRAPH 2001

Summary:
VDS Cons

- Currently nothing to prevent *mesh foldovers*
  - These can be very distracting artifacts
  - Could be made part of the view-dependent simplification criterion, though…

SIGGRAPH 2001

View-Dependent Versus Static LOD

- View-dependent LOD is superior to traditional static LOD when:
  - Models contain very large individual objects (e.g., terrains)
  - Simplification must be completely automatic (e.g., complex CAD models)
  - Experimenting with view-dependent simplification criteria

SIGGRAPH 2001

View-Dependent Versus Static LOD

- Static LOD is often the better choice:
  - Simplest programming model
  - Reduced run-time CPU load
  - Easier to leverage hardware:
    - Compile LODs into vertex arrays/display lists
    - Stripe LODs into triangle strips
    - Utilize vertex caches and such

SIGGRAPH 2001

View-Dependent Versus Static LOD

- Applications that may want to use:
  - Static LOD
    - Video games
    - Simulators
    - Many walkthrough-style demos
  - Dynamic and view-dependent LOD
    - CAD design review tools
    - Medical & scientific visualization toolkits
    - Terrain flyovers

SIGGRAPH 2001

VDSlib

- Implementation: *VDSlib*
  - A public-domain view-dependent simplification and rendering package
  - Flexible C++ interface lets users:
    - Construct vertex trees for objects or scenes
    - Specify with callbacks how to simplify, cull, and render them
  - Available at *http://vdslib.virginia.edu*

SIGGRAPH 2001

VDSlib: Ongoing Work

- Ongoing research projects using VDSlib:
  - Out-of-core LOD for interactive rendering of *truly* massive models
  - Perceptually-guided view-dependent LOD, including gaze-directed techniques
  - Non-photorealistic rendering using VDSlib as a framework

SIGGRAPH 2001

The End

SIGGRAPH 2001

Appendix:
Related Work

- Hoppe: Progressive Meshes (SIGGRAPH 96, 97, other papers)
  - Edge collapse vs. vertex merging
  - Pros:
    - Dynamic, view-dependent simplification
    - Elegant scheme for mesh attributes
  - Cons:
    - Requires clean mesh topology
    - Slow preprocess (though not implicit to PM)
    - Still per-object LOD

SIGGRAPH 2001

Appendix:
Web Resources

- VDSlib: *http://vdslib.virginia.edu*
  - A public-domain view-dependent simplification library
- My work on view-dependent simplification:
  http://www.cs.virginia.edu/~luebke/simplification.html
  - A SIGGRAPH paper
  - My dissertation on VDS
  - The attached tech report on VDS for CAD applications
  - A survey of LOD algorithms written for graphics developers

SIGGRAPH 2001

Appendix:
Web Resources

- Hughes Hoppe's work on progressive meshes:
  http://www.research.microsoft.com/~hhoppe
  - 2 SIGGRAPH papers
  - A paper on efficient implementation of progressive meshes
  - A paper on terrain rendering using progressive meshes
  - Much more…
- Michael Garland's work on quadric error metrics:
  http://www.uiuc.edu/~garland
  - A SIGGRAPH paper
  - Garland's dissertation on QEM
  - A follow-up paper extending QEM to surface attributes
  - Public domain code for generating LODs with QEM

SIGGRAPH 2001

Appendix:
Web Resources

- The Multi-Tesselation (MT) home page:
  http://www.disi.unige.it/person/MagilloP/MT/
  - A different approach to dynamic and view-dependent simplification by De Floriani, Magillo, and Puppo.
  - Includes code and sample software

SIGGRAPH 2001

Appendix:
Attached Papers

- David Luebke. *Robust View-Dependent Simplification For Very Large-Scale CAD Visualization*, University of Virginia Tech Report CS-99-33.
  - An updated version of the original SIGGRAPH '97 paper describing the view-dependent simplification framework presented here.

SIGGRAPH 2001

Appendix:
Attached Papers

- Benjamin Hallen, David Luebke. *Perceptually Driven Interactive Rendering*, University of Virginia Tech Report CS-01-01.
  - Describes ongoing work applying perceptual metrics (see Martin's talk) to view-dependent polygonal simplification.

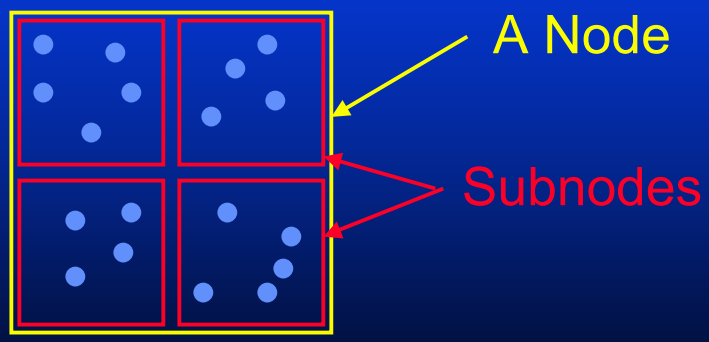
SIGGRAPH 2001

The Vertex Tree:
Livetris and Subtris

- Computing livetris and subtris:
  - `node->livetris` = triangles with exactly one corner vertex supported by node
  - `node->subtris` = triangles with:
    - Two or three corners in different subnodes
    - No two corners in the same subnode

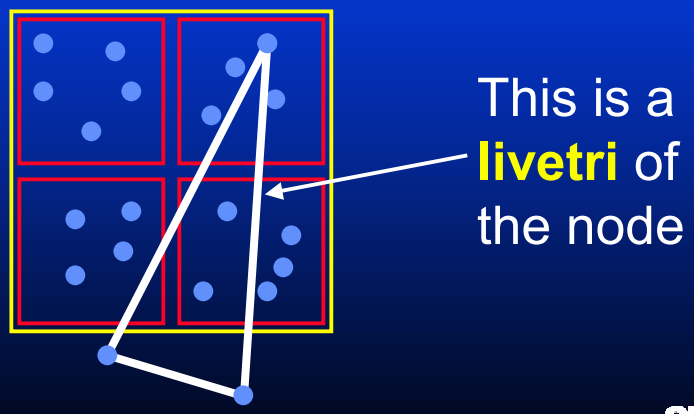
SIGGRAPH 2001

The Vertex Tree:
Livetris and Subtris



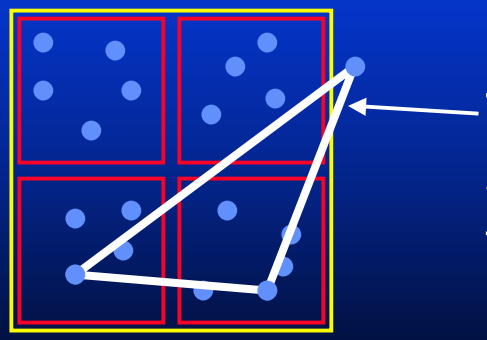
SIGGRAPH 2001

The Vertex Tree:
Livetris and Subtris



SIGGRAPH 2001

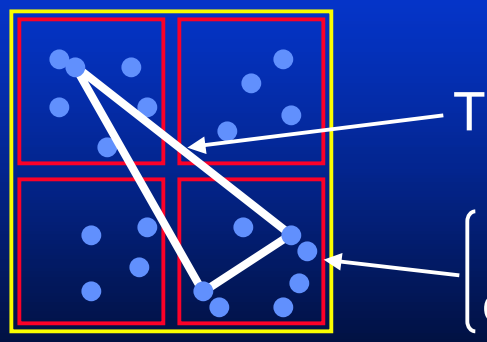
The Vertex Tree:
Livetris and Subtris



This is a **subtri** of the node

SIGGRAPH 2001

The Vertex Tree:
Livetris and Subtris



This is neither.

It's a subtri of this subnode

SIGGRAPH 2001

Algorithms for Generalized LODs

Amitabh Varshney
Department of Computer Science
University of Maryland at College Park

Outline

- Geometry and Topology Simplifications
- Implementing View-dependent LODs
- Parallel Computation of Continuous LODs
- Variable-Precision Rendering

Outline

- Geometry and Topology Simplifications
  - **Static Simplification of Topology**
  - View-dependent Simplification of Topology
- Implementing View-dependent LODs
- Parallel Computation of Continuous LODs
- Variable-Precision Rendering

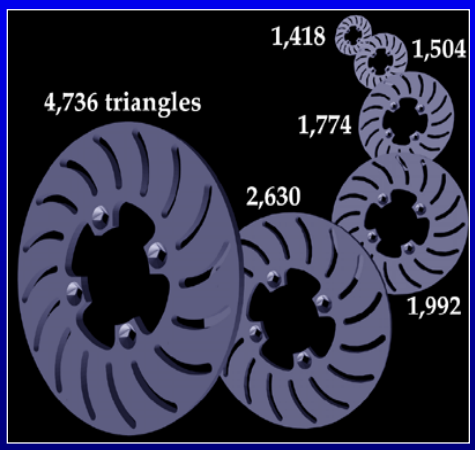
LOD Algorithms Classification

	View-Independent	View-Dependent
Topology Preserving	Turk 92, Schroeder et al 92 Cohen et al 96, Hoppe 96, Cignoni et al 98	Xia & Varshney 96, Hoppe 97, De Floriani et al 98 Gueziec et al 98, Klein et al 98
Topology Simplifying	Rossignac & Borrel 93, He et al 96, El-Sana & Varshney 97 Schroeder 97, Garland & Heckbert 97	Luebke & Erikson 97 El-Sana & Varshney 99

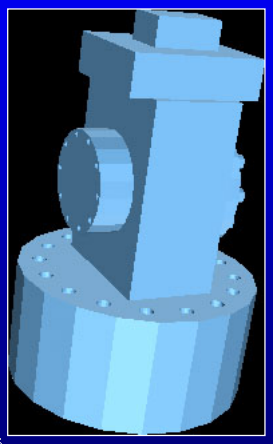
Geometry & Topology Simplifications

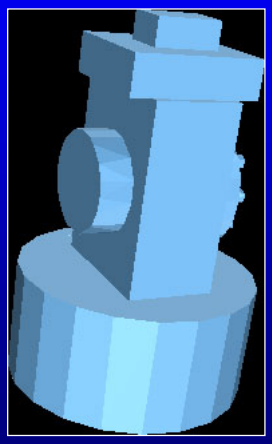
- Geometry Simplification
  - Reducing the number of geometric primitives (vertices, edges, triangles)
- Topology Simplification
  - Reducing the number of holes, tunnels, cavities
- Geometry + Topology Simplification
  - Aggressive simplifications
  - May not be suitable for some applications

Why Static Simplification of Topology?



Static Simplification of Topology





Local and Global Algorithms

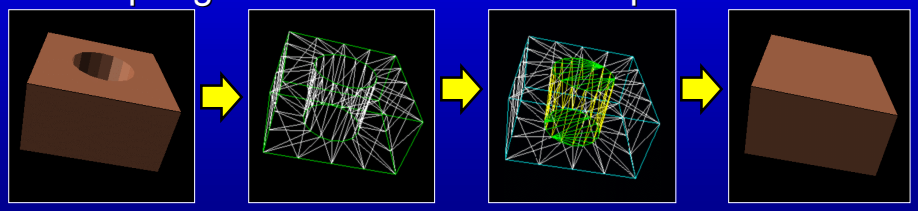
- Collapsing vertex pairs / virtual edges
  - Schroeder, *Visualization 97*
  - Popovic and Hoppe, *Siggraph 97*
  - Garland and Heckbert, *Siggraph 97*
- Collapsing primitives in a cell
  - Rossignac and Borrel, *Modeling in Comp. Graphics 93*
  - Luebke and Erikson, *Siggraph 97*

Local and Global Algorithms

- Low-pass filtering in Volumetric domain
  - He *et al.*, *Visualization 95*
- Rolling a sphere (L_2), cube (L_1, L_∞)
  - El-Sana and Varshney, *Visualization 97*

Our Approach to Static Topology Simplification

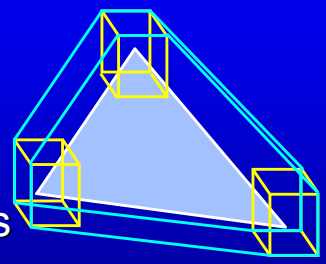
- Similar to α-Hulls
- Roll a sphere of radius α over the object
- Fill-up regions inaccessible to the sphere



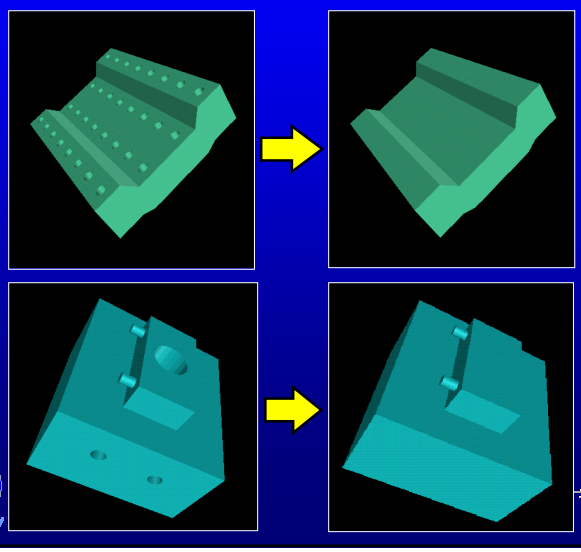
El-Sana and Varshney, Visualization 97, IEEE TVCG 98

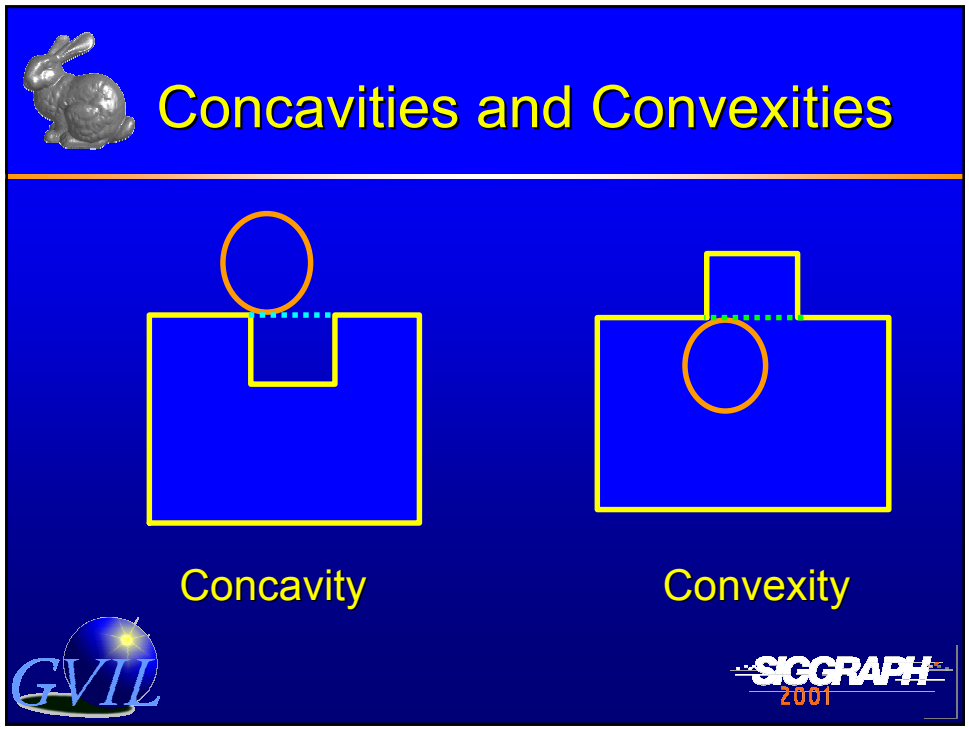
Alpha Prisms

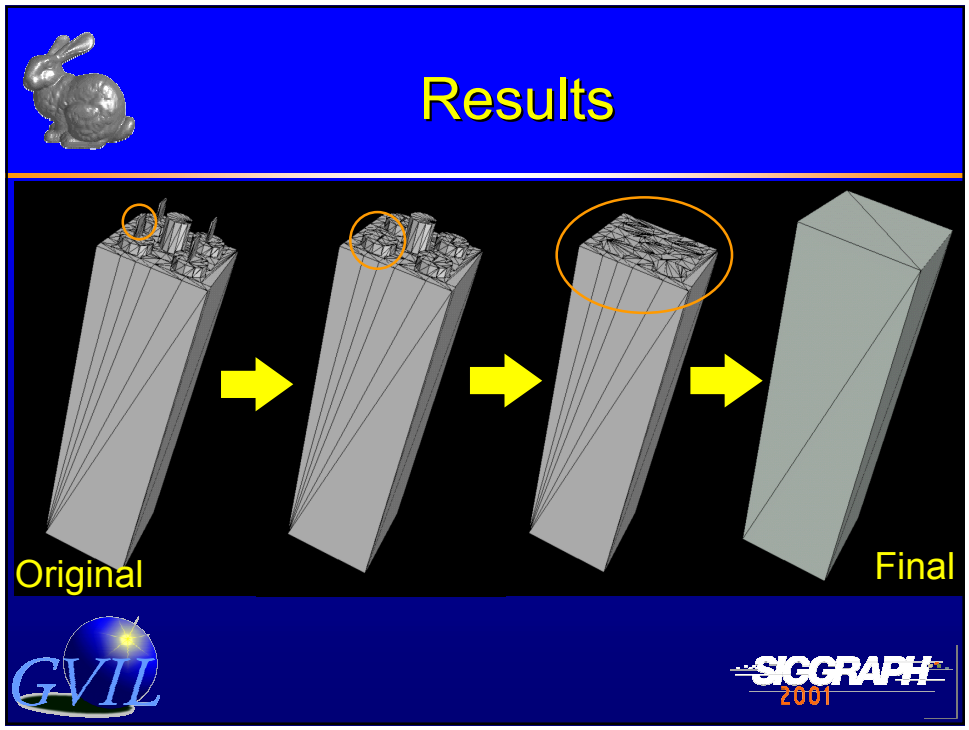
- Alpha prism
  - Convolve triangle with α-side cube (L_∞ metric)
  - Convex polyhedron
- Compute union of α prisms
  - Fills-up all features less than α
- Generate the surface from the union



Results



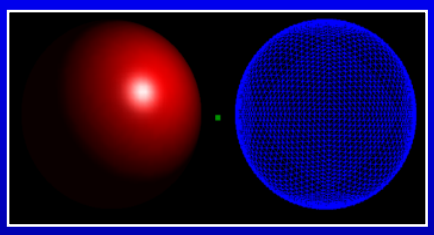

Concavities and Convexities
Concavity
Convexity
GVIL
SIGGRAPH
2001


Results
Original
Final
GVIL
SIGGRAPH
2001

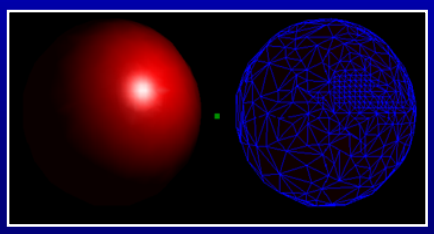
Outline

- Geometry and Topology Simplifications
  - Static Simplification of Topology
  - **View-dependent Simplification of Topology**
- Implementing View-dependent LODs
- Parallel Computation of Continuous LODs
- Variable-Precision Rendering

Illumination- and View-Dependent Detail



8192 triangles



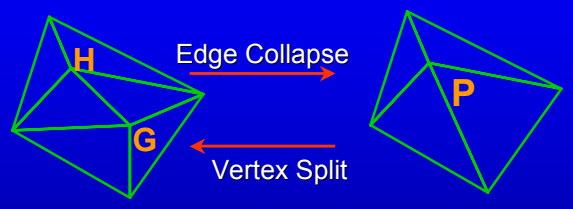
537 triangles

View-Dependent Topology Simplification

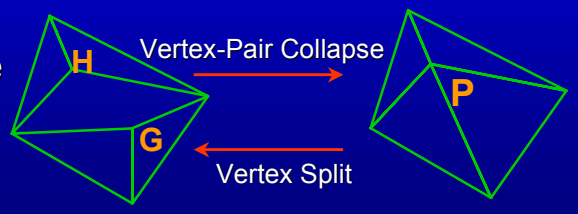
- Aggressive simplification
- Varied topology simplification
- Connect different objects
- Efficient fold-over prevention policy
- Real-time

Edge and Vertex-Pair Collapses

Edge Collapse



Vertex-Pair Collapse
(Virtual Edge)



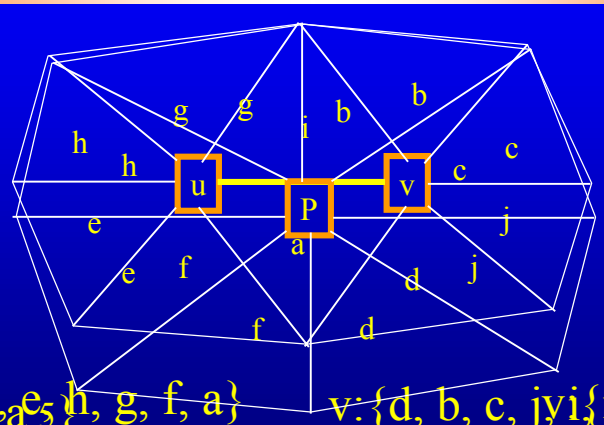
Simplifying Genus

- Allow virtual edge collapses
- Limit potentially $O(n^2)$ virtual edges
- Typical constraints:
  - Delaunay edges
  - Edges that span neighboring cells in a spatial subdivision: octree, grids, etc.
  - Maximum edge length

Virtual Edges

- Subdivide the dataset into patches
  - Initialize each triangle to a patch
  - Merge two patches that:
    - **Share at least one edge**
    - **Their normals differ less than threshold**
- Construct Delaunay triangulation using only the vertices on patch boundaries

Vertex Collapse



u:{j, e, h, g, f, a}

v:{d, b, c, j, i, a}

P:{e, h, g, f, d, b, c, j}

SIGGRAPH 2001

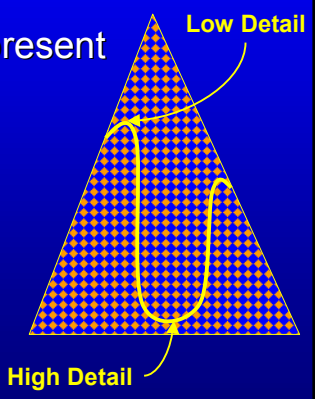
View Dependence Tree

- Use an appropriate distance metric
- Construct the set of virtual edges
- Build a heap of all the edges (virtual and real) using the given metric.
- While not empty (heap)
  - Extract (minimum) edge
  - Collapse its two vertices

SIGGRAPH 2001

View Dependence Tree

- Hierarchy of vertex-pair collapses
- Different levels in this hierarchy represent different levels of detail
- Construct the hierarchy offline
- Run-time navigation involves:
  - Vertex split: Refinement
  - Vertex collapse: Simplification



Display Triangles

- Initialized to triangles adjacent to roots
- Add triangles
  - Vertex Split
- Remove triangles
  - Edge Collapse
- Update triangle
  - Vertex Split
  - Edge Collapse

Simplification Factors

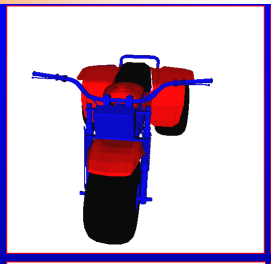
- Screen-space projection
- Local illumination
- Visibility culling
- Silhouette boundaries
- LOD transfer function
- Prevent fold-overs
  - Implicit Dependencies

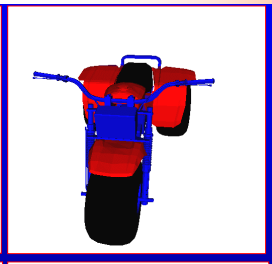
Run-time Traversal

```
for each active node n do
  switch(NextStat(n)){
    case SPLIT  : if ( CanSplit(n))
                      Split(n);
    case MERGE: if ( CanMerge(n) &&
                      CanMerge(Sibling(n))
                      Merge(n);
    case STAY   : // No Change on the active-nodes list }
```

Results

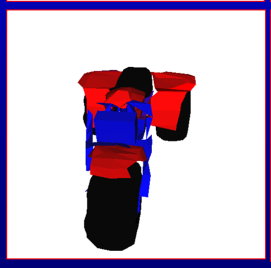
13.5K tris





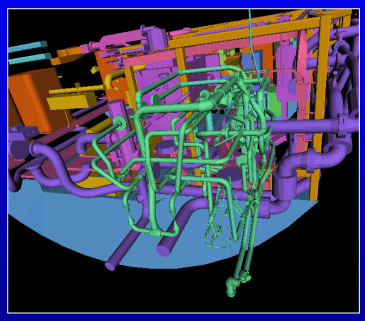
8.2K tris

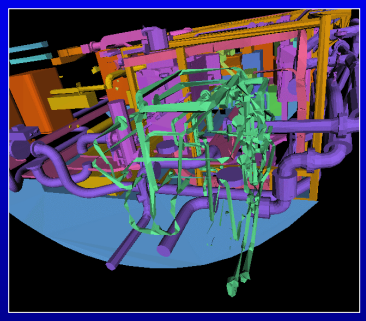
2.0K tris



200 tris

Results



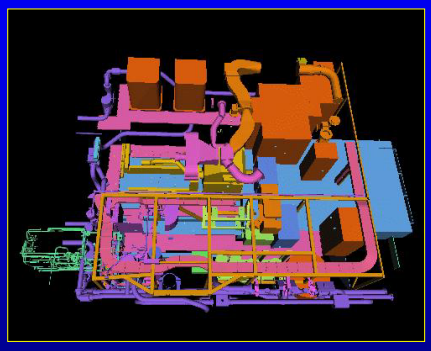


Close

Far

El-Sana and Varshney, Eurographics 99

Results



Original (340K tris)

Simplified (49K tris)

Auxilliary Machine Room Dataset

Ordering Geometry and Topology Simplifications

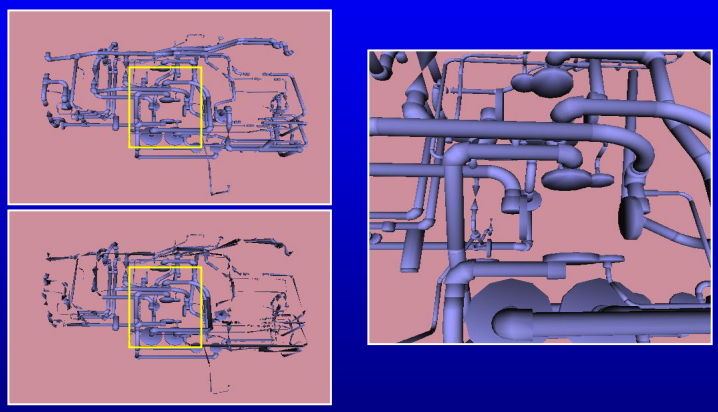
- Geometry-Simplification-First
  - Topology-preserving geometry simplification
  - Topology simplification
- Equal Preference
  - Treat Topology-preserving and Topology- reducing simplifications equally
  - Distance metrics that consider local surface normals *with* vertex coordinates fare better

Distance Function with Normals

Define Eye Normal as opposite of view direction
Define *distance* to use vertex coordinates and normals

- Backface Simplification
  - Backfacing triangles are *farther* away
- Foveation/View Frustum Detail
  - Triangles inside the field-of-view are *closer*
- Illumination
  - Regions with high diffuse illumination are *closer*

Foveation Results

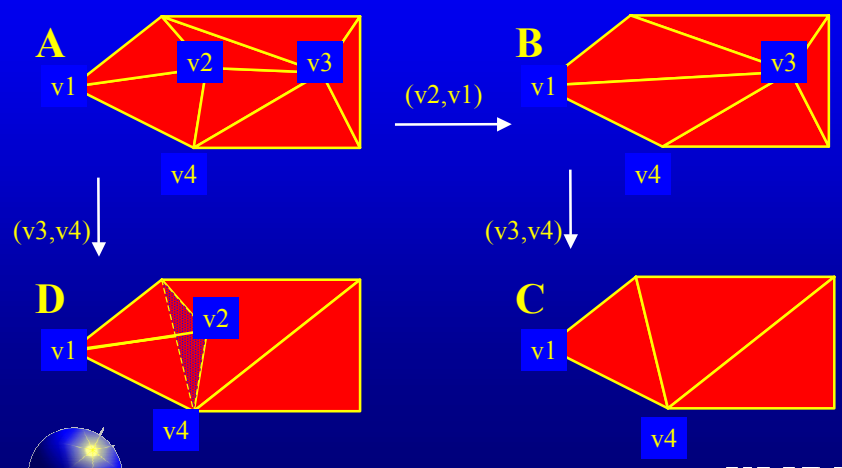


El-Sana and Varshney, Eurographics 99

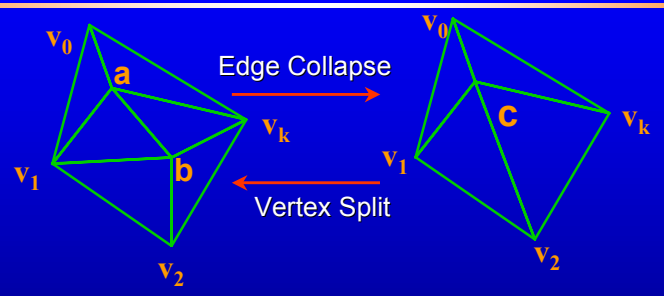
Outline

- Geometry and Topology Simplifications
- Implementing View-dependent LODs
  - Explicit and Implicit Dependencies
  - Maintaining triangle strips
- Parallel Computation of Continuous LODs
- Variable-Precision Rendering

Mesh Folding Problem

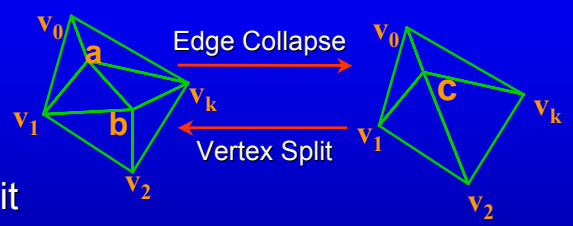


Explicit Dependencies



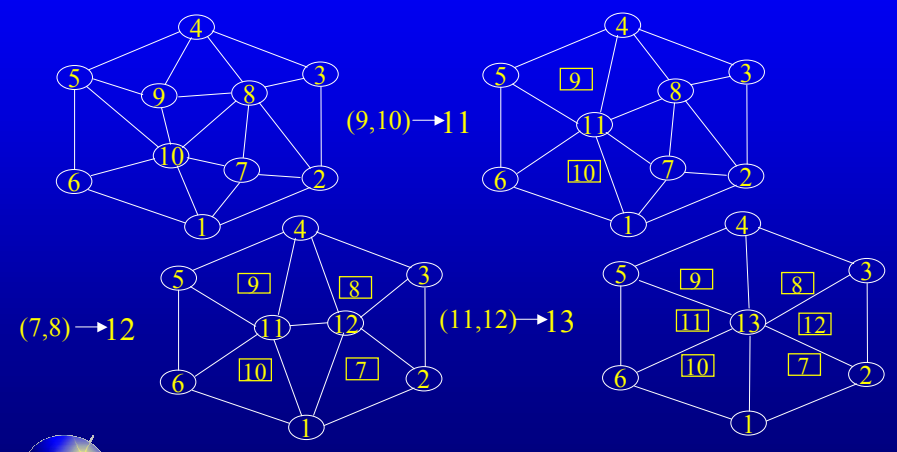
Neighborhood of an edge collapse is determined and fixed during preprocessing and used for validity checks at run-time

Explicit Dependencies



- Vertex split
  - Vertex *c* can split to *(a, b)* only if vertices v_0 , v_1 , … v_k are present and adjacent to *c* at run-time.
- Vertex-pair collapse
  - Vertex-pair *(a, b)* can collapse to vertex *c* only when all the vertices v_0 , v_1 , … v_k are present and adjacent to *(a, b)*.

Motivation for Implicit Dependencies



Implicit Dependencies

- Observations
  - Collapsibility graph is a Directed Acyclic Graph
  - Validity check involves determining the age of a node relative to its neighbors
- Solution
  - Each node is assigned a unique integer as *id*
  - Assign new nodes progressively increasing id-number

Implicit Dependencies

- Vertex ***v*** can split if:
  - Its *id* is greater than the *id* of all its neighbors
- Vertex-pair ***(u, v)*** can collapse to ***w*** if:
  - ***w*** 's *id* is less than the *id* of the parents of the neighbors of the two vertices (*u*,*v*)

Implicit Dependencies

- Vertex needs to maintain only two values:
  - Max ID of all its neighbors
  - Min ID of parents of all its neighbors
- Run-time checks become constant time
  - check against the above two values instead of all neighbors
- Localized memory accesses
  - Eg: Stanford Dragon (871K triangles, 874K nodes)
    - avg memory access distance for dependency checks comes down to ~1 byte from 14 MB

Outline

- Geometry and Topology Simplifications
- Implementing View-dependent LODs
  - Explicit and Implicit Dependencies
  - Maintaining triangle strips
- Parallel Computation of Continuous LODs
- Variable-Precision Rendering

Recent Research on Triangle Strips

Akeley, Haeberli, Burns, *1990*

Arkin et al. Visual Computer 96

Evans, Skiena, Varshney, *Visualization 96*

Duchaineau et al., Visualization 97

Gumhold & Strasser, *Siggraph 98*

El-Sana, Azanli, Varshney, Visualization 99

Xiang, Held, Mitchell, I3D *99*

Deering, *Siggraph 95*

Bar-Yehuda & Gotsman, ACM TOG 96

Chow, *Visualization 97*

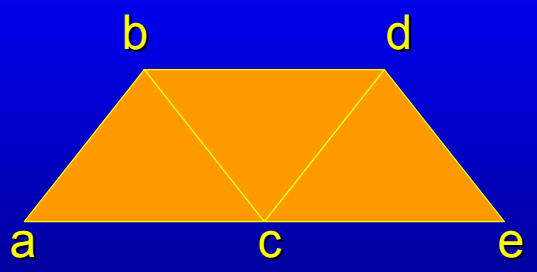
Speckmann & Snoeyink, *CCCG 97*

Taubin et al., *Siggraph 98*

Hoppe, *Siggraph 99*

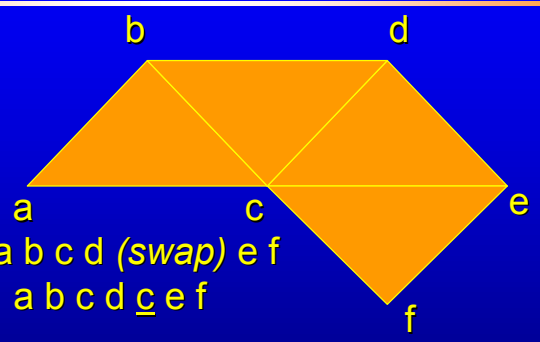
Velho et al., Visual Computer 99

Triangle Strip



Triangles: (abc), (bcd), (cde)
Triangle Strip: abcde

Generalized Triangle Strips



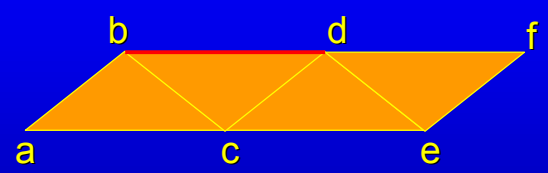
Triangle Strip: a b c d *(swap)* e f
a b c d <u>c</u> e f

Repeating vertices changes direction

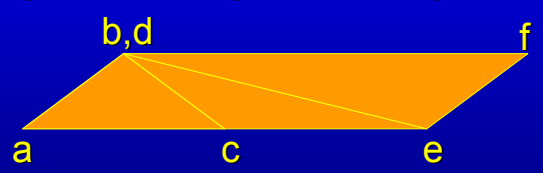
Triangle Strips with LODs

- Triangle strips
  - 2X speedup
  - Hardware / Software support
- Static LODs
  - Off-line computation of triangle strips per LOD
- View-dependent simplification
  - Connectivity changes every frame
  - Requires run-time update of triangle strips

Edge Collapse in a Triangle Strip



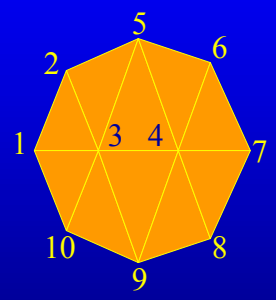
(abcdef)



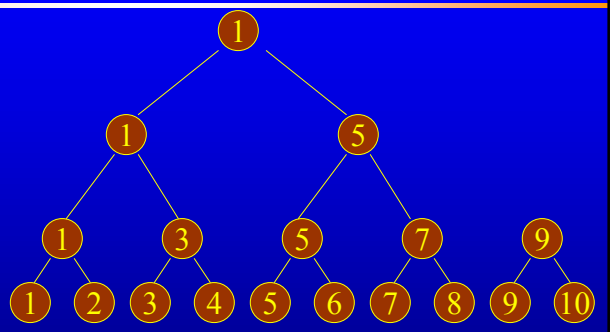
(abcbef)

Repeating vertices can represent edge collapses

Merge Tree

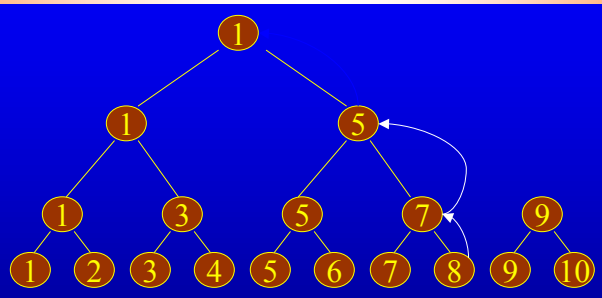


Mesh



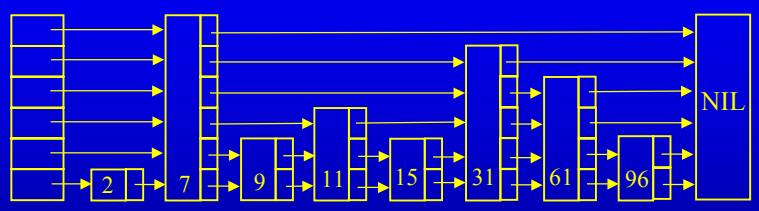
Merge Tree

Following Parent Pointers



- Replace each Triangle Strip vertex by its closest active ancestor
- Need efficient pointer hopping

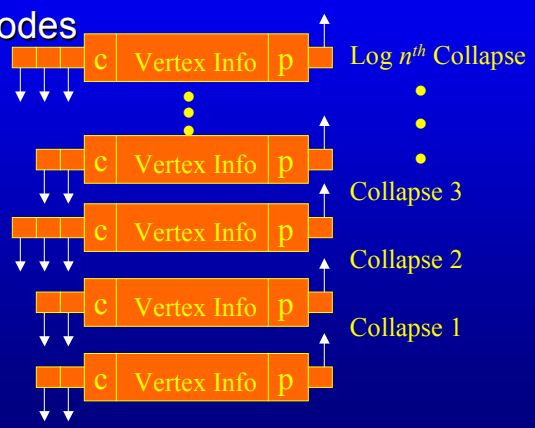
Skip Lists



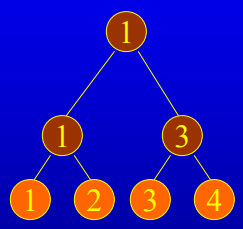
- Pugh [CACM 1990]
- Probabilistic balancing
- Fast searches
- Compressed trees

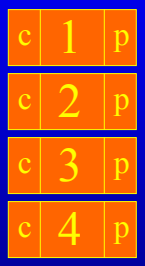
Skip Strip Data Structure

- Array of Skip Strip Nodes
- Merge
  - Increment p
  - Increment c
- Split
  - Decrement p
  - Decrement c

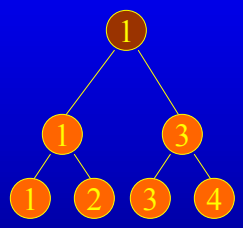


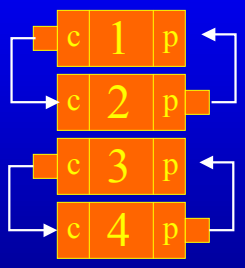
Building a Skip Strip



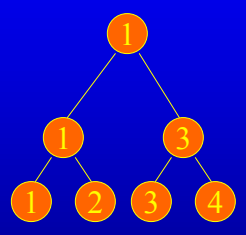


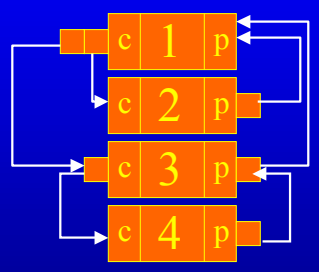
Building a Skip Strip



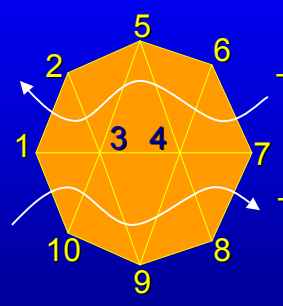


Building a Skip Strip





Skip Strips with LODs

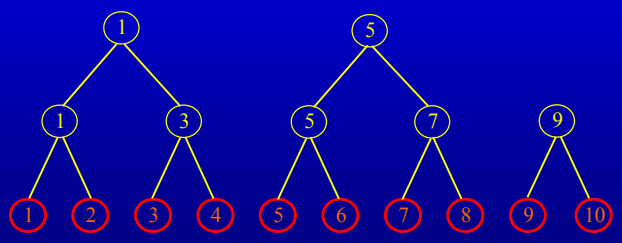


Triangle Strip A (7 6 4 5 3 2 1)

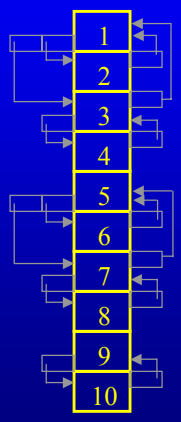
Triangle Strip B (1 10 3 9 4 8 7)

Skip Strip Example

Triangle Strip A: 7 6 4 5 3 2 1
Display Strip A: 7 6 4 5 3 2 1
Triangle Strip B: 1 10 3 9 4 8 7
Display Strip B: 1 10 3 9 4 8 7



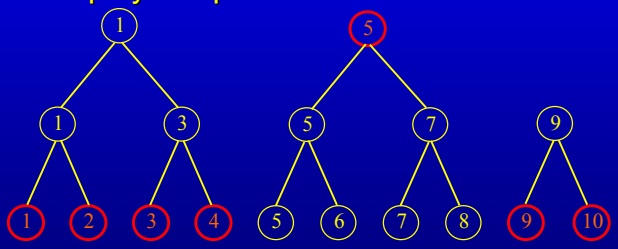
Highest Resolution



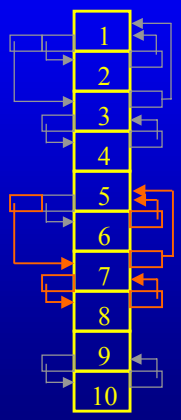
SIGGRAPH 2001

Skip Strip Example

Triangle Strip A: 7 6 4 5 3 2 1
Display Strip A: 5 5 4 5 3 2 1
Triangle Strip B: 1 10 3 9 4 8 7
Display Strip B: 1 10 3 9 4 5 5



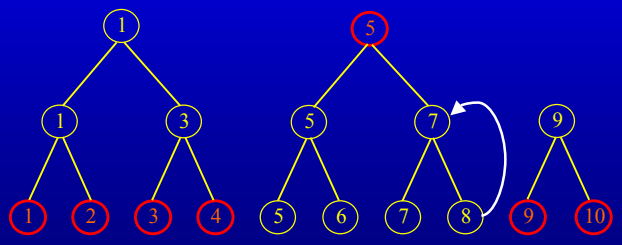
Lower Resolution

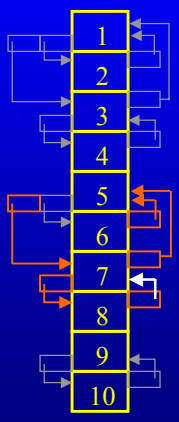


SIGGRAPH 2001

Skip Strip Example

Triangle Strip A: 7 6 4 5 3 2 1
Display Strip A: 5 5 4 5 3 2 1
Triangle Strip B: 1 10 3 9 4 8 7
Display Strip B: 1 10 3 9 4 5 5

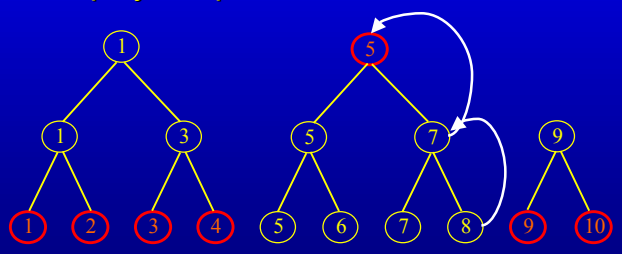


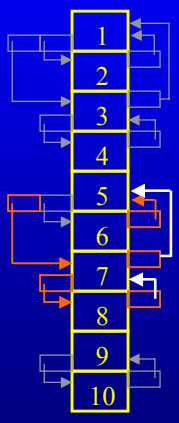


Lower Resolution

Skip Strip Example

Triangle Strip A: 7 6 4 5 3 2 1
Display Strip A: 5 5 4 5 3 2 1
Triangle Strip B: 1 10 3 9 4 8 7
Display Strip B: 1 10 3 9 4 5 5

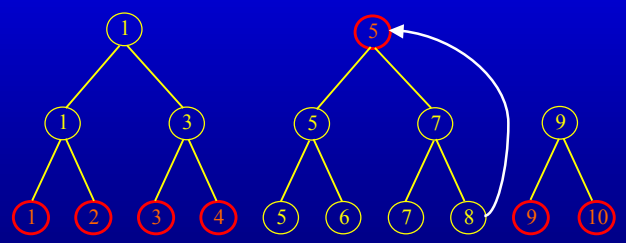




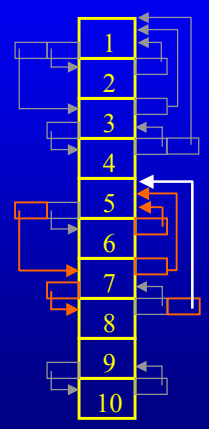
Lower Resolution

Optimized Skip Strip Example

Triangle Strip A: 7 6 4 5 3 2 1
Display Strip A: 5 5 4 5 3 2 1
Triangle Strip B: 1 10 3 9 4 8 7
Display Strip B: 1 10 3 9 4 5 5



Lower Resolution



Real-Time Display

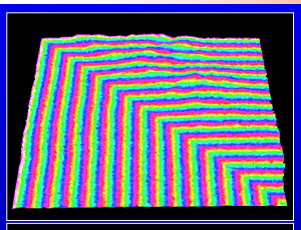
- Determine the display vertices
- Determine the display strips
  - Determine which strips have changed
  - Traverse the changed triangle strips
  - Follow Skip Strip pointers to get appropriate ancestors
  - Remove redundant vertices

Efficient Skipping

- Reduce the traversed non-active vertices
  - Compress the traversed paths
  - Update the compressed paths in lazy fashion
  - Cache the active path index
- Efficiency
  - We use only *O(log log n)* jumps

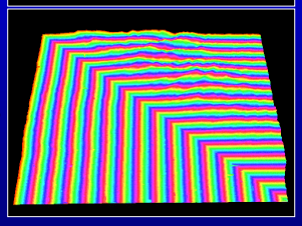
Results: Terrain

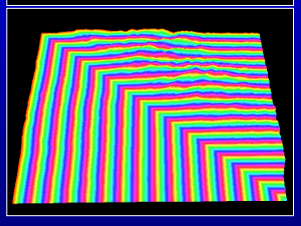
255K tris



32K tris

255K tris

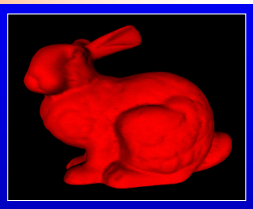


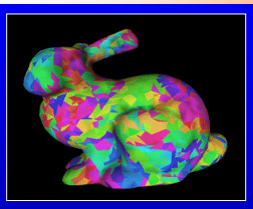


522K tris

Results: Bunny

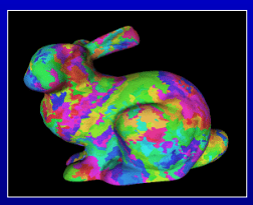
30K tris

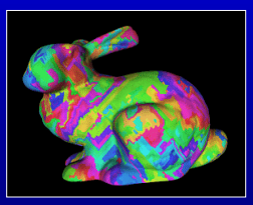




5K tris

30K tris



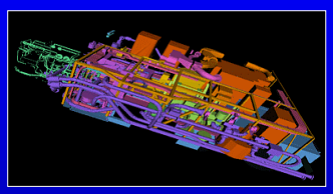


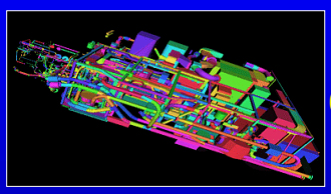
65K tris

El-Sana and Varshney, Visualization 99

Auxiliary Machine Room

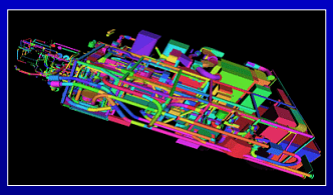
65K tris

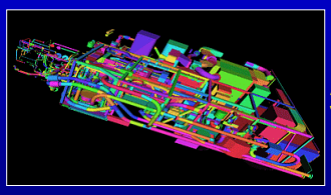




65K tris

170K tris





340K tris

El-Sana and Varshney, Visualization 99

Results

- Skip Strips make execution of split and merge operations more efficient
- Applicable to any hierarchical vertex scheme
- Four our four sample datasets:
  - Skip Strips provided ~ 1.5 to 2.0 X improvement over sending raw view-dependent triangles
  - Skip Strips provided ~ 1.6 to 1.7 X improvement over per-frame greedy triangle strip generation

Outline

- Geometry and Topology Simplifications
- Implementing View-dependent LODs
- Parallel Computation of Continuous LODs
  - Mesh Partitioning
  - Load Balancing
- Variable-Precision Rendering

Why Parallel Processing?

- Large datasets
  - Can be visualized faster
- Parallel machines widespread
- Interactive design
  - Changes and updates
- Interleave acquisition of range data with Merge Tree construction
  - 3D Camcorder (IBM)

Parallel Preprocessing

- Sequential Initialization
  - Small fraction
- Parallel Processing
  - Small critical section
  - Lock-free processing

Sequential Initialization

- Load Dataset
- Subdivide vertices
  - Using Octree
  - Depth depends on the cell size limit
- Subdivide edges in a disjoint manner
  - Edge $E_i = (V_0, V_1)$ belongs to the cell of V_j, where $id(V_j) = \text{MAX}\ (id(V_0)\ ,\ id(V_1))$

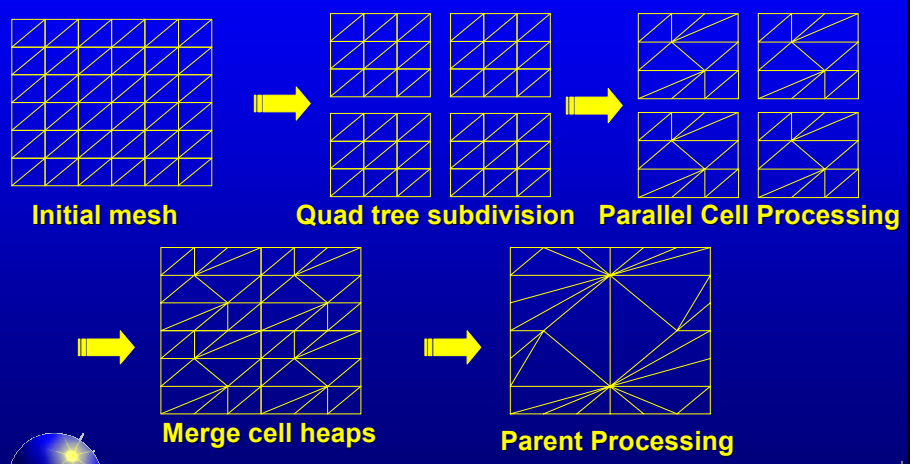
Parallel Cell Processing

- If current cell is a leaf
  - Compute the edge list
  - Build the edge heap
- While (length(edge) < cell_side)

  if valid_edge_collapse(edge)

  collapse(edge)
- After processing all eight child cells
  - Merge heaps of child cells
  - Assign to parent

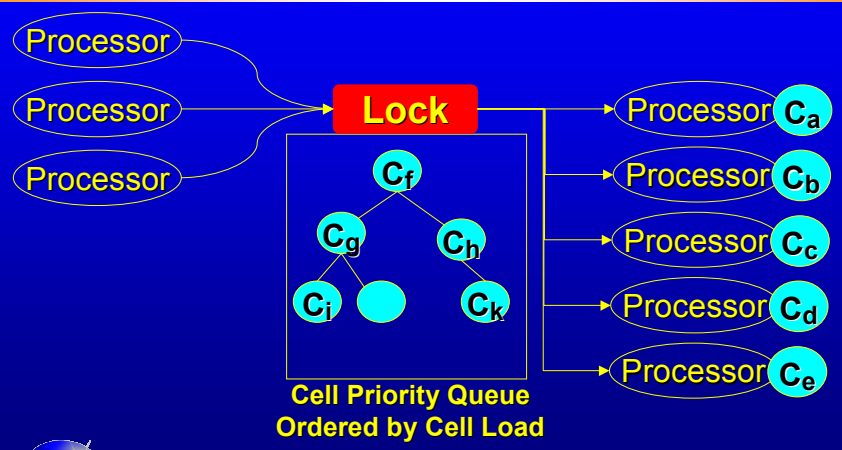
Cell Processing



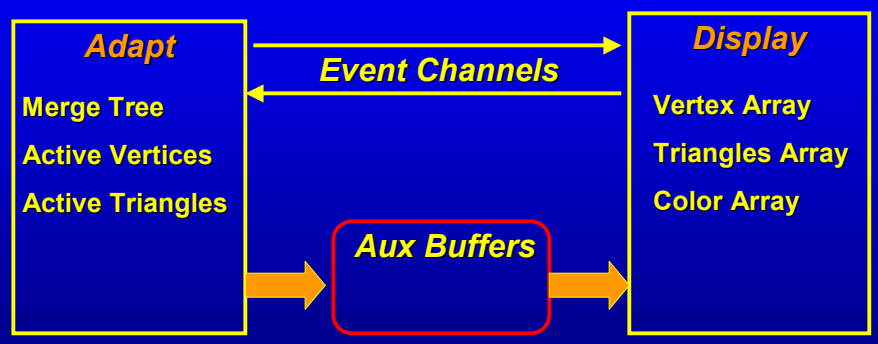
Cells Queue

- Large datasets ⇒ large number of cells
- Usually, Number of cells >> Processors
- Process lightly-loaded cells first
- Save the cells in queue sorted by number of edges

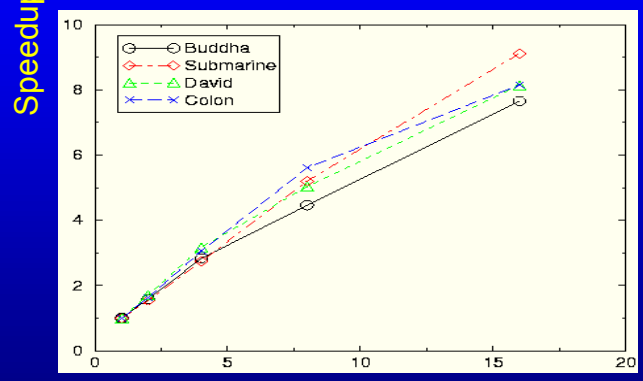
Algorithm Flow



Parallel Navigation



Results



El-Sana and Varshney, SPIE Electronic Imaging 99

Results

- Linearly Scalable Algorithm
- Highly Load Balanced
- Applications:
  - Large Mesh Visualization
  - Interleaved 3D Data Acquisition and Display
  - Interactive CAD

Outline

- Geometry and Topology Simplifications
- Implementing View-dependent LODs
- Parallel Computation of Continuous LODs
- Variable-Precision Rendering

Defining Level of Detail

- *Number* of Primitives
- *Precision* of primitives
  - –Colors (Heckbert *82*, Xiang *97*)
  - –Normals (Deering *95*, Zhang & Hoff *97*)
  - –Vertex coordinates (King & Rossignac *99*)

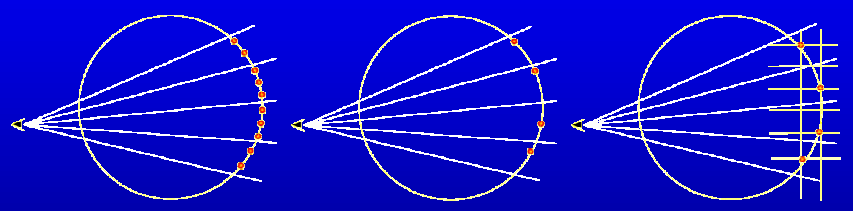
Variable-Precision Rendering

- Reduce the precision of graphics primitives
- Relate the number of bits of input precision for a given display accuracy
- Speedup 3D transformation and lighting by taking advantage of SIMD parallelism
- Explore spatio-temporal coherence

Related Work

Sugihara *89*
Milenkovic & Nackman *90*
Rossignac & Borrel *93*
Deering *95*
Fortune & Van Wyk *96*
Chow *97*
Luebke & Erikson *97*
Fortune *98*
Taubin & Rossignac *98*
Taubin et al. *98*
Li & Kuo *98*
Cohen-Or et al. *99*
Bajaj et al. *99*
King & Rossignac *99*
Bajaj et al. *2000*
Pajarola & Rossignac *2000*

Variable-Precision vs. Multiresolution



Original Multiresolution Variable Precision

Assumptions

- Minimum-sized cube covering the object
  - x, y, z normalized to range [-1.0, 1.0]
- N-bit fixed-point representation of operands
- Rounding to the nearest integer
- Worst-case study

Error Analysis

- Representation error of a
  - Half bit, i.e., $\varepsilon_a \le \frac{1}{2}$
- Addition error of $(a+b)$

$$\varepsilon_{(a+b)} \le \varepsilon_a + \varepsilon_b \le \frac{1}{2} + \frac{1}{2} = 1, \text{ i.e., lose one bit of accuracy}$$

- Multiplication error of $(a \times b)$

$$\varepsilon_{(a\times b)} \le |a\varepsilon_b + b\varepsilon_a| \le \frac{1}{2} + \frac{1}{2} = 1, \text{ i.e., lose one bit of accuracy}$$

Error Analysis

- Division error of (a/b)

Generated error $\varepsilon_{(a/b)}^{gen} < 1$ due to truncation

Propagated error $\varepsilon_{(a/b)}^{prop} = \frac{\partial}{\partial a}\left(\frac{a}{b}\right)\varepsilon_a + \frac{\partial}{\partial b}\left(\frac{a}{b}\right)\varepsilon_b$

$$= \frac{\varepsilon_a}{b} + \frac{a}{b^2}\varepsilon_b \le \frac{1}{b} \quad \begin{pmatrix} \varepsilon_a, \varepsilon_b < 1/2 \\ a < b \end{pmatrix}$$

$$\varepsilon_{(a/b)} = \varepsilon_{(a/b)}^{gen} + \varepsilon_{(a/b)}^{prop} < 1 + \frac{\text{distance of far plane from eye}}{\text{distance of scene vertex to eye}}$$

Putting it all together

$$m = n + 3 + \left\lceil \log_2\left(1 + \frac{\text{distance of far plane from eye}}{\text{distance of scene vertex to eye}}\right)\right\rceil$$

m is number of bits of input data

n is output accuracy after transformation

e.g., 1024×1024 window with pixel-level accuracy

object half-way across the view volume

$n = 10$

$m = 10 + 3 + \lceil \log_2(1+2) \rceil = 15$

View-dependent Transformation

- Construct bounding volume hierarchy
- Find the projected size of the object
- Determine the nearest visible vertex accuracy

$$near_bits = m - \left\lfloor \left| \log_2\left(\frac{\text{projected range}}{2}\right) \right| \right\rfloor$$

View-dependent Transformation

Accuracy needed for each vertex:

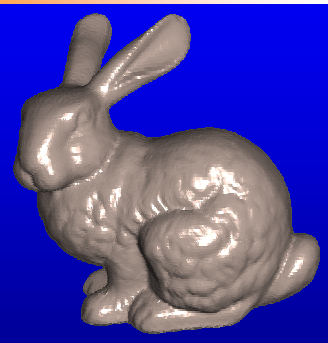
$$vertex_bits = near_bits - \left\lfloor \log_2\left(\frac{\text{transformed W of the vertex}}{\text{distance of nearest vertex to eye}}\right) \right\rfloor$$

Compute by using bounding volume hierarchy

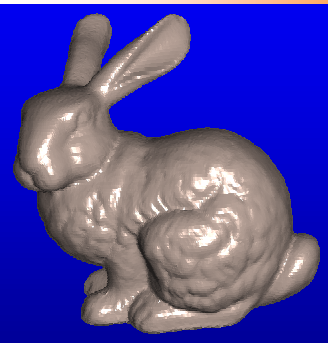
Spatio-temporal Coherence

- Spatial coherence
  - Using differences in neighboring vertices
  - $M x' = M (x + \Delta x) = M x + M \Delta x$
  - Top-down octree traversal
- Temporal Coherence
  - Frame-to-frame
  - $M' x = (M + \Delta M) x = M x + \Delta M x$
  - Can be combined with spatial coherence

Transformation Result



Floating Point
(32 bits/vertex coordinate)



Variable Precision
(7.9 bits/vertex coordinate)

Variable-Precision Lighting

$$
\begin{aligned}
Color = {} & emission_{mat} + ambient_{model} \times ambient_{mat} + \\
& \sum_{i=0}^{m-1} \left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i \times (spotlight_effect)_i \times \\
& (ambient_{light} \times ambient_{mat} + \\
& (\max\{\mathbf{L} \cdot \mathbf{N}, 0\}) \times diffuse_{light} \times diffuse_{mat} + \\
& (\max\{\mathbf{H} \cdot \mathbf{N}, 0\})^{shin} \times specular_{light} \times specular_{mat})_i
\end{aligned}
$$

Sources of Illumination Errors

- Operands with different accuracy
- Square-root operation error
- Specular exponentiation error

Illumination Errors

Operands a and b with different bits of accuracy:
n (for a) and n' (for b) and $n > n'$

- Addition error of $(a+b)$

$$\varepsilon_{(a+b)} \le \varepsilon_a + \varepsilon_b \le 2^{-(n+1)} + 2^{-(n'+1)} \approx 2^{-(n'+1)}$$

i.e. same accuracy as the less accurate operand

- Multiplication error of $(a \times b)$

$$\varepsilon_{(a\times b)} \le |a\varepsilon_b + b\varepsilon_a| \le 2^{-(n+1)} \times 1 + 2^{-(n'+1)} \times 1 \approx 2^{-(n'+1)}$$

Same as in the addition case

Illumination Errors

Square-root operation error

- Fixed-point operation ⇒ table lookup
- 2n bits operand
- Most significant n bits as index
- The result will have n bits of accuracy for most cases

Illumination Errors

Specular exponentiation error of (a^{shin})

- Operand a with n bits accuracy, $shin < 128$
- Error maximized by large a, ε_a, and $shin$

$$(a+\varepsilon_a)^{shin} \approx a^{shin} + a\varepsilon_a \times shin \ \ (if\ \varepsilon_a << a)$$

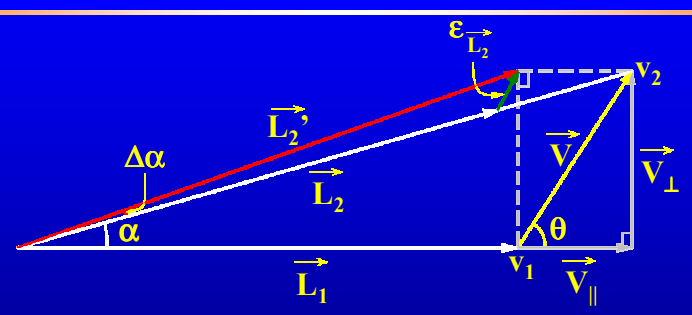
$$\varepsilon_{(a^{shin})} \approx a\varepsilon_a \times shin < 1 \times 2^{-(n+1)} \times 128 = 2^{-(n-6)}$$

i.e., lose 6 bits of accuracy at most.

Illumination Errors

- Dot-product of vectors lose ~ 1-2 bits accuracy
- Putting it all together
  - Specular (least accurate) decides the overall accuracy
  - lose 1 bit for normalization, 1~2 bits for dot product, 6 bits for exponentiation
  - Total loss of accuracy: 8 ~ 9 bits
  - So: *m = n+8 or n+9*
    (n = output accuracy, m = input accuracy)

Incremental Lighting

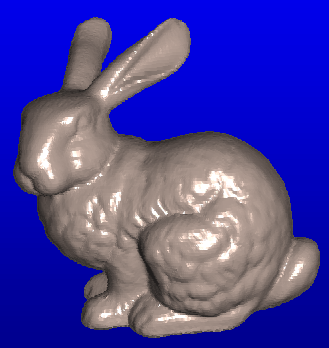


The error of using L_2' as an estimate of L_2 is in the order of $\left(\|\vec{V}\|^2 / 2\|\vec{L_1}\|^2\right)$

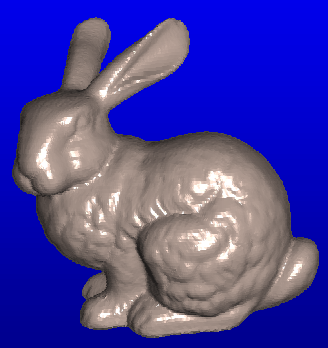
Implementation Notes

- Vertices processed in groups as a tradeoff between
  - L2 cache size
  - Expensive cost of resetting MMX register flag between changes in operand types
- Avoid error buildup
  - Matrix setup and composition per frame is full precision
  - Transformations are variable precision
  - Computation cost is negligible

Results: Bunny

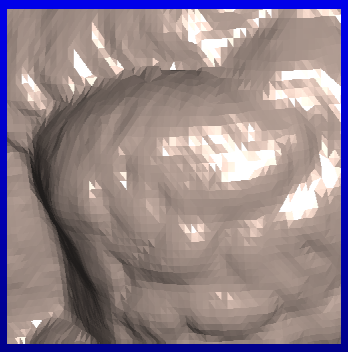


Floating Point

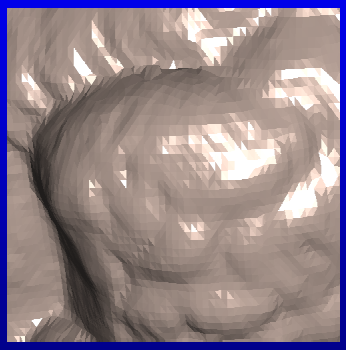


Variable Precision

Results: Bunny

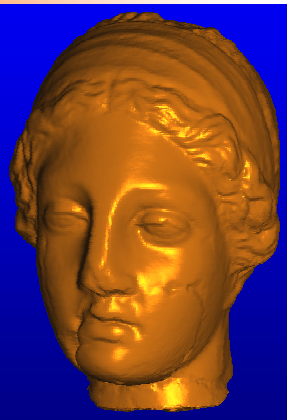


Floating Point
Close-up

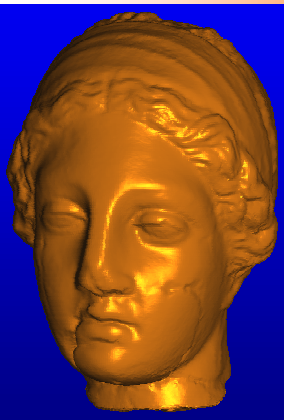


Variable Precision
Close-up

Results: Venus

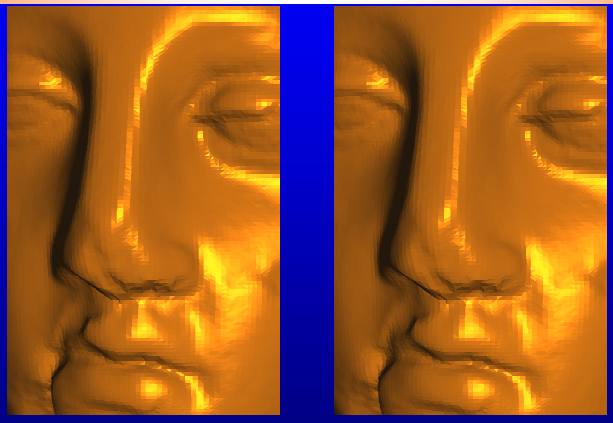


Floating Point



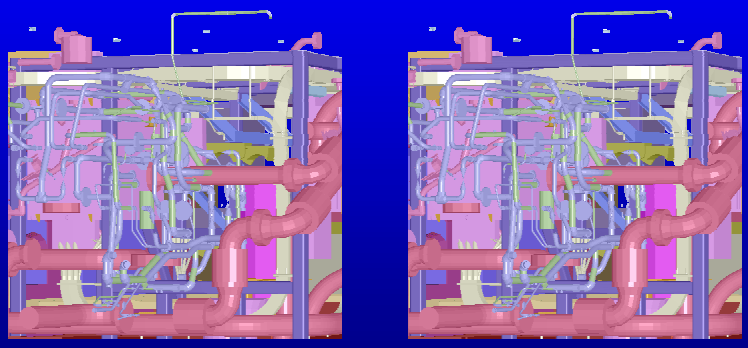
Variable Precision

Results: Venus



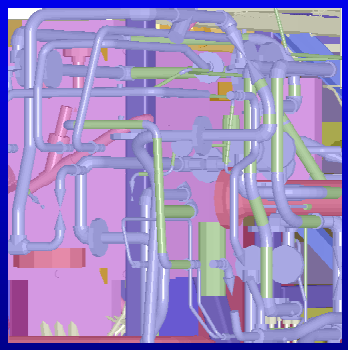
Floating Point Variable Precision

Results
Auxiliary Machine Room

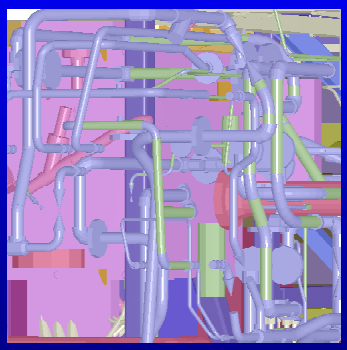


Floating Point Variable Precision

Results
Auxiliary Machine Room



Floating Point
Close-up



Variable Precision
Close-up

Conclusions

- More efficient transformation and lighting
- Complementary to multiresolution approaches
- For the datasets we tested
  - –Using PII 400MHz PC with 128M RAM
  - –Voodoo3 3500 graphics card and Glide API
  - –Provides a factor of 4 or more speedup

Conclusions

- Static and View-dependent LODs for simplifications of geometry and topology
- Implicit Dependencies for localizing data accesses
- Skip Strips: Updating triangle strips with view-dependent LODs
- Parallel LOD computation and navigation
- Variable-Precision transformations and lighting

Acknowledgements

National Science Foundation
Dept of Defense
Honda R & D, North America
General Dynamics
UMIACS, University of Maryland
SUNY Stony Brook
Stanford Graphics Lab
Cyberware, Inc.

Measuring Simplification Error

Jonathan Cohen

Computer Science Department
Johns Hopkins University

<cohen@cs.jhu.edu>
http://www.cs.jhu.edu/~cohen

SIGGRAPH 2001

What I'll Talk About

- Introduction
- Geometric Error
- Attribute Error
- Appearance-Preserving Simplification

SIGGRAPH 2001

Why Measure Error?

- Guide simplification process
  - Making better choices produces better simplifications
- Know quality of results
  - Object-space error bounds describes quality
- Know when to show a particular LOD
  - Which LOD for a given screen-space error
- Balance quality for large environments
  - What error bound for a given polygon count

SIGGRAPH 2001

Ford Bronco Model

Triangles:
41,855
27,970
20,922
12,939
8,385
4,766

courtesy of Division and Viewpoint

SIGGRAPH 2001

What I'll Talk About

- Introduction
- Geometric Error
- Attribute Error
- Appearance-Preserving Simplification

SIGGRAPH 2001

Geometric Error Measures

- Promote accurate 3D shape preservation
- Also preserves screen-space shape
  - Silhouettes
  - Pixel coverage

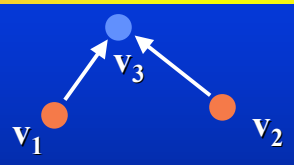
SIGGRAPH 2001

Classifying Geometric Error Metrics

- Vertex-Vertex Distance
- Vertex-Plane Distance
- Point-Surface Distance
- Surface-Surface Distance

SIGGRAPH 2001

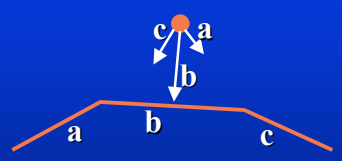
Vertex-Vertex Distance



- $E = \max(\| v_3 - v_1 \|, \| v_3 - v_2 \|)$
- Appropriate during topology changes
  - Rossignac and Borrel 93
  - Luebke and Erikson 97
- Loose for topology-preserving collapses

SIGGRAPH 2001

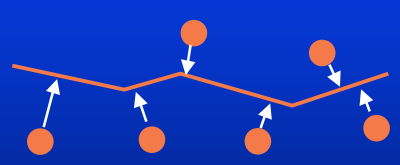
Vertex-Plane Distance



- Store set of planes with each vertex
  - Error based on distance from vertex to planes
  - When vertices are merged, merge sets
- Ronfard and Rossignac 96
  - Store plane sets, compute max distance
- *Error Quadrics* - Garland and Heckbert 96
  - Store quadratic form, compute sum of square distances

SIGGRAPH 2001

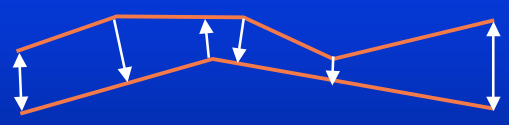
Point-Surface Distance



- Used in Hoppe 93 and 96
- Map point set to closest points on simplified surface
- Compute sum of square distances

SIGGRAPH 2001

Surface-Surface Distance



- Bound maximum distance between input and simplified surfaces
  - Tolerance Volumes - Guéziec 96
  - Simplification Envelopes - Cohen/Varshney 96
  - Hausdorf Distance - Klein 96
  - Mapping Distance - Bajaj/Schikore 96, Cohen et al. 97

SIGGRAPH 2001

Vertex-Vertex != Surface-Surface

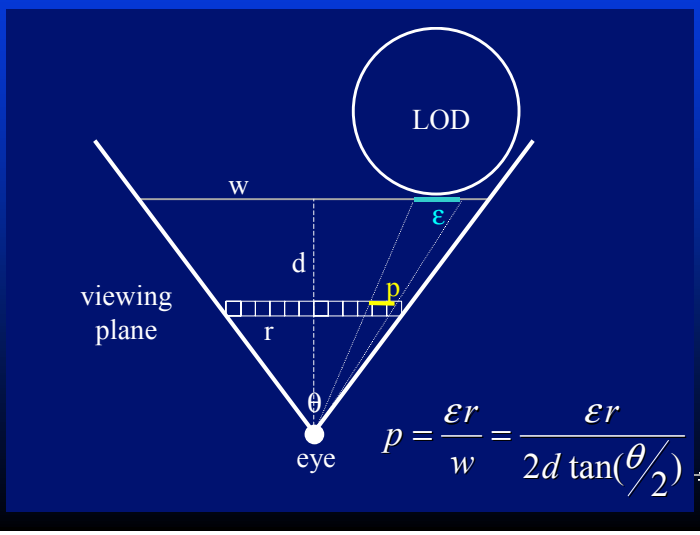
- Error is zero at vertices and exterior edges
- Error is non-zero everywhere else
  - not captured by vertex-vertex or vertex-plane metrics

SIGGRAPH 2001

Memoryless Simplification

- Lindstrom/Turk 98
- No measure of error from original mesh
  - *Incremental* rather than total error
- Preserve volume and area as simplification progresses
- Low error demonstrated after-the-fact
  - Metro - Cignoni et al. 96

Screen-space Geometric Error



$$p = \frac{\varepsilon r}{w} = \frac{\varepsilon r}{2d\tan(\theta/2)}$$

Geometric Error Observations

- Vertex-vertex and vertex-plane distance
  - Fast
  - Low error shown after-the-fact, but not guaranteed by metric
- Cannot guarantee quality without surface-surface distance bound
- Hoppe's point-surface approximates one-sided surface-surface
- Good error measures useful at run-time
  - 3D average or maximum error distance

SIGGRAPH 2001

Video Example

Cohen, Varshney, et al., "Simplification Envelopes," *Proceedings of SIGGRAPH 96.*

SIGGRAPH 2001

What I'll Talk About

- Introduction
- Geometric Error
- Attribute Error
- Appearance-Preserving Simplification

SIGGRAPH 2001

Attribute Error Metrics

- Attributes include colors, normals, and texture coordinates
- Promote accuracy of final pixel colors

SIGGRAPH 2001

Classifying Attribute Error Metrics

- Vertex-Vertex Distance
- Vertex-Plane Distance
- Point-Surface Distance
- Surface-Surface Distance

SIGGRAPH 2001

Vertex-Vertex Distance

- GAPS point clouds - Erikson/Manocha 98
  - Measure sum of square distances from vertex to its constituent vertices (area-weighted)
  - Used for colors, normals, and texture coordinates
  - Stored as 5 floats for 3D attributes (e.g. rgb)
- Normal cones
  - Luebke/Erikson 97, Xia et al. 97

SIGGRAPH 2001

Vertex-Plane Distance

- Higher-dimensional error quadrics
  - Garland and Heckbert 98
  - Vertices live in higher-dimensional position + attribute space
  - Planes defined in this space
- Multiple attribute quadrics
  - Hoppe 99
  - Decouples affects of position and attributes
  - Reduces storage and computational complexity

SIGGRAPH 2001

Point-Surface Distance

- Extension of geometric point-surface distance
  - Hoppe 96
- Geometric correspondences found between original surface samples and simplified surface
- Sum of square attribute distances minimized
- Used primarily for vertex colors

SIGGRAPH 2001

Surface-Surface Distance

- Bajaj / Schikore 96
  - Geometric projections provide local mappings
  - Maximum distance of scalar attributes measured over surface

SIGGRAPH 2001

Screen-space Attribute Error

- Texture coordinates work like geometric error
  - Cohen et al. 98
- Normal error controls dynamic refinement around highlights
  - Xia et al. 97, Klein 98
  - Doesn't allow more simplification as objects recede
- Color control?

SIGGRAPH 2001

What I'll Talk About

- Introduction
- Geometric Error
- Attribute Error
- Appearance-Preserving Simplification
  - Cohen et al. 98

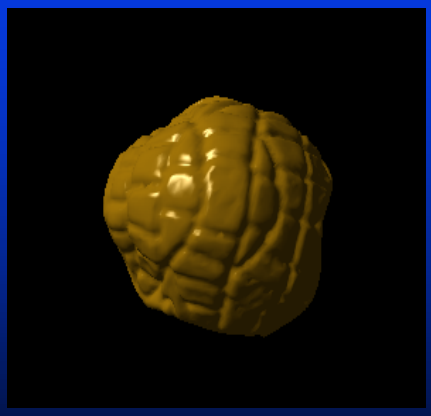
SIGGRAPH 2001

Appearance Preservation

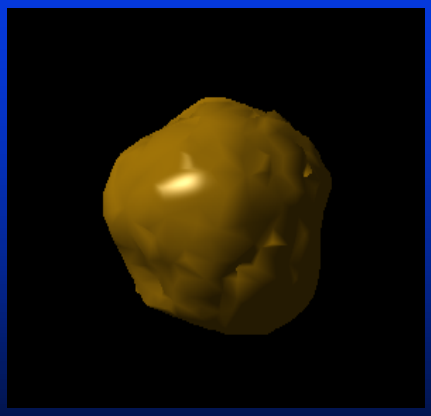
- Preserve three appearance attributes:
  - Surface Position
  - Surface Curvature
  - Material Color
- Each may require different sampling

SIGGRAPH 2001

Normals Undersampled

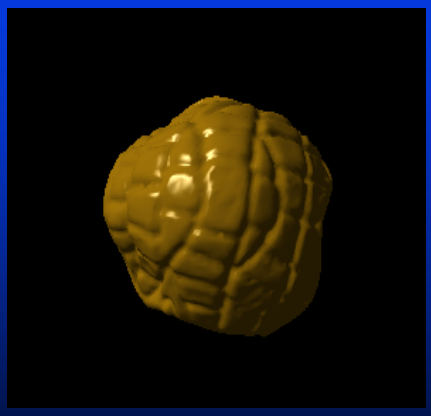


13,433 triangles

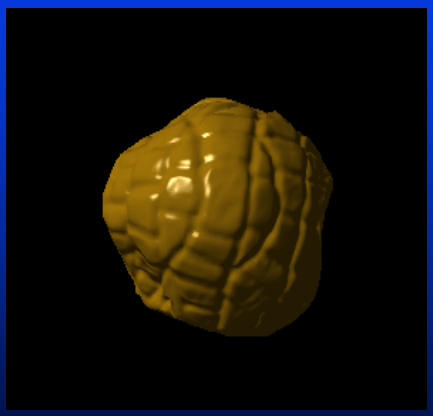


1,749 triangles
10 pixels of surface deviation

Normals Properly Sampled

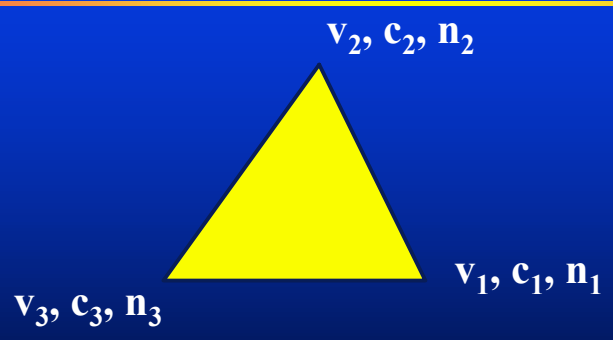


13,433 triangles



1,749 triangles,
10 pixels of deviation

Traditional Polygonal Representation

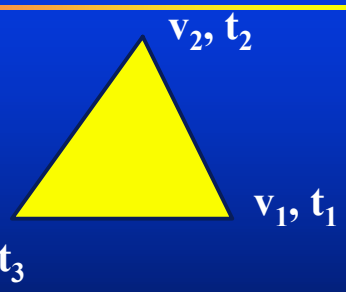


v = vertex coordinate = (x,y,z)
c = color = (r,g,b)
n = normal = (n_x,n_y,n_z)

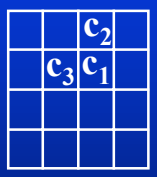
Traditional Simplification

- Filters surface position, colors, and normals
- Must filter all three *equally*

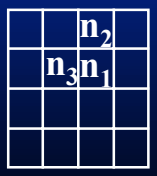
Decoupled Representation



texture map



normal map



v = vertex coordinate = (x,y,z)
t = texture coordinate = (u,v)
c = color = (r,g,b)
n = normal vector = (n_x,n_y,n_z)

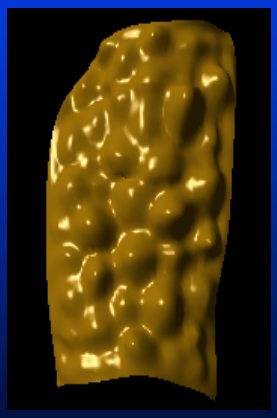
SIGGRAPH 2001

Decoupled Approach

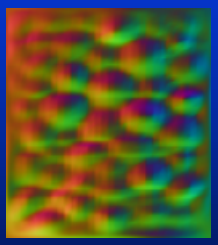
- Simplification filters surface position and texture coordinates
- Color and normal attributes filtered per-pixel (mip-mapping, etc.)

SIGGRAPH 2001

Sample Normal Map



polygonal surface patch



normal map

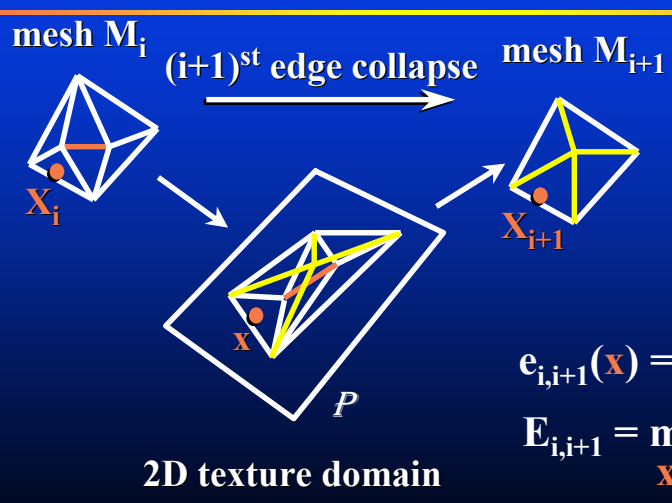
Normal Map vs. Bump Map

- Normal map
  - Absolute normals in object space
  - Constant as object is simplified
    - Same normal map okay for all LODs
- Bump map
  - Perturbations of triangle normal
  - Changes as object is simplified
    - Need different bump map for each LOD

Texture Deviation Metric

- Distance between *corresponding* 3D points
  - Same 2D texture coordinates
  - Projects at run time to 2D pixel deviation
- Intuitive error tolerance specification
  - Pixels of deviation for both surface position and texture error!

Point Correspondence



$$e_{i,i+1}(x) = \| X_{i+1} - X_i \|$$

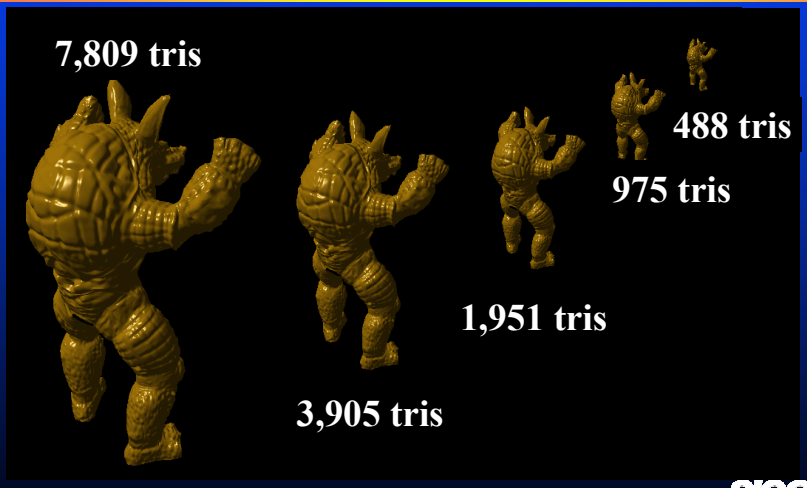
$$E_{i,i+1} = \max_{x \in P} e_{i,i+1}(x)$$

Hardware Requirements

- Texture and normal (or bump) map capability
  - Bandwidth for attribute map lookups
  - Per-pixel lighting computation
- Demonstrated on PixelFlow
- Possible on commercial hardware
  - Normal map shading by Heidrich 99

SIGGRAPH 2001

APS Level-of-detail Hierarchy



model courtesy of Stanford and Caltech

SIGGRAPH 2001

Video Example

- Cohen, Olano, and Manocha, "Appearance-Preserving Simplification," *Proceedings of SIGGRAPH 98.*

APS Executive Summary

- Colors and normals stored in texture and normal maps
- *Texture deviation* computed using parametric correspondence
- Preserves colors and normals, bounding texture motion in object and screen space

Conclusions

- Variety of approaches to bounding object and attribute space errors
- Screen-space geometric error known
- Screen-space texture error known
- Screen-space color and normal error still have room for improvement
  - Employ bounds on color and normal deviation at run-time
  - Guarantee appearance, but simplify more as objects recede
  - Texture and normal map approach requires parameterization, texture management, and improved shading hardware

SIGGRAPH 2001

References

- Bajaj, Chandrajit and Daniel Schikore. Error-bounded Reduction of Triangle Meshes with Multivariate Data. *SPIE.* vol. 2656. 1996. pp. 34-45.
- Cohen, Jonathan, Dinesh Manocha, and Marc Olano. Simplifying Polygonal Models using Successive Mappings. Proceedings of IEEE Visualization '97. pp. 395-402.
- Cohen, Jonathan, Marc Olano, and Dinesh Manocha. Appearance-Preserving Simplification. Proceedings of ACM SIGGRAPH 98. pp. 115-122.
- Cohen, Jonathan, Amitabh Varshney, Dinesh Manocha, Gregory Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification Envelopes. Proceedings of SIGGRAPH 96. pp. 119-128.
- DeFloriani, Leila, Paola Magillo, and Enrico Puppo. Building and Traversing a Surface at Variable Resolution. Proceedings of IEEE Visualization '97. pp. 103-110.
- Erikson, Carl and Dinesh Manocha. GAPS: General and Automatic Polygonal Simplification. Proceedings of 1999 ACM Symposium on Interactive 3D Graphics. pp. 79-88.

References

- Garland, Michael and Paul Heckbert. Simplifying Surfaces with Color and Texture using Quadric Error Metrics. Proceedings of IEEE Visualization '98. pp. 263-269, 542.
- Garland, Michael and Paul Heckbert. Surface Simplification using Quadric Error Bounds. Proceedings of SIGGRAPH 97. pp. 209-216.
- Guéziec, André. Surface Simplification with Variable Tolerance. Proceedings of Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery (MRCAS '95). pp. 132-139.
- Hoppe, Hugues. Progressive Meshes. Proceedings of SIGGRAPH 96. pp. 99-108.
- Hoppe, Hugues. View-Dependent Refinement of Progressive Meshes. Proceedings of SIGGRAPH 97. pp. 189-198.
- Hoppe, Hugues, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. Proceedings of SIGGRAPH 93. pp. 19-26.
- Hoppe, Hugues H. New Quadric Metric for Simplifying Meshes with Appearance Attributes. Proceedings of IEEE Visualization '99. pp. 59-66.

References

- Klein, Reinhard, Gunther Liebich, and Wolfgang Straßer. Mesh Reduction with Error Control. Proceedings of IEEE Visualization '96.
- Lindstrom, Peter and Greg Turk. Fast and Memory Efficient Polygonal Simplification. Proceedings of IEEE Visualization '98. pp. 279-286, 544.
- Luebke, David and Carl Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. Proceedings of SIGGRAPH 97. pp. 199-208.
- Ronfard, Remi and Jarek Rossignac. Full-range Approximation of Triangulated Polyhedra. Computer Graphics Forum. vol. 15(3). 1996. pp. 67-76 and 462.
- Rossignac, Jarek and Paul Borrel. Multi-Resolution 3D Approximations for Rendering. Modeling in Computer Graphics. Springer-Verlag1993. pp. 455-465.
- Xia, Julie C., Jihad El-Sana, and Amitabh Varshney. Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models. IEEE Transactions on Visualization and Computer Graphics. vol. 3(2). 1997. pp. 171-183.

Advanced Issues in Level of Detail

Visual Perception and Level of Detail

Martin Reddy

SRI International
<reddy@ai.sri.com>

SIGGRAPH 2001

Contents

- Why Care About Perceptual Issues?
- Overview of LOD Selection Criteria
- Example Perceptual LOD Systems
- Perceptual Theory
- Applying the Perceptual Theory
- Conclusions & Future Research

SIGGRAPH 2001

Introduction

- **What are we going to talk about?**
  - Guiding LOD selection with reference to models of visual perception
- **Why do we care about visual perception?**
  - Optimize computational resources
  - Minimize LOD popping effects
  - Develop a principled scheme for selecting LOD
- **Caveats!**
  - Visual perception is hard! Let's do what we can.

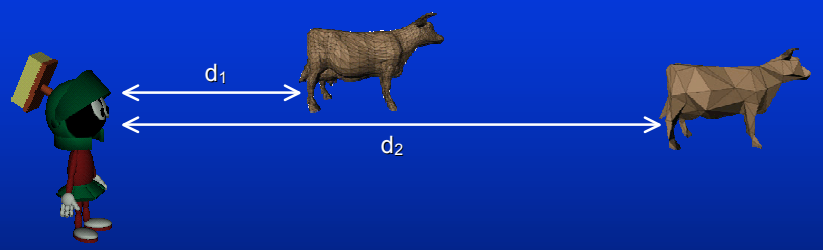
SIGGRAPH 2001

Summary of Level of Detail

- **Primary LOD selection criteria**
  - Distance or Size
  - Velocity
  - Eccentricity
  - Depth of Field
- **Additional LOD constraints**
  - Fixed-frame rate schedulers (reactive or predictive)
  - Hysteresis (switching lag)
  - Priority schemes
  - Alpha-blended transitions (fading regions)
  - Geomorph transitions (morph geometry)

SIGGRAPH 2001

Distance LOD



Select resolution based upon the distance between an element and the viewpoint, i.e. coarser resolution for distant geometry.

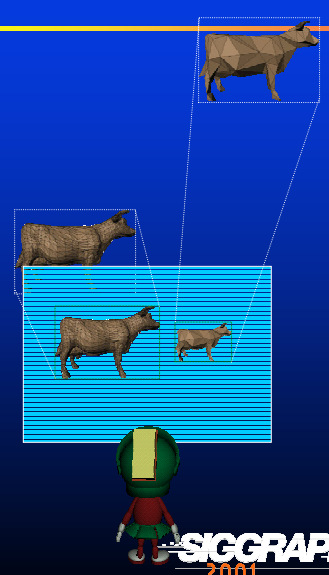
- Simple to calculate (3-D Euclidean distance)
- Scale dependent
- Resolution dependent
- Field of View dependent

Size LOD

Select resolution based upon the projected screen size (or area) of an element. Objects appear smaller as they move further away.

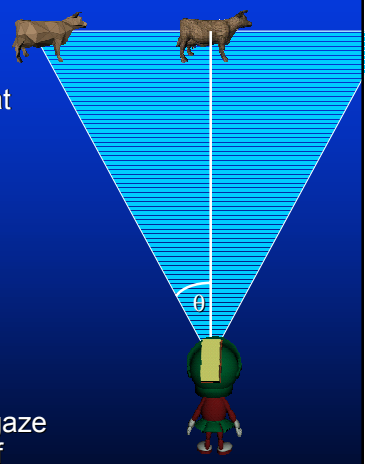
- Requires 3-D → 2-D projection
- Scale invariant
- Resolution invariant
- Field of View invariant

- Bounding spheres or ellipsoids normally used instead of boxes as more efficient to calculate projected extent



Eccentricity LOD

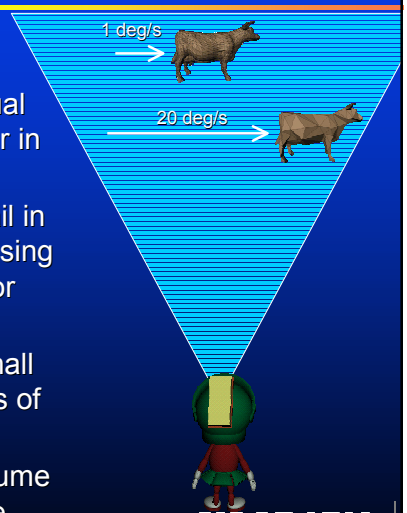
- Resolution is selected based upon the degree to which an element exists in the visual periphery, i.e. display elements that the user is looking at in high resolution.
- Humans can resolve less detail in their peripheral field due to:
  - more retinal photoreceptors (rods/cones) towards fovea
  - retinal and cortical cell receptive field sizes increases linearly with eccentricity
  - 80% of cortical cells devoted to central 10 degrees of vision
- Use eye tracking system to track user's gaze or assume user looking towards center of display



SIGGRAPH 2001

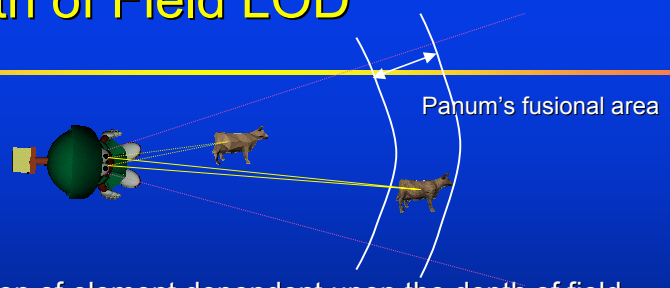
Velocity LOD

- Resolution based upon the angular velocity of an element across the visual field, i.e. faster moving objects appear in lower resolution
- Humans can resolve less spatial detail in objects moving across the retina, causing objects to blur as they move/ rotate, or the user's gaze moves
- It is believed visual information for small features are destroyed by the process of integrating stimulus energy over time
- Without eye tracking technology, assume angular velocity across display device



SIGGRAPH 2001

Depth of Field LOD



- Resolution of element dependent upon the depth of field focus of the user's eyes, i.e. objects out with the fusional area appear in lower detail
- Under binocular vision, both eyes converge on object at certain distance in order to focus retinal image
- Objects in front or behind this fusional area are unfocused, suffering from double images
- Must track both eyes accurately to evaluate convergence distance

Example: Funkhouser & Séquin

SIGGRAPH 1993 - Visualize Complex Virtual Envs.

- Achieve (predictive) fixed frame rate by

  Maximise Σ_s Benefit(*Object*, *Lod*, *Algorithm*)

  Subject to Σ_s Cost(*Object*, *Lod*, *Algorithm*) $\leq$ TargetFrameRate
- Benefit = contribution to model perception:
  - Size: larger objects contribute more to image
  - Accuracy: no of verts/polys, shading model, etc.
  - Priority: account for inherent importance
  - Eccentricity: based on distance from center of display
  - Velocity: ratio of apparent speed to average polygon size
  - Hysteresis: use state from previous frame
- No head/eye tracking.

  No results on perceptual criteria.

Example: Funkhouser & Séquin

"Adaptive Display Algorithm"

Funkhouser and Séquin

(University of California, Berkeley)

SIGGRAPH 2001

Example: Hitchner & McGreevy

SPIE (1993) - NASA VPE Testbed

- Achieve (reactive) fixed frame rate by:
  detail = geometry primitives per unit area
  interest = importance to the user
  - Eccentricity factor: $Fn_E = \gamma_{static}$ / eccentricity
  - Velocity factor: $Fn_V = \gamma_{dynamic}$ / velocity
  - Distance factor: $Fn_D = \beta$ / velocity

  (where γ_{static}, $\gamma_{dynamic}$, and β are arbitrary scaling factors)
- Used a head-mounted display with 6 degree-of-freedom head tracker.

SIGGRAPH 2001

Example: Ohshima et al.

VRAIS (1996) - Gaze-Directed Rendering

- Used 3 criteria to evaluate visual acuity:
  - eccentricity: $g(\Delta\phi) = 1 - \Delta\phi/c_2$ $(0 \le \Delta\phi \le c_2$; else 0)
  - velocity: $f(\theta) = \exp(-(\theta-\alpha)/c_1)$ $(\alpha < \theta$; else 1)
  - depth of field: $h(\Delta\varphi) = \exp(-(\Delta\varphi-b)/c_3)$ $(b < \Delta\varphi$; else 1)

    arbitrary scaling factors set as follows:
    c_1 = 6.2 deg, c_2 = 180 deg, c_3 = 0.62 deg, b = 0 deg.
    α = visual angle occupied by object
- Saccadic suppression:
  - skip rendering when gaze velocity > 180 deg/s
- Used head tracker as a substitute for eye tracker. Used 60 deg projection screen.

SIGGRAPH 2001

Example: Ohshima et al.

"Gaze-directed Adaptive Rendering"

Ohshima, Yamamoto, and Tamura

(Canon Inc.)

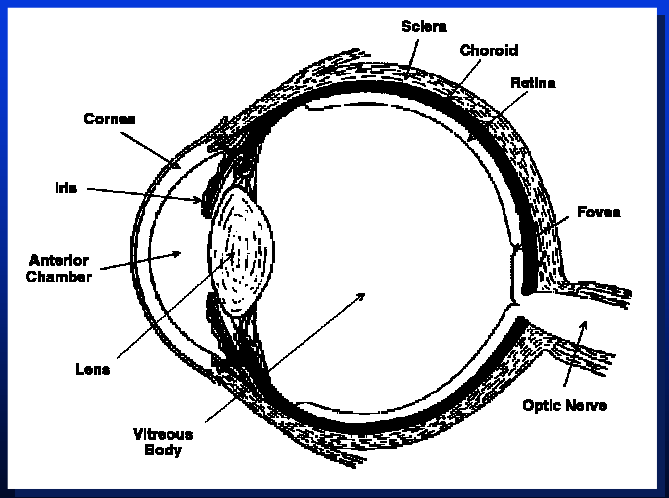
SIGGRAPH 2001

Example: Lindstrom & Turk

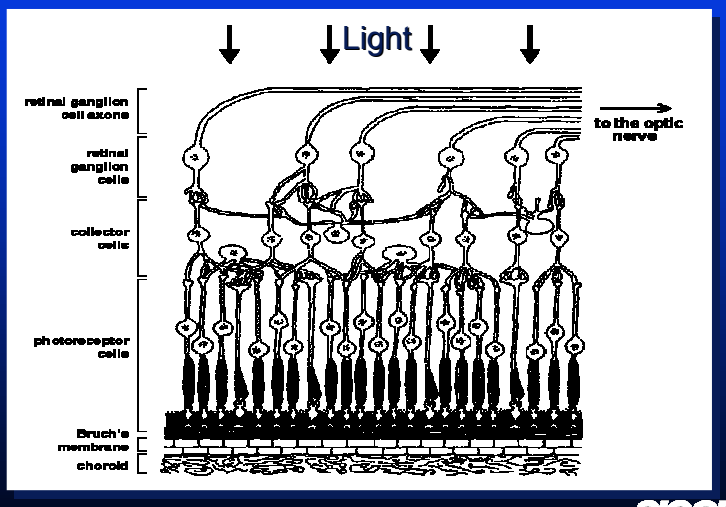
ACM Trans. (2000) - Image-Driven Simplification

- Render each LOD off-screen and analyze images to decide which parts of the model to simplify
  - Guides simplification based upon the visual effect of the reduction rather than some geometric metric
  - Uses a sphere of cameras to capture multiple viewpoints
  - Deals with surface properties and textures
  - Uses RMS error of luminances to compute image distances (fast but not perceptually based)
  - Not real-time (several secs to mins or hours)

The Eye



The Retina



SIGGRAPH 2001

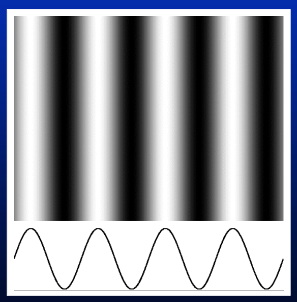
The Limits of Vision

- **Visual acuity**
  - Retina can resolve detail of around 0.5 min of arc
  - 130 million photoreceptors / 1 million ganglion cells
- **Peripheral Vision**
  - Highest sensitivity to spatial detail at fovea (the central 4 to 5 degrees of vision)
  - 35-fold reduction from fovea → periphery
- **Motion Sensitivity**
  - Eye less sensitive to detail moving across retina
  - Fast moving objects become “blurred”

SIGGRAPH 2001

Measuring Limits of Vision

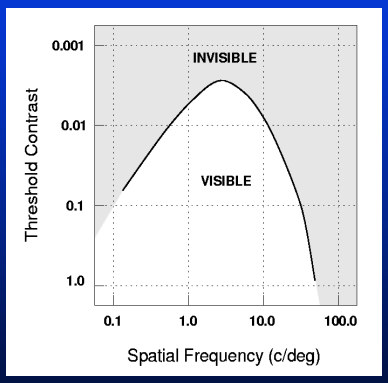
- "Contrast Grating" used to analyze contrast sensitivity . Can vary:
  - Spatial frequency (bar spacing) - cycles per deg (c/deg)
  - Contrast (amplitude)
  - Orientation
  - Velocity
  - Eccentricity



SIGGRAPH 2001

Modeling Limits of Vision

- Results of Contrast Grating tests can be modeled with a Contrast Sensitivity Function
- CSF defines the bandwidth of vision



SIGGRAPH 2001

Velocity CSF

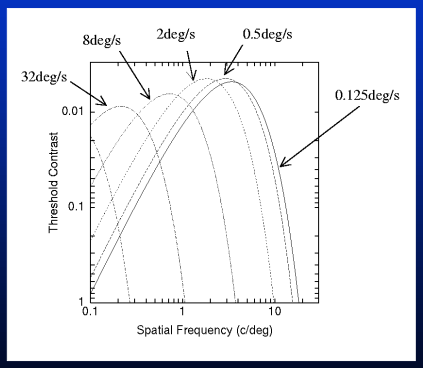
- Kelly (1979) developed an equation to predict the CSF for various stimulus velocities

$$G(\alpha,v) = [250.1+299.3\,|\log_{10}(v/3)|^3] \times v\alpha^2 10^{-5.5\alpha(v+2)/45.9}$$

where,

α = spatial frequency (c/deg)

v = velocity (deg/s)



SIGGRAPH 2001

Eccentricity CSF

- Rovamo & Virsu (1979) developed equations to model the decline of sensitivity with eccentricity for the 4 principal half-meridians of the retina

Nasal: $Mn(E) = 1 / (1+0.33E+0.00007E^3)$

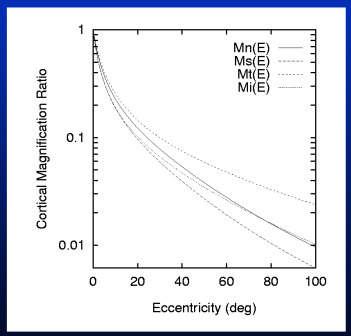
Superior: $Ms(E) = 1 / (1+0.42E+0.00012E^3)$

Temporal: $Mt(E) = 1 / (1+0.29E+0.000012E^3)$

Inferior: $Mi(E) = 1 / (1+0.42E+0.000055E^3)$

where,

E = eccentricity (deg)



SIGGRAPH 2001

Other CSF Factors

- Background illumination
  - Contrast sensitivity degrades in dim conditions
- Display Device Settings
  - Brightness, contrast, color, and gamma
- Viewer's level of light adaption
  - Photoreceptor range and pupil dilation controlled by a feedback loop
- Viewer's visual system efficiency
  - e.g., myopia causes light to converge in front of retina
- Viewer's age
  - Contrast sensitivity less developed in infants & declines with old age

SIGGRAPH 2001

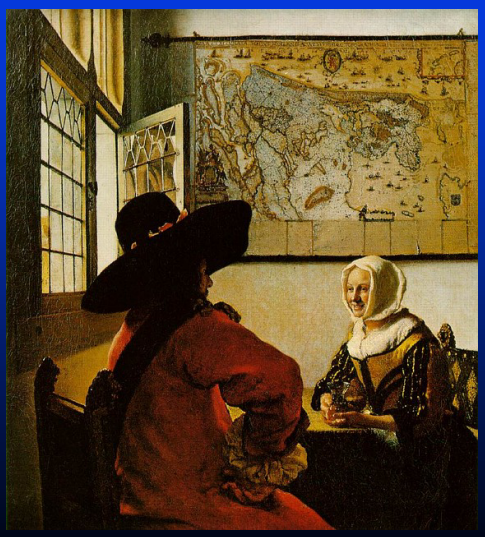
Other CSF Factors (continued)

- Viewer's emotional state
  - Affects dilation of pupils: smaller pupil = less light = drop in visual acuity
- Auditory Stimuli?
  - Recent Nature paper shows visual perception affected by a adding an audible beep during task

Therefore, perceptual data are normally based upon a "Standard Observer", assuming ideal environmental and viewer conditions.

SIGGRAPH 2001

Visual Perception Software



Vermeer

"Officer and Laughing Girl", 1658-60

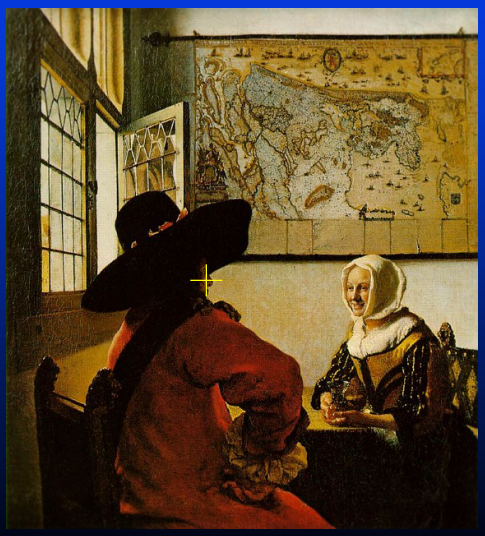
120 x 135 degrees FOV

No eccentricity blurring

No velocity blurring

SIGGRAPH 2001

Visual Perception Software



Vermeer

"Officer and Laughing Girl", 1658-60

120 x 135 degrees FOV

Eccentricity blurring

No velocity blurring

SIGGRAPH 2001

Visual Perception Software

Vermeer

"Officer and Laughing Girl", 1658-60

120 x 135 degrees FOV

Eccentricity blurring

Velocity = 60 deg/s

SIGGRAPH 2001

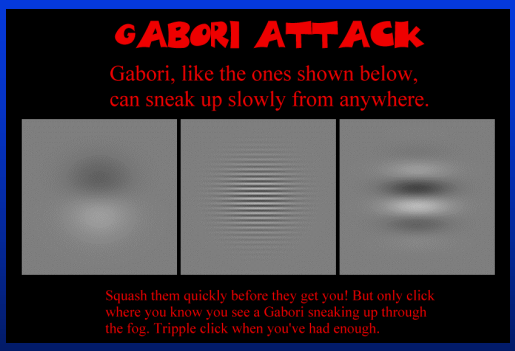
Visual Perception Software

PERCEPT + SIMULATING THE SENSITIVITY OF THE HUMAN VISUAL SYSTEM

- Visualize the limits of vision in an intuitive manner
- Calculate highest spatial frequency of each pixel given its eccentricity and a constant user-specified velocity. Then blur this pixel using an equivalently-sized Gaussian filter
- Based on Kelly, Rovamo & Virsu models
- Open Source, C code (GPL)
- **http://www.ai.sri.com/~reddy/percept/**

SIGGRAPH 2001

Visual Perception Software

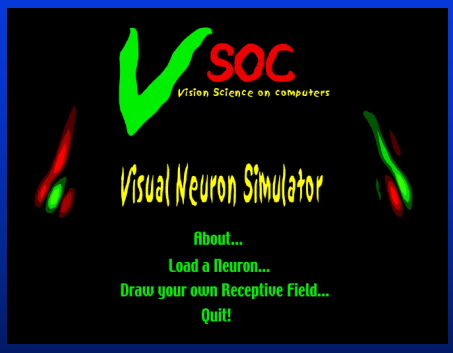


A program for testing contrast sensitivity functions that is presented as a simple video game. From Berkeley VSOC.

http://vsoc.berkeley.edu/vsoc/

SIGGRAPH 2001

Visual Perception Software



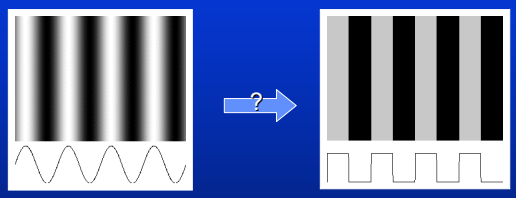
A real-time visual neuron simulator. Explore the receptive fields of artificial neurons. From Berkeley VSOC.

http://vsoc.berkeley.edu/vsoc/

SIGGRAPH 2001

The CSF & Computer Graphics

- Sine v Square Waveform

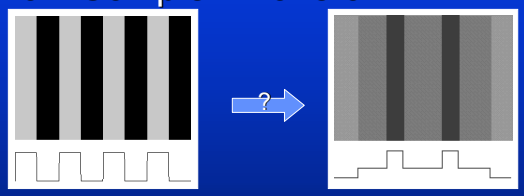


  – Above the peak frequency, the amplitude of square wave CSF is largely determined by the fundamental sine wave
  – The limit of vision is the same in each case

SIGGRAPH 2001

The CSF & Computer Graphics

- Harmonic v Complex Waveform

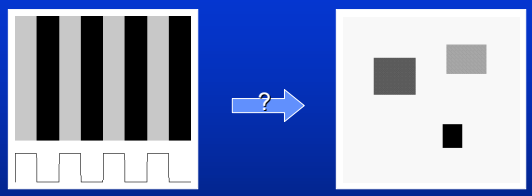


  – Visibility of complex grating is characterized by the independent contributions from each harmonic component
  – below-threshold high-frequency components can be removed without perceivable change

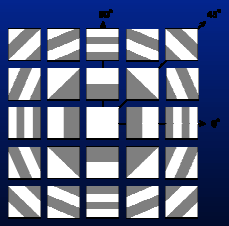
SIGGRAPH 2001

The CSF & Computer Graphics

- 2-D v 1-D Waveform



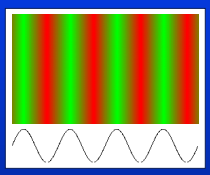
  – Introduce an orientation parameter to describe 2-D features, e.g. (2 c/deg, 90 deg)

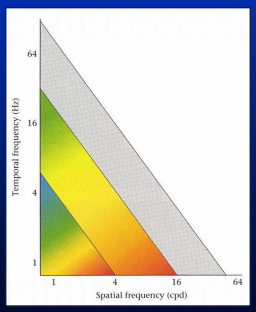


SIGGRAPH 2001

The CSF and Computer Graphics

- Color v Greyscale Gratings
  - Luminance channel more effective than chromatic channels for
    - Form detection
    - Motion
    - Stereoscopic depth
  - For example
    - Luminance upper resolution = 60 c/deg
    - Red/Green upper resolution = 12 c/deg
  - Therefore, use (simpler) achromatic CSF threshold models rather than chromatic ones



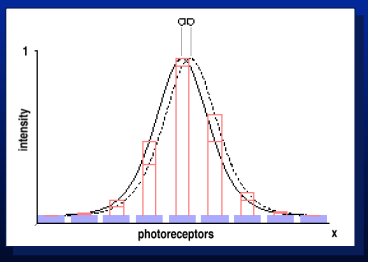


SIGGRAPH 2001

What about hyperacuity?

- Hyperacuity describes the paradox that certain stimuli can be perceived that are smaller than the size of a single photo-receptor cell
- Photoreceptors subtend 25-30 sec of arc (= 60 c/deg)
- But it is possible to discriminate the non co-linearity of two thick abutting lines to a resolution of 2-5 sec of arc (Vernier Acuity)

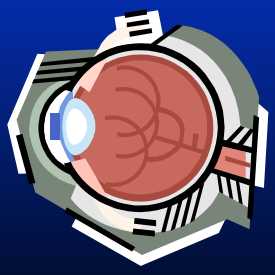
Due to differences in mean distribution of light sampled over a number of photoreceptors
Degrades markedly with ecc.
We are concerned with detection not discrimination though



SIGGRAPH 2001

What about saccades?

- A saccade is a rapid reflex movement of the eye to fixate a target onto the fovea (from French *saccader*, "to jerk")
- We do not appear to perceive detail during a saccade
- Saccades can occur at velocities of up to 800 deg/s
- Duration can be many milliseconds:
  - duration (ms) = 20 + angularDist * 2
  - e.g. 10 deg saccade lasts ~40 ms



Ohshima et al.'s (1996) system suspended rendering at gaze velocities > 180 deg/s

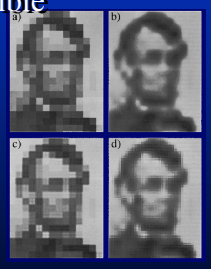
SIGGRAPH 2001

What about visual masking?

The presence of one visual pattern can affect the visibility of another pattern

e.g. a large adjacent stimulus (in time or space) can cause the threshold of a smaller stimulus to be increased - the smaller stimulus needs to be more intense for it to be visible

- Also, the detection threshold of a stimulus varies inversely as a function of its distance from an edge (larger errors can be tolerated around an edge)
- Ferwerda et al. (1997) developed a visual masking model for computer graphics



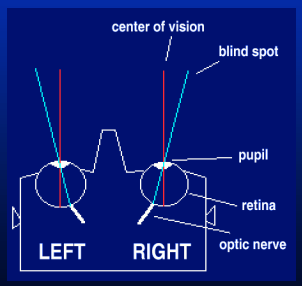
Harmon & Julesz (1973)

SIGGRAPH 2001

What about the blind spot?

- The blind spot is caused by the area of the retina where all axons of the retinal ganglion cells meet to form the optic nerve. There are no photoreceptors in this region.
- Can we reduce detail if an object falls onto the blind spot?
- Size of blind spot = 5 - 7 deg
- Located at ~17 deg eccentricity

We have 2 eyes! At least one eye will always detect the object, so don't bother with the blind spot



SIGGRAPH 2001

Applying Perceptual Metrics

- **Monitor object statistics**
  Calculate the projected size (deg), velocity (deg/s), and eccentricity (deg) of any part of an object. (Relative to the display, the user's head, or the user's gaze.)
- **Measure perceived detail in imagery**
  Describe the perceived spatial detail of any part of an object in terms of its spatial frequencies (c/deg)
- **Model user's visual acuity**
  Use mathematical model to estimate the contrast sensitivity of the user under various conditions (velocity, eccentricity, etc.)

SIGGRAPH 2001

Perceptual LOD Systems

- Reddy (1997, 2001)
  - Static and view-dependent LOD Systems
  - Calculate spatial frequency profiles for each LOD off-line using a sphere of cameras
  - Ignore contrast (assume worst-case scenario)
- Luebke (2000)
  - Dynamic (view-dependent) LOD System
  - Calculate spatial frequency induced by folding a node in real-time
  - Incorporate contrast and silhouette conditions

SIGGRAPH 2001

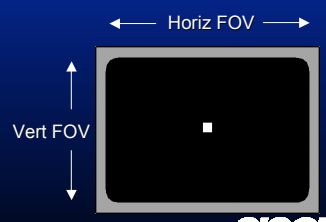
Calculating c/deg - Reddy

Isolate visual feature

Extract relative fundamental spatial frequencies (c/pixel)

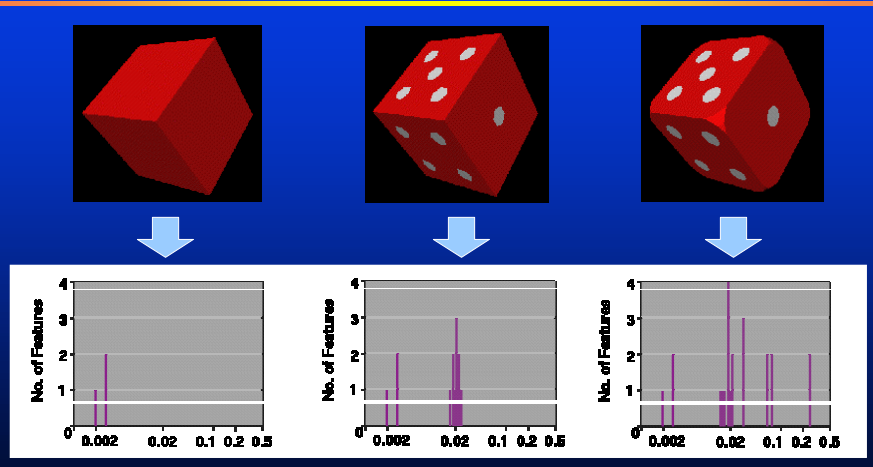
1 feature = 1/2 contrast cycle, so 4 pixels = 1/8 c/pixel

Apply Field of View scaling (c/deg)



SIGGRAPH 2001

Calculating c/deg - Reddy



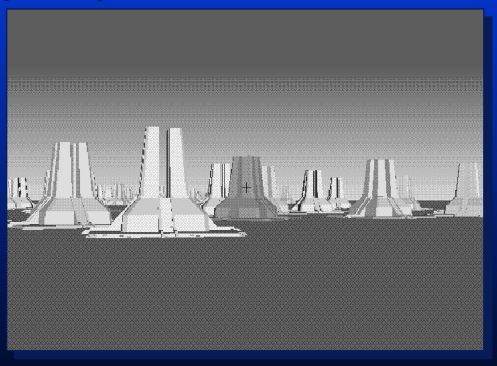
x-axis = relative spatial frequency (c/pixel). Scale by field of view for c/deg

SIGGRAPH 2001

Sample Results - Reddy

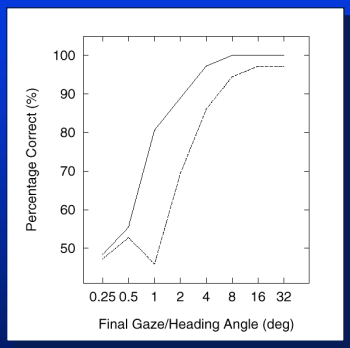
- Passive psychophysical navigation task used to evaluate benefit of perceptual criteria

4 LODs for each object
LOD varied by size, velocity, & eccentricity
User focused on crosshair in center
Desktop configuration with bite bar to constrain head movements

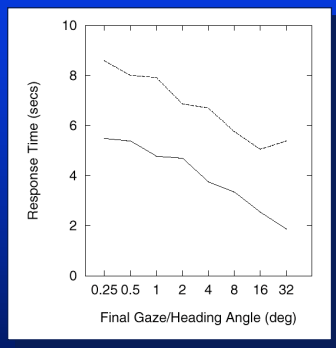


SIGGRAPH 2001

Sample Results - Reddy



User task proficiency increased 2.8 times (at the 95% performance threshold)

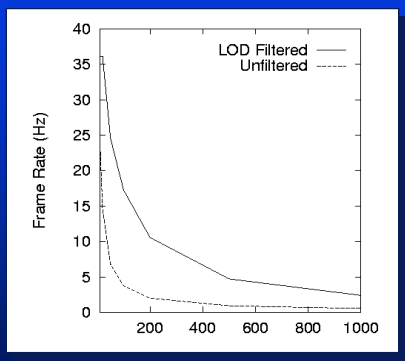


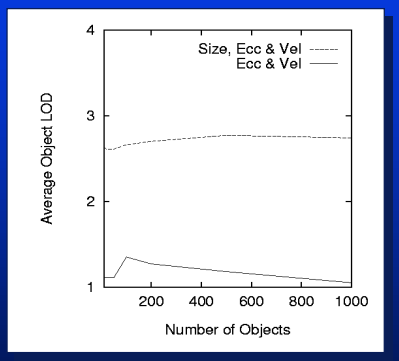
Average response time improved 1.67 times

Solid line = perceptual LOD optimizations. Broken line = no LOD

SIGGRAPH 2001

Sample Results - Reddy

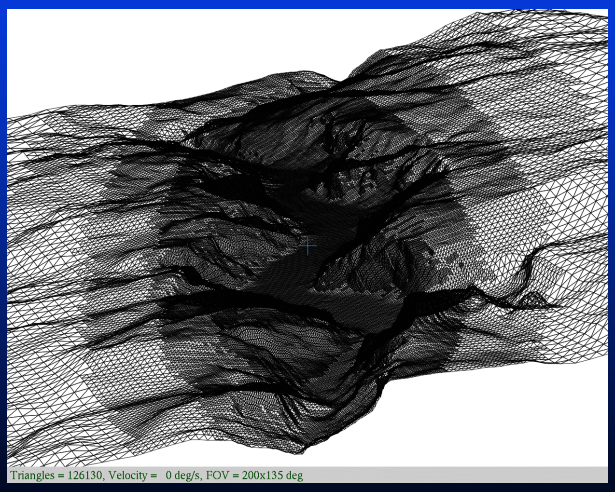




4-5 times improvement in frame rate when using LOD

Size LOD accounts for 90-95% of improvement

View-Dependent Perceptual LOD



Original Model

1,116,720 tris

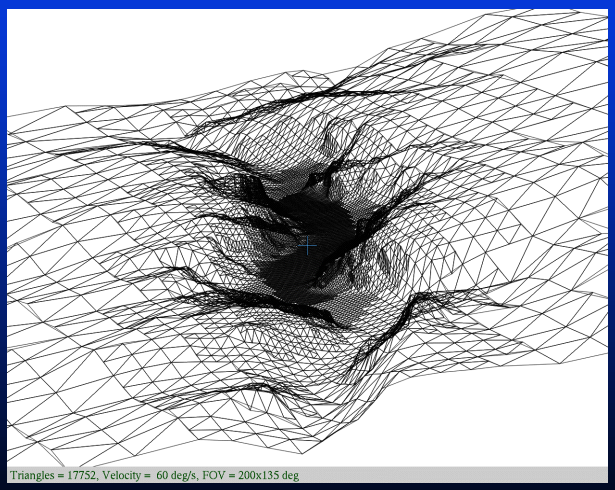
Eccentricity Opt

126,130 tris

Field of View

200x135 deg

View-Dependent Perceptual LOD



Original Model

1,116,720 tris

Ecc + 60 deg/s

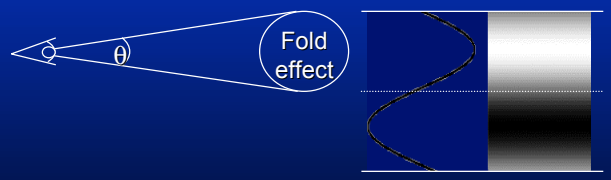
17,752 tris

Field of View

200x135 deg

Calculating c/deg - Luebke

- Use bounding spheres to model extent of change caused by the folding a vertex in a hierarchical vertex tree
- Compute spatial frequency of change based upon angular projection of sphere (θ^o fold extent gives frequency of one cycle per 2 θ^o)



Contrast calculated by comparing intensities of all vertices in original and simplified surface (if silhouette edge, compare against brightest and darkest intensities in the scene).

Movie - Luebke

"Perceptually Driven Simplification"

Luebke

(University of Virginia)

SIGGRAPH 2001

Enabling Technologies

- Head tracking
  - Commonly used in virtual reality systems
  - Fast head movement = high angular velocity
- Eye Tracking
  - Required for true perceptual LOD optimizations
  - Generally cumbersome and suffers from problems of lag, drift, resolution, etc.
  - But perhaps head tracking is enough:
    - Resting gaze orientation ~= head orientation
    - Most saccades occur with 15 deg of gaze point

SIGGRAPH 2001

Conclusions

- Perceptual LOD more applicable to immersive systems:
  - head/eye tracking better than center of display
  - extra benefit when user moves head/gaze
  - one display per person (multiple viewers possible though)
- Velocity and eccentricity optimizations should be used in conjunction for maximum benefit
- Distance / Size LOD offers the most advantage (e.g. around 90-95% in non-immersive system).
- View-dependent LOD system best for maximum resolution reduction
- Supporting Velocity optimizations can give big wins in a dynamic environment
- Less reduction than a non-perceptual system, but can do perceptually linear fixed frame rate also.

SIGGRAPH 2001

Further Research Areas

- Need to think about temporal effects of switching between two different representations (flicker frequency). The peripheral field is highly sensitive to flicker. Does this even matter?
- Need better perceptual metrics to assess the spatial frequency and contrast of a computer-generated image or of a polygonal model.

Need more results on the benefit of using various perceptual criteria under different viewing conditions and display devices (e.g. immersive systems).

SIGGRAPH 2001

Web Resources

Slides for this presentation
http://www.ai.sri.com/~reddy/talks/s2001/

Perceptually Modulated LOD Thesis
http://www.ai.sri.com/~reddy/thesis/

David Luebke's VDS Library
http://vdslib.virginia.edu/

Mike Krus' LOD Resources
http://www.multimania.com/krus/CG/LODS/

Lee Bull (Pip)'s LOD World
http://www.cs.ucl.ac.uk/staff/P.Bull/lod/lod.html

Multiresolution Modeling
http://www.cs.cmu.edu/~garland/multires/

SIGGRAPH 2001

LOD Bibliography

- Funkhouser, T. A. and Séquin, C. H. (1993). "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments". SIGGRAPH '93 Proceedings, 27: 247-254.
- Hitchner, L. E. and McGreevy, M. W. (1993). "Methods for User-Based Reduction of Model Complexity for Virtual Planetary Exploration". Proceedings of the SPIE, 1913: 622-636.
- Ohshima, T., Yamamoto, H. and Tamura, H. (1996). "Gaze-Directed Adaptive Rendering for Interacting with Virtual Space". Proceedings of VRAIS '96, pp. 103-110.
- Koller, D., Lindstrom, P., Ribarsky, W., Hodges, L., Faust, N. and Turner, G. (1995). "Virtual GIS: A Real-Time 3D Geographics Information System". Proceedings of Visualization '95, pp 94-100.
- Astheimer, P. and Pöche, M-L. (1994). "Level-of-Detail Generation and its Application to Virtual Reality". Proceedings of VRST '94, pp. 299-309.
- Watson, B. A., Walker, N., Hodges, L. and Reddy, M. (1997). "An Evaluation of Level of Detail Degradation in Head-Mounted Display Peripheries". Presence: Teleoperators and Virtual Enviornments, 6(6).
- Lindstrom, P. and Turk, G. (2000). "Image-Driven Simplification". ACM Transactions on Graphics, 19(3): 204-241.
- Reddy, M. (1997). "Perceptually Modulated Level of Detail for Virtual Environments". PhD Thesis CST-134-97, University of Edinburgh, UK.

SIGGRAPH 2001

Perception Bibliography

- Kelly, D. H. (1979). “Motion and Vision II: Stabilized Spatio-Temporal Threshold Surface”. Journal of the Optical Society of America, 69(10): 1340-1349
- Rovamo, J. and Virsu, V. (1979). “An Estimation and Application of the Human Cortical Magnification Factor”, Experimental Brain Research, 37: 495-510.
- Nakayama, K. (1990). “Properties of Early Motion Processing: Implications for the Sensing of Egomotion”. In The Perception and Control of Self Motion, Lawrence Erlbaum, Hillsdale, NJ, pp. 69-80.
- Campbell, F. W. and Robson, J. G. (1968). “Application of Fourier Analysis to the Visibility of Gratings”. Journal of Physiology, 197: 551-566.
- Burr, D. C. and Ross, J. (1982). “Contrast Sensitivity at High Velocities”. Vision Research, 22: 479-484.
- Koenderink, J. J., Bouman, M. A., de Mesquita, A. E. B. and Slappendel, S. (1978). “Perimetry of Contrast Detection Thresholds of Moving Spatial Sine Wave Patterns”. Journal of the Optical Society of America, 68(6): 845-854.
- Lamming, D. (1991). “On the Limits of Visual Detection”. In Vision and Visual Dysfunction: Limits of Vision, vol 5. MacMillan Press Ltd., chapter 3, pp 15-22.
- Tyler, C. W. (1985). “Analysis of Visual Modulation Sensitivity. II. Peripheral Retina and the Role of Photoreceptor Dimensions”. Journal of the Optical Society of America, A2(3): 393-398.
- Wandel, B. A. (1995). “Foundations of Vision”. Sinauer Associates Inc., Sunderland, MA.

SIGGRAPH 2001

Temporal and Visual Fidelity

Benjamin Watson

Dept. Computer Science
Northwestern University

watson@cs.nwu.edu

SIGGRAPH 2001

The Visual/Temporal Tradeoff

Dynamic LOD is striking a compromise:

- *Visual*: polys, textures, lighting…
- *Temporal*: frame rate, delay…

Most have emphasized visual

Here we examine both in some detail

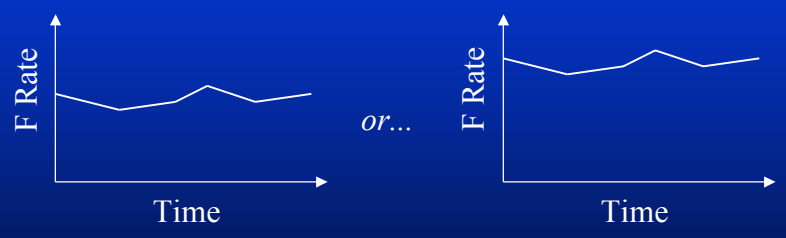
SIGGRAPH 2001

Outline

- Temporal Fidelity
  - Motivating questions
  - Temporal Basics
  - Previous Research
  - Answers and Implications
- Visual Fidelity (Quickly)
- Future Directions

SIGGRAPH 2001

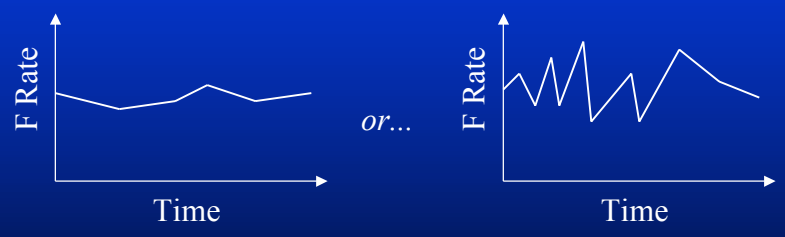
Questions: A Good Mean?



Frame rate: what's "good enough?"

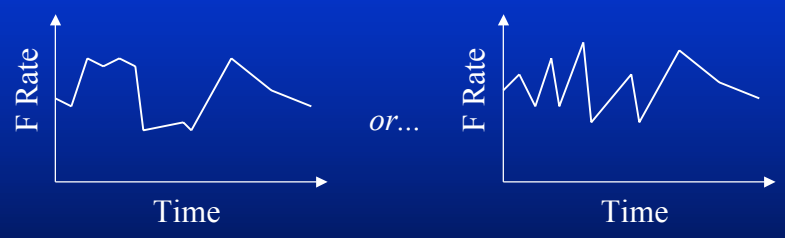
SIGGRAPH 2001

Questions: How Constant?



Frame rate: how much change?

Questions: Pattern Effects?



Frame rate: effects of frequency?

Other Questions

Effects of frame rate vs. delay?

display continuity vs. age

How do effects vary by task?

e.g. placing vs. navigating

SIGGRAPH 2001

Basics: Frame Rate

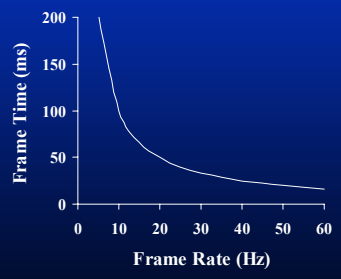
Def: displayed samples/sec

– *Not* constant, that takes work

Frame time: ms/frame

– Nonlinear relationship

– Better predictor human performance: linear delay relationship

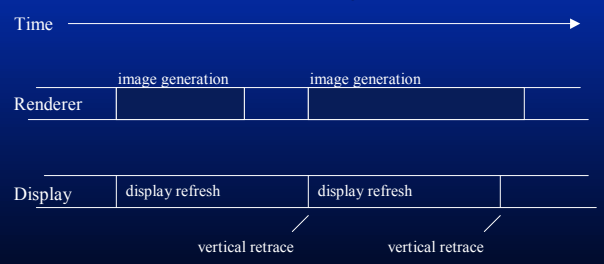


SIGGRAPH 2001

Basics: Frame Rate

Refresh rate: display refresh/sec

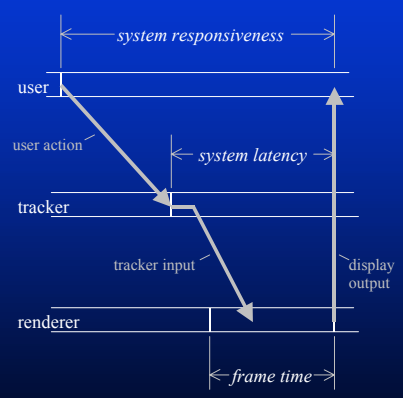
- One frame time = a multiple of refresh time
- Mean frame time may not be



Basics: System Latency

Def: age of displayed sample

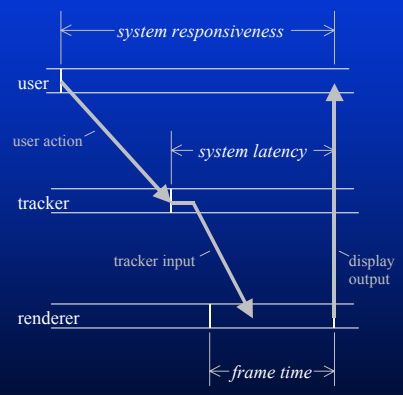
- some of frame time, plus input collection time
- Also varies



Basics: System Responsiveness

Def (SR): delay from input to display

- system latency, plus delay between event and sample



Basics: System Responsiveness

Only one sample used per frame

So delay between event and sample:

- Mean: half frame time
- Variation: random, range 1 frame time

So frame time has great effect on SR

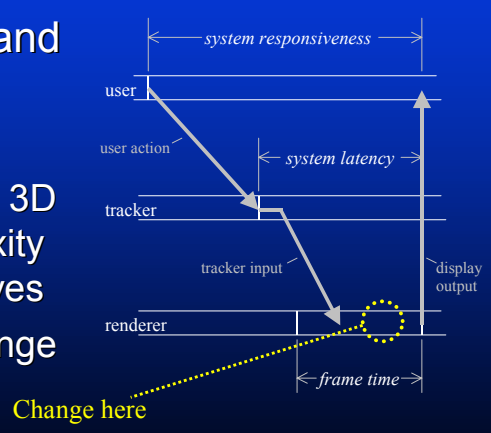
Not So Basic: Complex Systems

VR systems may have
- More than one input device
- More than one output device

For each I/O link
- Different latency, SR

Temporal Fidelity Control: *Frame-Latency*

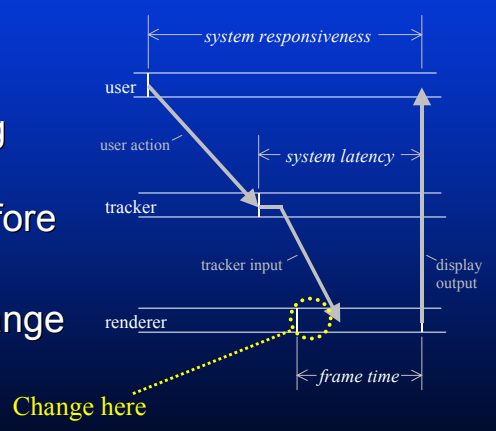
Both frame time and latency change (FL)
- e.g. by varying 3D model complexity after input arrives
- SR += 3/2 change



Temporal Fidelity Control: *Frame-Only*

Only frame time changes (FO)

- e.g. by varying simulation complexity before input arrives
- SR += 1/2 change

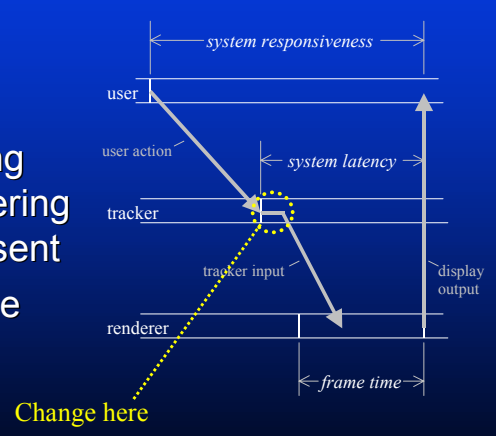


SIGGRAPH 2001

Temporal Fidelity Control: *Latency-Only*

Only latency changes (LO)

- e.g. by varying degree of filtering before input sent
- SR += change



SIGGRAPH 2001

Temporal Fidelity Control: *Summary*

Change Type	Abbr	Ftm?	Ltcy?	SR?
Frame only	FO	1 x		0.5 x
Latency only	LO		1 x	1.0 x
Frame latency	FL	1 x	1 x	1.5 x

Moral: not all delay or speedup is equal

- Differ in frame rate/latency effects
- Differ in SR effects

SIGGRAPH 2001

Outline Update

Temporal Fidelity

- Motivating questions
- Temporal Basics
- Previous Research
- Answers and Implications

Visual Fidelity (Quickly)

Implications

SIGGRAPH 2001

Research: Tasks With Prediction

If tasks require prediction:

- Environment is changing
- Must observe, anticipate, act

Therefore:

- Need samples for prediction: *frame time*
- Account for sample age: *system latency*

SIGGRAPH 2001

Research: Tasks With Feedback

If tasks require feedback:

- Must act, evaluate results, act….

Therefore:

- Must wait for results: *responsiveness*
- Feedback iterations multiply SR effects

SIGGRAPH 2001

Research: Catching Tasks

Catching tasks:

- Visually track, manually intercept
- e.g. shooting, catching moving objects

Prediction important, not feedback

- Expect: At same SR, FL and FO more effect than LO

SIGGRAPH 2001

Research: Catching Tasks

Watson, Richard:

- ~15 Hz “enough” (290 ms)
- Examined only FL effects

Watson:

- SR change 100+ ms significant
- Perceivable patterns had effects

SIGGRAPH 2001

Research: Tracking Tasks

Tracking tasks:

- Manually track a moving object
- Similar to navigation

Both prediction and feedback crucial

- Expect: At same SR, FL and FO slightly more effect than LO

SIGGRAPH 2001

Research: Tracking Tasks

Tharp, Bryson (small studies):

- Compared mean FO, mean LO

Obtained similar results:

- Faster SR always helped users
- Given same SR, LO better than FO
- Difficulty increased SR effects

SIGGRAPH 2001

Research: Placement Tasks

Placement tasks:

- Move to well-known location
- e.g. selection, grasping static objects

Feedback important, not prediction

- Expect: SR is key, whether FL, FO or LO

SIGGRAPH 2001

Research: Placement Tasks

Bryson (small study):

- Given same SR, FO and LO same effects
- More SR always better

Mackenzie & Ware (LO only):

- SR improved down to 25 ms
- Difficulty increased SR effects

SIGGRAPH 2001

Research: Placement Tasks

Ware & Balakrishnan:

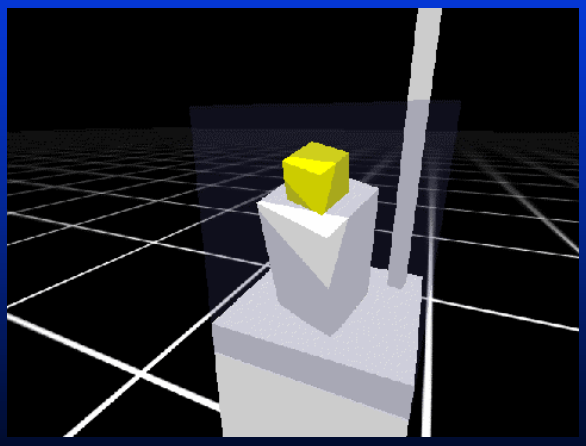
- Given same SR, FL/FO/LO same effects

Watson (FL only):

- Difficulty increased SR effects
- SR change (100+ ms) had effects
- Perceivable patterns had effects

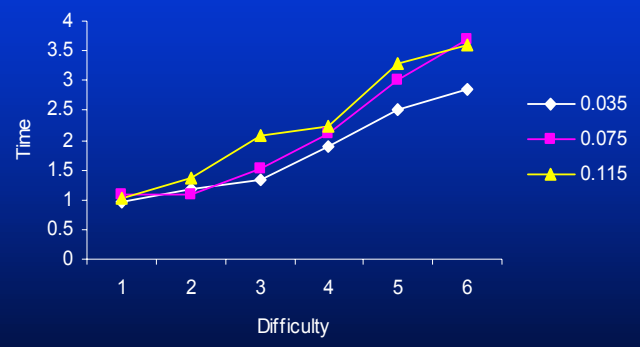
SIGGRAPH 2001

Research Details: Task



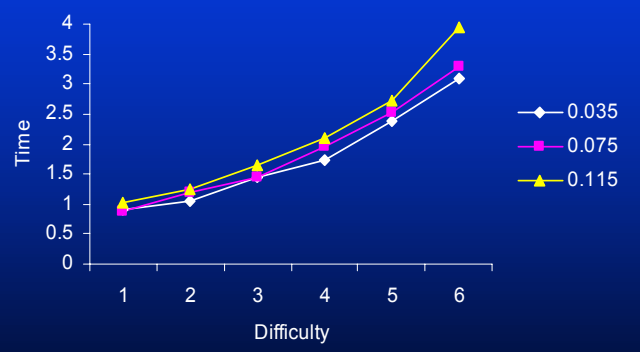
SIGGRAPH 2001

Research Details: Placement



F rate and delay become more important with difficulty

Research Details: Placement



Feedback lessens effects of delay (esp. @ high difficulty)

Answers: A Good Mean?

Predictive tasks:

- "Enough" is quite possible
- This applies to SR and frame time

Feedback tasks:

- Really hard to get enough SR

Answers: How Constant?

Not very:

- Up to 100 ms change okay!

Predictive tasks:

- Constancy impt at low frame rates

Feedback tasks:

- Constancy impt when mean SR good

Answers: Pattern Effects?

Yes, there are effects:

- Only when change is severe
- Only when frequencies are large

More research needed:

- Transient change?
- Asymmetric change?
  - trend: worsening SR is worse than improving SR

SIGGRAPH 2001

Other Answers

Effects of frame rate vs. SR?

- Generally, SR dominates

How do effects vary by task?

- Predictive: frame time
- Feedback: SR

SIGGRAPH 2001

LOD Implications

Improve SR before frame time

Easier to get "enough" frame time

Only minor efforts to control change

User can tolerate quite a bit

SIGGRAPH 2001

LOD Implications

After change, patterns

Control only large scale changes

Avoid repetitive patterns of change

Control should be sensitive to task!

According to predictive, feedback makeup

SIGGRAPH 2001

LOD Implications

If delay (speedup) is necessary (possible):

Prefer FO, LO, and FL, in that order (reverse for speedup)

Control should be sensitive to difficulty

Higher difficulty, better temporal detail

SIGGRAPH 2001

LOD Implications

Feedback can compensate for delay, diff

Improved visual feedback != more polys

Especially in tasks depending on feedback

Visual/temporal detail interactions?

SIGGRAPH 2001

Visual vs. Temporal

Information is the key:

*It can be **spatially** inaccurate (wrong place)*

*Or, **temporally** inaccurate (wrong time)*

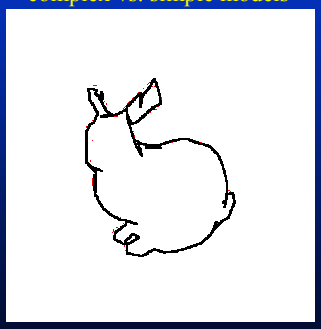
How to compare these inaccuracies?

SIGGRAPH 2001

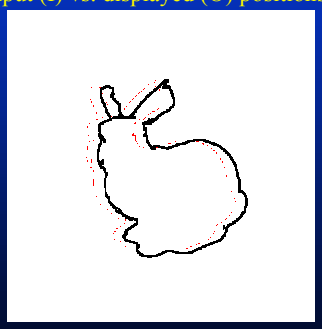
I/O Differencing

Key observation: wrong time = wrong place

complex vs. simple models

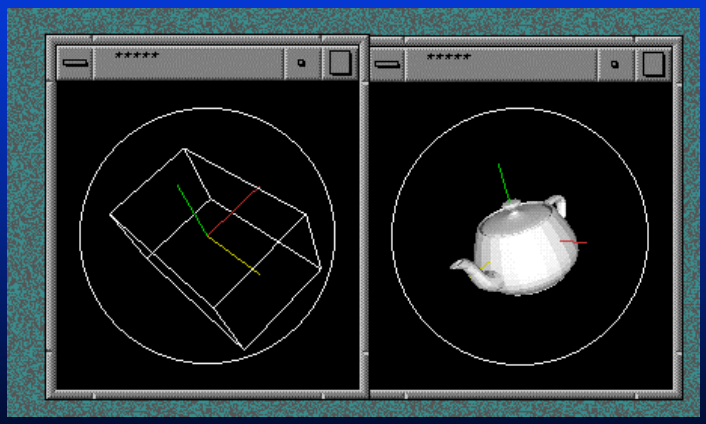


Input (I) vs. displayed (O) positions



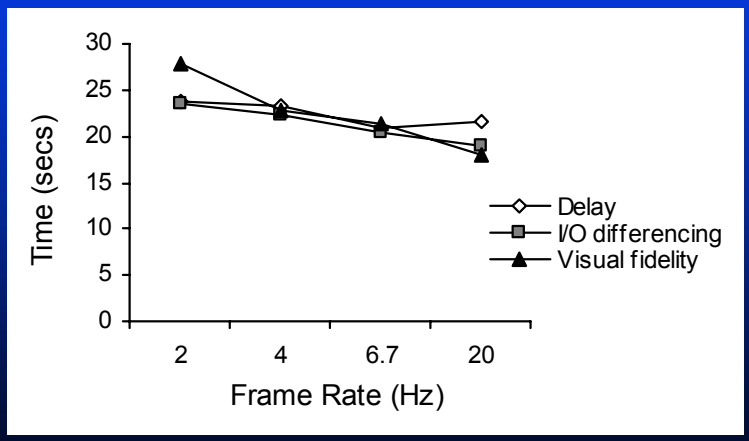
SIGGRAPH 2001

I/O Differencing Task



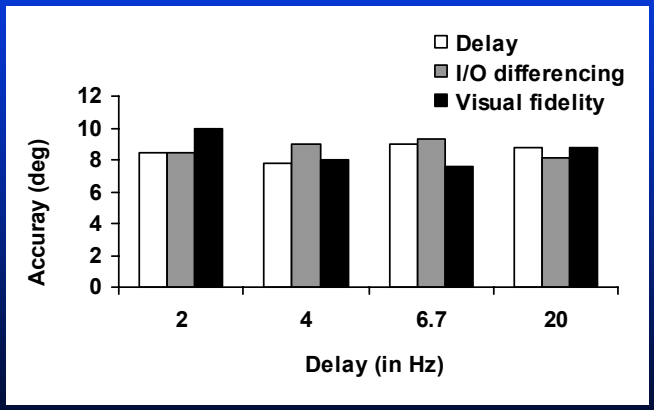
SIGGRAPH 2001

I/O Differencing Results

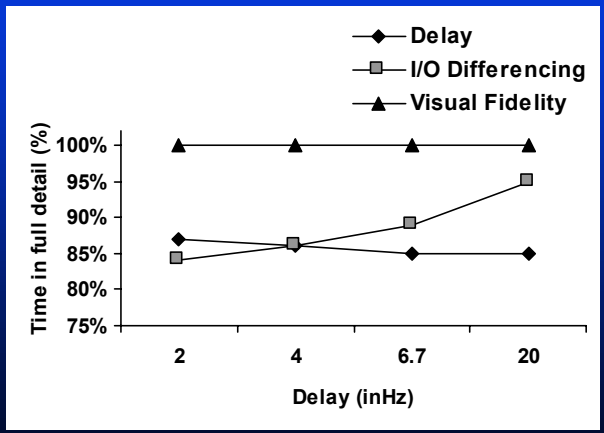


SIGGRAPH 2001

I/O Differencing Results



I/O Differencing Results



Visual Fidelity Control

When controlling visual fidelity:

- How well are we doing?
- How might we do better?

For answers, we need

- Experimental measures of visual fidelity
- Automatic measures of visual fidelity

SIGGRAPH 2001

Experimental Vis Fidelity

Psychophysical experimentation

- Nice internal validity, but
- External validity is a big concern!

We studied 3 higher level measures

- Forced choice
- Ratings
- Naming times

SIGGRAPH 2001

Experimental Vis Fidelity

Our independent variables

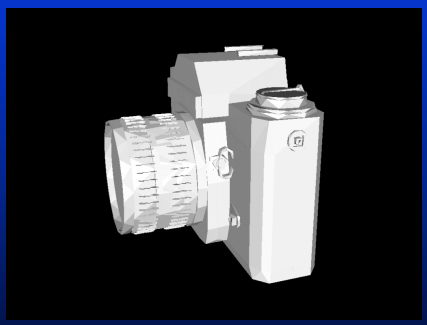
- 2 simplification algorithms: *QSlim, Cluster*
- 3 levels of simplification: *0%, 50%, 80%*
- 2 groups of stimuli: *animals, objects*

36 experimental stimuli

36 subjects

SIGGRAPH 2001

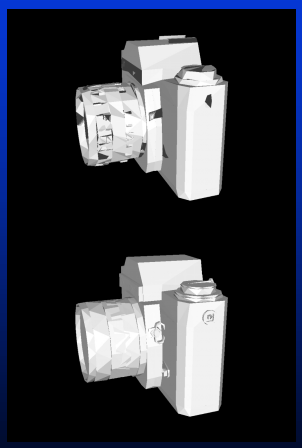
Some Typical Stimuli



Original Models

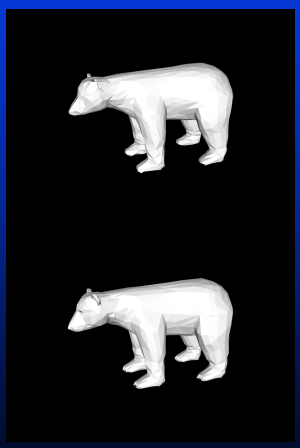
SIGGRAPH 2001

Some Typical Stimuli



clustering

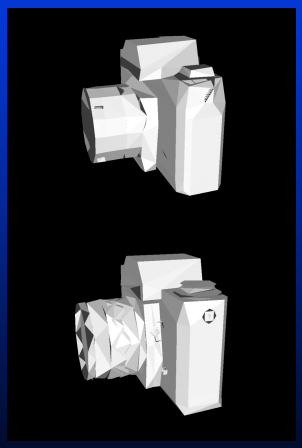
qslim



50% Simplified

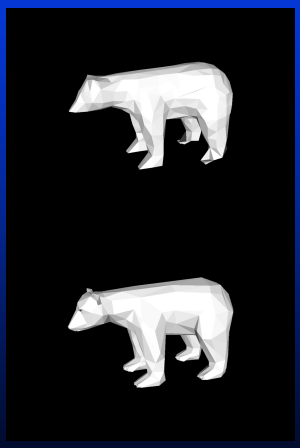
SIGGRAPH 2001

Some Typical Stimuli



clustering

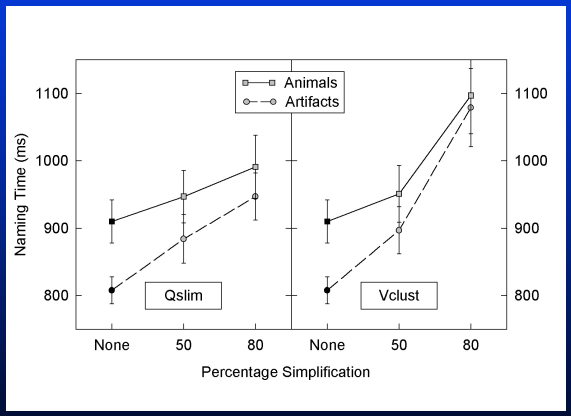
qslim



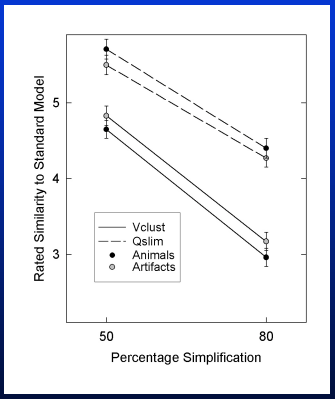
80% Simplified

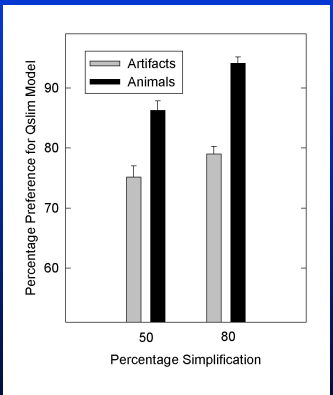
SIGGRAPH 2001

Naming Time Results



Rating & Choice Results





Measure Comparisons

	Simp Alg	Simp Level	Object Type
Naming Time	*QSlim better*	*80% worst*	*Animals harder*
Ratings	*QSlim better*	*80% worst*	
Forced Choice	*QSlim better*	*80%: Q even better*	*Animals: Q even better*

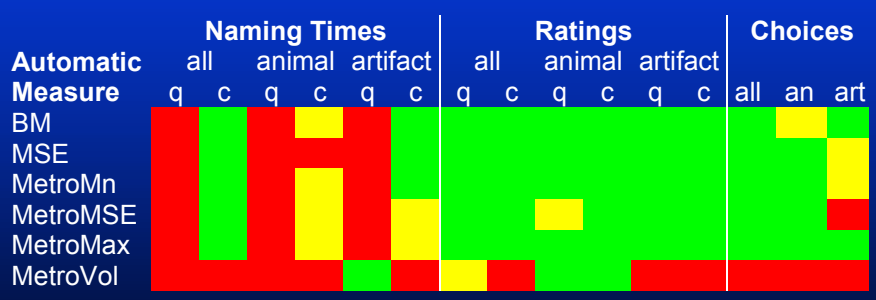
Lessons:

Measures agree with intutition

Measures disagree on object type

SIGGRAPH 2001

Expmt/Auto Correlations



Stat Sig Correlation

Correlation < .2

Stat Insig Corr > .2

SIGGRAPH 2001

Implications, Questions

Some support for:

Measuring fidelity with 3D distance

Quadric metrics (weaker in artifacts)

Questions:

Luebke/Reddy's measures effective?

Modeling naming times: distillation effect?

"Top-down" simplification by object type?

SIGGRAPH 2001

Bibliography

Barfield, W. & Hendrix, C. (1995). The effect of update rate on the sense of presence within virtual environments. *Virtual Reality*, *1*, 1, 3-16.

Bryson, S. (1993). Implementing virtual reality. *SIGGRAPH 1993 Course #43 Notes*, 1.1.1-1.1.5, 1.3.1-1.3.26; 16.1-16.12.

Fitts, P. (1954). The information capacity of the human motor system in controlling the amplitude of movement. *J. Exp. Psychology*, *47*, 6, 381-381.

MacKenzie, S. (1995). Movement time prediction in human-computer interfaces. In Baeker, R., Buxton, B., Grudin, J. & Greenberg, G. (Eds.), *Readings in human-computer interaction* (2nd ed.), 483-493. Los Altos, CA: Kaufmann.

MacKenzie, S. & Ware, C. (1993). Lag as a determinant of human performance in interactive systems. *Proc. INTERCHI Conf. on Human Factors in Computing Systems*, 488-493.

SIGGRAPH 2001

Bibliography

Mine, M. (1993). Characterization of end-to-end delays in head-mounted display systems. University of North Carolina at Chapel Hill, Department of Computer Science. Technical report TR93-001.

Richard, P., Birebent, G., Coiffent, P., Burdea, G., Gomez, D. & Langrana, N. (1996). Effect of frame rate and force feedback on virtual object manipulation. *Presence*, *5*, 1, 95-108.

Sheridan, T. & Ferrell, W. (1963). Remote manipulative control with transmission delay. *IEEE Trans. Human Factors in Electronics*, *4*, 25-29.

Tharp, G., Liu, A., French, L., & Stark, L. (1992). Timing considerations of helmet mounted display performance. In *Human Vision, Visual Processing, and Digital Displays III*. SPIE Vol. 1666, 570-576.

Ware, C. & Balakrishnan, R. (1994). Reaching for objects in VR displays: lag and frame rate. *ACM Trans. Computer-Human Interaction*, *1*, 4, 331-357.

SIGGRAPH 2001

Bibliography

Watson, B. (1997). Level of detail management. Dissertation, College of Computing, Georgia Inst. Technology.

Watson, B., Spaulding, V., Walker, N., & Ribarsky, W.R. (1997). Evaluation of the effects of frame time variation on VR task performance. *IEEE Virtual Reality Annual Symposium* (VRAIS), 38-44.

Watson, B., Walker, N., Hodges, L. & Worden, A. (1997). Managing level of detail through peripheral degradation: effects on search performance with a head-mounted display. *ACM Trans. Computer-Human Interaction*.

Watson, B., Walker, N., Ribarsky, W. & Spaulding, V. (1998). Effects of variation in system responsiveness on user performance in virtual environments. *Human Factors*, Special Section on Virtual Environments, in press.

Wloka, M. (1995). Lag in multiprocessor VR. *Presence*, *4*, 1, 50-63.

SIGGRAPH 2001

Perceptually Driven Interactive Rendering

David Luebke and Benjamin Hallen

University of Virginia Tech Report #CS-2001-01

Abstract

We present a framework for accelerating interactive rendering, grounded in psychophysical models of visual perception. This framework is applicable to rendering techniques that regulate resolution—and hence rendering load—using a hierarchy of local simplification operations. Our method drives those local operations directly by perceptual metrics: the effect of each simplification on the final image is considered in terms of the contrast the operation will induce in the image and the spatial frequency of the resulting change. A simple and conservative perceptual model determines under what conditions the simplification operation will be perceptible, enabling *imperceptible simplification* in which operations are performed only when judged imperceptible. Alternatively, simplifications may be ordered according to their perceptibility, providing a principled approach to best-effort rendering. Our approach addresses many interesting topics in the acceleration of interactive rendering, including imperceptible simplification, silhouette preservation, and gaze-directed rendering. We demonstrate our framework applied to two quite different multiresolution rendering paradigms: view-dependent polygonal simplification and the *QSplat* point-based rendering system of Rusinkiewicz and Levoy [26].

1 Introduction

Interactive rendering of large-scale geometric datasets continues to present a challenge for the field of computer graphics. Such interactive visualization is an enabling technology for many far-flung fields, ranging from scientific and medical visualization to entertainment, architecture, military training, and industrial design. Despite tremendous strides in computer graphics hardware, the growth of large-scale models continues to outstrip our capability to render them interactively. A great deal of research has therefore focused on algorithmic techniques for managing the rendering complexity of these models. *Level of detail* management offers a powerful tool for this task. Level of detail or *LOD* methods hinge on the observation that most of the complexity in a detailed 3-D model is unnecessary when rendering that model from a given viewpoint. These methods simplify small, distant, or otherwise unimportant portions of the scene, reducing the rendering cost while attempting to retain visual fidelity.

Visual fidelity, however, has traditionally been difficult to quantify, so most LOD algorithms settle for geometric measures of quality. For example, the most common use of LOD management is *polygonal simplification*, in which a 3-D polygonal model is replaced with a simpler model using fewer polygons. Here geometric fidelity of the simplified surface may be measured with the distance of that surface from the original mesh, or with the volume of distortion created by the simplification. Such metrics are useful for certain CAD applications, such as finite element analysis, and for certain medical and scientific visualization tasks, such as co-registering surfaces or measuring volumes. Probably the most common purpose of simplification, however, is to accelerate interactive rendering. For this purpose, the most important measure of fidelity is not geometric, but perceptual: does the simplification *look* like the original?

In this paper, we describe a framework for LOD management guided directly by perceptual metrics. These metrics derive from the *contrast sensitivity function* or *CSF*, a measure of the perceptibility of visual stimuli. Testing local simplification operations against a model of the CSF provides a principled approach to the tradeoff between fidelity and performance. This approach addresses several interesting problems in regulating level of detail:

- **Imperceptible simplification:** We evaluate simplification operations by the "worst-case" contrast and spatial frequency they induce in the image, and apply only those operations judged imperceptible. We hypothesize that the resulting simplified model is indistinguishable from the original, and test that hypothesis in this paper.
- **Best-effort simplification:** Often we wish to render the best image possible within time or polygon constraints. Ordering simplification operations according to the viewing distance at which their effect on the image becomes perceptible furnishes a framework for simplifying to a budget.
- **Silhouette preservation:** Silhouettes have long been recognized as visually important, but how important? Our model quantifies silhouette importance by accounting for their increased contrast, and preserves them accordingly.
- **Gaze-directed rendering:** If the system can monitor the user's gaze, the image may be simplified more aggressively in the periphery than at the center of vision. We can extend our model to incorporate *eccentricity*, or the falloff of visual acuity in the periphery.

Our framework applies to any rendering system based on hierarchical simplifications or approximations. Many interactive rendering acceleration schemes fall into this category, including

Figure 1: Perceptually driven QSplat. Splats drawn in blue have been simplified. Left: QSplat's highest quality rendering mode traverses until each splat is less than a pixel in size. Right: perceptually driven rendering traverses nodes only where dictated by the local contrast and spatial frequency. Our model can also account for gaze direction: here, the user's gaze rests on Lucy's torch.

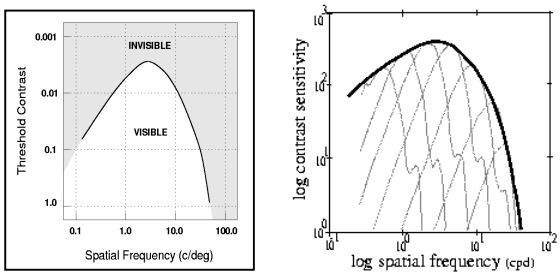


Figure 2: (a) The *contrast sensitivity function* measures the perceptibility of visual stimuli (sinusoidal gratings) in terms of their contrast and spatial frequency (cycles per degree). (b) The shape of the CSF is attributed to the aggregate response of multiple bandpass mechanisms in the visual system. Courtesy Martin Reddy, Mahesh Ramasubramanian.

the use of polygonal simplification, texture-based imposters, and some forms of image- and point-based rendering. We have applied perceptually driven interactive rendering to two such schemes: view-dependent polygonal simplification and the point-based renderer QSplat.

Below we briefly present some background on visual perception, followed by an overview of our framework. We illustrate this overview using QSplat, which was relatively straightforward to dovetail with perceptually driven rendering. We then turn to the application of our framework to polygonal simplification, which was more involved. We describe the underlying algorithm and the evaluation of contrast and spatial frequency induced by simplification, which is ultimately the heart of the problem. Finally, we consider our framework in relation to previous work on perceptually guided rendering, and close with some thoughts on future extensions and improvements.

2 The Contrast Sensitivity Function

A large body of perceptual psychology literature focuses on the perceptibility of visual stimuli. The simplest relation established in this literature is *Weber's law*, which predicts the minimum detectable difference in luminance between a test spot on a uniform visual field. Weber's law states that at daylight levels the threshold difference in luminance increases linearly with background luminance. Interesting scenes are not uniform, however, but contain complex frequency content. Outside a small frequency range, the threshold sensitivity predicted by Weber's law drops off significantly. Many perception studies have therefore examined the perceptibility of *contrast gratings*, sinusoidal patterns that alternate between two extreme luminance values L_{max} and L_{min}. Campbell first characterized the perceptibility of a contrast grating in terms of its contrast and spatial frequency [4]. Contrast grating studies use *Michaelson contrast*, defined as $(L_{max} - L_{min}) / (L_{max} + L_{min})$, and *spatial frequency*, defined as the number of cycles per degree of visual arc. The *threshold contrast* at a given spatial frequency is the minimum contrast which can be perceived in a grating of that frequency, and *contrast sensitivity* is defined as the reciprocal of threshold contrast. The *contrast sensitivity function* (CSF) plots contrast sensitivity against spatial frequency, and so describes the range of perceptible contrast gratings [Figure 2a].

Of course, most interesting images are more complex than the simple sinusoidal patterns used in contrast gratings. Campbell found that the perceptibility of complex signals could be determined by decomposing a signal into sinusoidal components using Fourier analysis [5]. In particular, if no frequency component of a signal is perceptible, the signal will not be perceptible.

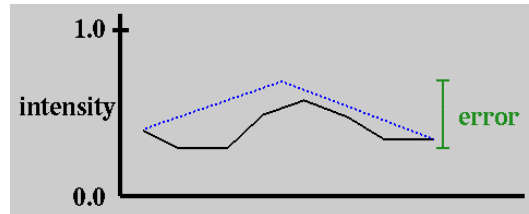


Figure 3: We use a conservative approximation of the change in intensity (error) induced by folding a node. Here the black line represents the original surface and the blue line the simplified surface.

Note that the CSF predicts the maximum perceptibility of a stationary grating at the center of view; many other factors can lower contrast sensitivity. Among these is *eccentricity*, or distance from the direction of gaze. The *fovea* is the region of highest sensitivity on the retina, occupying the central 1º or so of vision. *Visual acuity*, measured as the highest perceptible spatial frequency, is significantly lower in the visual periphery than at the fovea. The relationship between visual acuity and eccentricity, defined as angular distance from the fovea, was first characterized in humans by Rovamo and Virsu [25]. By extending our perceptual model to incorporate eccentricity, we can predict the visibility of peripheral features for use in gaze-directed rendering.

3 Overview

Our goal is to analyze LOD-based rendering algorithms within a principled perceptual framework inspired by the contrast grating studies described above. We map the change resulting from a local simplification operation to a *worst-case* contrast grating, meaning a grating with the most perceptible combination of contrast and frequency possibly induced by the operation. We apply the simplification only if we would not expect a grating with that contrast and frequency to be visible. Our hypothesis: if we can isolate the most perceptible frequency component possibly induced by a simplification operation, and determine that a contrast grating at that frequency would not be visible, we can perform the operation without perceptible effect.

3.1 Determining the Worst Case

To determine the worst-case frequency and contrast efficiently, we make some conservative simplifying assumptions. First, we observe that the peak contrast sensitivity occurs at approximately 2-4 cycles per degree, and that most local simplification operations on a complex model affect only much higher frequencies. We therefore assume that contrast at lower spatial frequencies is more perceptible than at higher frequencies (Section 6.1 describes how we adjust our perceptual model to ensure this assumption holds). The minimum frequency component of a region in the image spanning n degrees of the user's angular field of view is one cycle per $2n$ degrees. Put another way, the maximum wavelength needed to represent a region of the image is twice the maximum spatial extent of that region [Figure 3]. Consequently, finding the worst-case frequency induced by a simplification reduces to finding the screen-space extent of the affected region.

For the worst-case contrast, we determine a bound on the maximum change in luminance among all the pixels affected by the simplification. Put another way, the worst-case contrast of a

simplification operation is the maximum contrast between an image of the affected region at full resolution and an image of the region simplified. For 3-D models, there are two basic cases:

- The entire affected region lies interior to a surface that entirely faces the viewer. This is the simplest case: the contrast between the original region and the folded region is completely determined by the luminance of the local surface before and after the fold.
- The affected region includes a silhouette edge. This expands the possible contrast incurred by the simplification to include the portion of the scene *behind* the affected region, since simplifying the surface may expose a very bright or very dark feature occluded before simplification.

Consequently, silhouette regions of the object are simplified less aggressively—exactly the behavior we should expect in a perceptually driven simplification algorithm. Note, however, that even at these high contrast levels silhouette regions can still be simplified if they represent very fine details (high spatial frequencies) or are in the viewer's peripheral vision (high eccentricity).

3.2 An Empirical Perceptual Model

Many researchers have characterized the contrast sensitivity function. In early work, Kelly derived an abstract relationship for the perceptibility of sinusoidal gratings over a narrow range: $C_T = \alpha^2 e^{-\alpha}$ [14]. Here C_T represents the threshold contrast and α represents spatial frequency. The more accurate CSF curve given by Barten [2] is used in recent advanced global illumination algorithms, such as the physically based metric of Ramasubramanian, which also account for adaptation effects due to background illumination [23][3]. Modern perceptual theory attributes the shape of the CSF to the combined response curves of multiple bandpass mechanisms in the visual system, each processing only a small range of the visible spatial frequency spectrum [Figure 2b]. This multiscale visual processing can be emulated with a Laplacian pyramid for spatial decomposition [17]. Current perceptual rendering techniques also account for *contrast masking*, which represents the visual system's decreased contrast sensitivity in the presence of strong patterns. This can further increase the allowable error in an image [9][23][3].

Unfortunately, these sophisticated perceptual models, which employ the latest advances in understanding perception, are far too costly for the interactive framework we propose. In our framework, thousands of simplification operations must be considered every second, leaving less than a millisecond to evaluate the induced contrast and frequency. Clearly, we must forego the state-of-the-art perceptual models used in current global illumination work for a model that is simple, fast, and conservative.

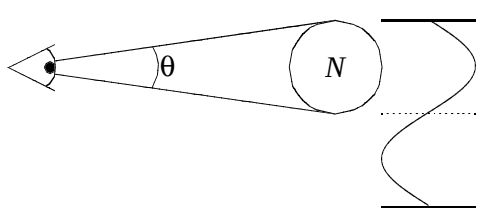


Figure 5: The lowest spatial frequency that can be affected by a node spanning θº of visual arc has one cycle per 2θº.

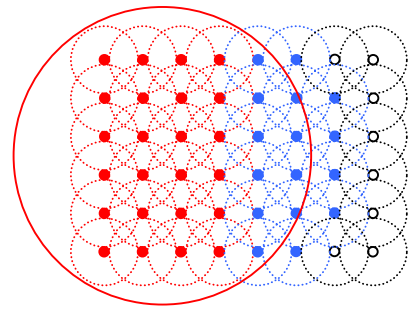


Figure 4: Estimating contrast in QSplat. Here the small circles represent leaf nodes (samples) with their bounding spheres. The sphere of an internal node **N** (drawn in red) must contain the spheres of all its descendents (also in red). To compute contrast induced by simplifying **N**, we must also account for nodes intersecting **N** (drawn in blue), since these nodes may be occluded by **N**'s splat.

To achieve simplicity and speed while still accounting for real-world factors that affect perception, such as ambient light, we chose to take an empirical approach. Recall our hypothesis: a simplification operation, mapped to a "worst-case" contrast grating, can be performed imperceptibly if that grating would not be perceptible. We build our perceptual model directly from contrast grating tests performed under the same conditions—room illumination, monitor, etc—under which our final system will run. A calibration procedure, detailed below, tests the ability of a user to detect contrast gratings, recording threshold contrast over a wide range of spatial frequency and eccentricity. We then build a lookup table from the resulting CSF curves and use linear interpolation at runtime to determine whether the user can perceive a given contrast at a given spatial frequency and eccentricity.

This model could certainly be improved, but we chose to focus on developing a framework for driving interactive rendering with a perceptual model, rather than on developing the model itself. Our empirical model is simple to implement and works well in practice; Figure 6 shows example CSF curves determined from a typical calibration procedure.

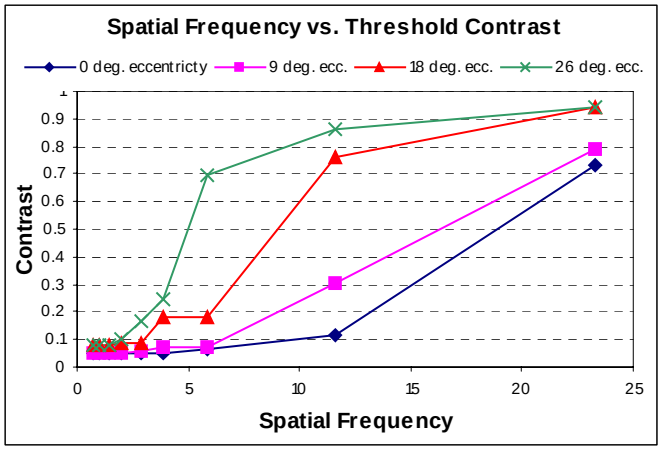


Figure 6: We use a simple model of contrast sensitivity based on an empirical calibration procedure. Shown here are results from one user's calibration.

4 Perceptually Driven Point Rendering

The QSplat point-rendering system lends itself nicely to our perceptual rendering framework. While Rusinkiewicz and Levoy describe many careful optimizations, tradeoffs, and design decisions that make QSplat exceptionally fast [26], at its heart it is a simple algorithm. Qsplat uses a hierarchy of bounding spheres for culling, rendering, and level of detail management. The leaf nodes in this hierarchy of spheres form a dense tiling of the original surface and typically represent samples from a 3-D scanning device. Internal nodes completely contain the spheres of their descendents, and average the color and normal of their descendents. The hierarchy is traversed in a view-dependent fashion: backfacing and invisible nodes are skipped and their descendents not traversed, while nodes below a certain size on screen are drawn (using a *splat*) without descending further. The core simplification operation in QSplat, then, is to draw a node splat rather than traverse its descendents.

4.1 Applying the Framework

To apply perceptually guided rendering to QSplat, we must evaluate the spatial frequency and contrast induced by a simplification. Since the sphere at each node completely contains all its descendents, we can bound the affected region of the image by the projected screen-space extent of the sphere. Since QSplat already computes this diameter, we need only convert it into the appropriate spatial frequency.

Determining Maximal Contrast

Given a worst-case spatial frequency for the change induced by folding a node, we next need to compute the worst-case contrast of that change. Recall that Michaelson contrast depends on L_{max} and L_{min}, the maximum and minimum brightness of the region affected by the fold operation. Brightness, or luminance, is measured in cd/m^2, so we must convert OpenGL vertex colors to actual luminance.

Simplifying a node **N** draws its splat rather than the splats of its descendents, but may also occlude nodes that do not descend from **N**. Though it is difficult to determine which nodes may be occluded in certain pathological cases, in practice we can account for occlusion by considering all nodes intersecting the bounding sphere of **N** [Figure 4]. The maximum contrast that can be caused by folding **N** is then determined by the color of **N** and the range of colors of nodes intersecting **N**. We then calculate the induced Michaelson contrast by converting all colors to luminance, and comparing the minimum and maximum luminance of the original surface to the luminance of the simplified node. We can efficiently compute this information during preprocessing, quantize contrast to 8 bits, and store it with the node.

Figure 7: Gaze-directed rendering. Left: At high quality QSplat uses 2.9 million points to render Lucy. Right: With the user's gaze 29º away, our system imperceptibly simplifies the model to 0.8 million points. Inset: at this distance all nodes are simplified. Originally rendered at 1600x1200 on a 23" monitor.

Of course, this precomputed contrast becomes invalid if the node lies on the silhouette. QSplat stores a normal and *normal cone* [28] with each node, and evaluates whether the node represents a surface region that faces entirely away from the viewer (for backface culling) or entirely towards the viewer (to disable backface culling). We can thus identify silhouette nodes with no additional computation, since all nodes that are neither entirely backfacing nor entirely frontfacing might lie on the silhouette. As with contrast, we need to modify the preprocessing to account for contained nodes when computing normal cones. For simplicity, and to avoid extra storage, we assume that simplifying any silhouette node could induce the maximum possible contrast.

Putting It All Together

Given the worst-case contrast and spatial frequency, both readily available at run-time, our perceptually driven algorithm for QSplat rendering is nearly as simple as the original:

```
TraverseHierarchy(node)
{
    if (node outside view frustum)
        skip this branch
    if (node is backfacing)
        skip this branch
    // Worst-case frequency has period twice sphere diameter:
    frequency = 1/(4*node->radius);
    if (node is frontfacing)
        contrast = node->contrast;
    else
        // Node is on silhouette, use maximum possible contrast
        contrast = maxContrast;
    if (IsPerceptible(frequency, contrast)
        // Simplifying node would be perceptible, keep going
        foreach child of node
            TraverseHierarchy(child);
    else
        // Can simplify node imperceptibly
        DrawSplat(node);
}
```

Here spatial frequency is represented in cycles per radian. The function `IsPerceptible()` looks up the given contrast and frequency and returns `TRUE` if simplifying the node to a single splat might be perceptible. In practice, we store the lookup table in units of wavelength rather than frequency to avoid the extra divide. If gaze-directed rendering is desired, the traversal function calculates angular distance from the gaze point to the node, and passes this value to `IsPerceptible()` as eccentricity.

Figures 1 and 7 demonstrate our perceptually driven QSplat system.

5 Perceptually Driven Polygonal Simplification

Regulating scene complexity and rendering time by simplifying small or distant objects was first proposed in Clark's seminal 1976 paper [6]. The basic approach described there remains the most common approach today: create several versions of each object at progressively coarser levels of detail in a preprocess, and choose at run-time which version (called *LODs*) will represent the object. The past decade has seen a flurry of research into *polygonal simplification*: algorithms for generating coarse LODs from full-resolution polygonal models and for managing which LODs replace each object. Several recent surveys examine the field of polygonal simplification [11][22][19]; in Section 7 we examine some algorithms relevant to perceptually based rendering.

One difficulty with traditional LOD-based approaches is their reliance on a few discrete levels of detail to represent each object. This limits the degree to which perceptual metrics can be applied, since the entire object must be simplified uniformly. For example, silhouette details tend to be more perceptible than interior details because of higher contrast, so the entire object must be treated as if it were on the silhouette. Similarly, if the user's eye rests on any portion of the object, a system that accounts for eccentricity must treat the entire object as if it were under direct scrutiny. The worst-case assumptions of per-object LOD can severely handicap perceptually guided polygonal simplification.

5.1 View-Dependent Simplification

View-dependent simplification methods offer a solution. Rather than calculating a series of static levels of detail in a preprocess, view-dependent systems build a dynamic data structure from which a desired level of detail may be extracted at run time. Objects in a view-dependent algorithm may span multiple resolutions, solving the worst-case behavior of traditional LOD. For example, portions of the object under the viewer's gaze can be represented at higher fidelity than portions in the peripheral vision, and regions of the object moving slowly across the visual field could utilize higher resolution than fast-moving regions.

Several researchers have independently proposed view-dependent algorithms, including Hoppe, Luebke, and Xia [12][18][29]. These algorithms share a common feature: each is a hierarchy of *vertex merge* operations that can be applied or reversed at run-time. Our chief contribution is a method for evaluating the perceptibility of a vertex merge operation, using factors such as contrast, spatial frequency, and eccentricity. We have implemented our system using *VDSlib*, a public-domain library based on the view-dependent simplification framework of Luebke [20]. VDSlib allows users to plug in custom callbacks for building, culling, simplifying, and rendering the vertex tree. We first augment the nodes of a VDSlib vertex tree with data specific to our perceptual simplification process, such as the contrast induced by a fold operation and the normal mask used for silhouette detection. Then at run time, our callback examines nodes, using contrast, spatial frequency, and possibly eccentricity to decide whether VDSlib should fold the node. Before describing the details of this process, we briefly review the VDSlib algorithm and notation.

The main data structure of VDSlib is the *vertex tree*, a hierarchical clustering of vertices. Vertices from the original model are grouped with nearby vertices into clusters, then the clusters are clustered together, and so on. Leaf nodes of the tree represent a single vertex from the original model; interior nodes represent multiple vertices clustered together, and the root node represents all vertices from the entire model, merged into a single cluster. In VDSlib parlance, a node **N** *supports* a vertex **V** if the leaf node associated with **V** descends from **N**. Similarly, **N** *supports* a triangle **T** if it supports one or more of the corner vertices of **T**. The set of triangles in the model supported by a node is called the *region of support* of the node.

Each node stores a representative vertex called the *proxy*. For leaf nodes, the proxy is exactly the vertex of the original model that the node represents; for interior nodes, the proxy is typically some average of the represented vertices. *Folding* a node merges all of the vertices supported by that node into the node's single proxy vertex. In the process, triangles whose vertices have been merged together are removed from the scene, decreasing the overall polygon count. Since folding a node is the core simplification operation of VDSlib, to apply our perceptual framework we must evaluate the contrast and spatial extent of the change in the rendered image induced by a fold.

5.2 Applying the Framework: VDSlib

The effect of folding a node in VDSlib is more complex than the effect of drawing a splat in QSplat. As the vertices and triangles supported by the node merge and shift, features in the image may shrink, stretch, or disappear completely. Shifting triangles on the visual silhouette may expose previously occluded features. To analyze the effect of folding a node, we should consider all of these changes. One possibility, recently demonstrated by Lindstrom and Turk for static LOD generation, is to render the scene before and after the operation and analyze the resulting images [16]. At present, however, the requisite rendering and image processing appears too expensive for dynamic simplification. Instead, we want a conservative worst-case bound on the changes in the image caused by folding the node. Since our goal is to evaluate a hypothetical change *at least* as perceptible as any changes that folding actually incurs, we consider the removal of a feature with worst-case size and contrast.

Spatial Frequency: Estimating Extent

Again, the minimum frequency induced by a simplification is determined by the spatial extent of the resulting change in the image. Notice that features in the image affected by a fold consist of triangles connecting vertices involved in the fold. The largest feature that can be removed or exposed by geometric distortion upon folding a node is therefore constrained by the distance vertices move during the fold. Thus, the problem of computing the minimum frequency induced by folding a node reduces to computing the screen-space extent of all vertices supported by the node.[1] As with QSplat, we use bounding spheres to estimate this extent, associating with each node a tight-fitting sphere that contains all vertices in the node's region of support. The angular extent of these bounding spheres, as seen from a given viewpoint, can be calculated very quickly. The minimum frequency affected by folding a node is then one cycle

[1] Technically, this holds when the model is flat shaded; for Gouraud-shaded models, adjacent vertices should also be included. However, we have not found this necessary in practice.

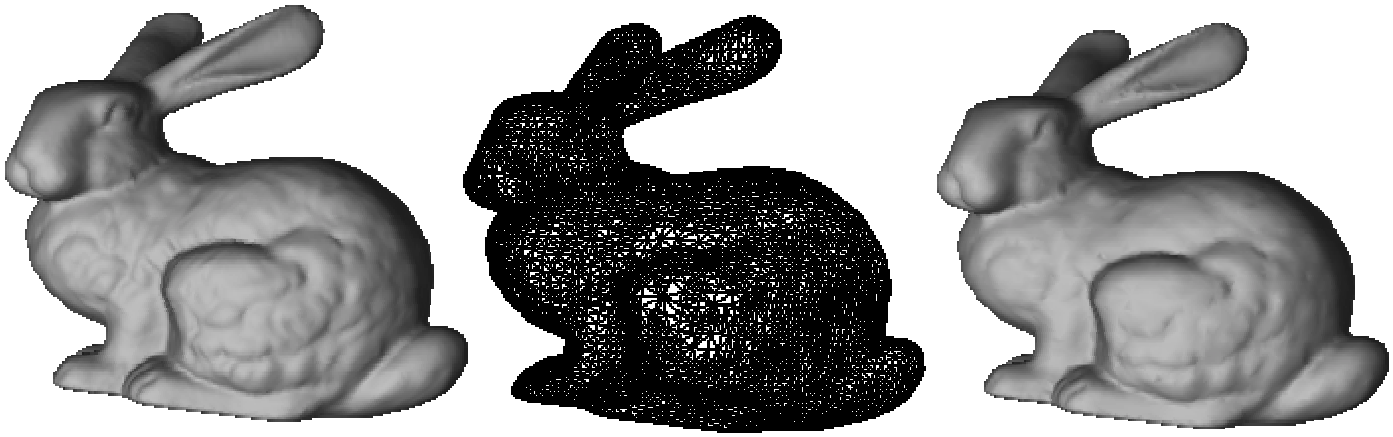


Figure 8: The original Stanford Bunny model (69,451 faces) and a simplification by our perceptually driven system (29,866 faces). In this view the user's gaze is 29° from the center of the bunny…equivalent to looking at the page number on this page from a distance of 10". Note that the silhouette is well preserved, along with strong interior details (the line of the haunch, the shape of the eye, etc.) while subtle bumps on the surface are simplified.

per two degrees of angular extent spanned by the node's bounding sphere [Figure 4].

Contrast: Estimating Intensity Change

Determining the exact contrast induced by folding a node would be as expensive as rendering the unfolded geometry. Instead, we obtain a conservative lower bound by comparing the intensities of all the vertices the node supports in the original model with the intensities of the vertices in the simplified surface [Figure 3]. The greatest difference between the intensities of the surface vertices before folding and after folding bounds the maximum contrast between the simplified surface and the original surface, since in a Gouraud-shaded model extremes of intensity always occur at the vertices. This conservative test may overestimate the contrast induced by folding, but will not underestimate it.

When the node's region of support includes a silhouette, we must be even more conservative. Lacking knowledge about what lies behind the model, we must assume the worst: moving a silhouette edge might expose the darkest or brightest object in the scene, including the background. Hence we must compare the range of vertex intensities of the node's region of support against the brightest and darkest intensities in the scene, and use the maximum possible difference in intensity for calculating the contrast induced by the fold.

Determining Silhouette Nodes

Since nodes affecting silhouette edges must be treated differently, we require an efficient method for identifying such nodes. For a given view, we define *silhouette node*s as those nodes supporting both front-facing and back-facing triangles in the original mesh. We initially employed Shirman's *cone of normals* approach [28], used by both Luebke and Hoppe [18][12], to determine silhouette nodes. Unfortunately, the cone of normals sometimes proves overly conservative, especially in models with sharp edges; we found that too many interior polygons were being classified as belonging to silhouette nodes. Instead, we used a bitwise approach inspired by the rapid backface culling technique of Zhang and Hoff [30]. We map the Gauss sphere of normal space to a *normal cube* whose faces are tiled into cells, in effect quantizing the space of normals. Each node in the model stores a *normal mask*, a bit vector representing the normals of all its supported triangles. A bit in the mask is set if a triangle normal falls within the corresponding cell of the normal cube.

The accuracy of the normal mask is bounded only by the number of cells, which depends on the length of the bit vector. This improves significantly over the cone of normals, which can greatly overestimate the range of normals. Normal masks are efficient to compute, since they can be propagated up the vertex tree using bitwise-OR operations. Testing whether the node might lie on the silhouette can also be made very efficient by precomputing two bitmasks, representing the space of normals that might be backfacing and frontfacing, respectively. A node may be on the silhouette if its normal mask overlaps with both the frontfacing and the backfacing bitmasks. The test to classify a silhouette node therefore reduces to two bitwise-AND operations, whose cost depends on the length of the bit vector. We chose 48 bytes (64 bits per face of the normal cube) for accuracy, but if memory is at a premium, fewer bits could be used to trade off accuracy for compactness.

5.3 Perceptually Guided Best-Effort Rendering

Imperceptible simplification makes a guarantee about the visual fidelity of the simplified scene. Often, however, a guarantee about the complexity, and thus rendering time, is desired instead. VDSlib supports *triangle budget simplification*, which allows the user to specify how many triangles the scene should contain. Using a user-specified run-time error metric, VDSlib then minimizes the total error induced by all folded nodes within this triangle budget constraint. Internally, VDSlib performs triangle budget simplification using a greedy approach. A priority queue of boundary nodes is sorted by induced error, as evaluated by a user-supplied callback. The node **N** with the greatest error is unfolded, adding some triangles to the scene. **N** is then removed from the priority queue and its children inserted back into the queue. This process iterates until unfolding the top node of the queue would exceed the triangle budget. [2]

For principled best-effort rendering, then, we must generate a sound perceptual measure of the error introduced by folding a node. The key is to recast our metric for evaluating the

[2] Note that this assumes error decreases monotonically; folding a node should not induce more error than folding its parent.

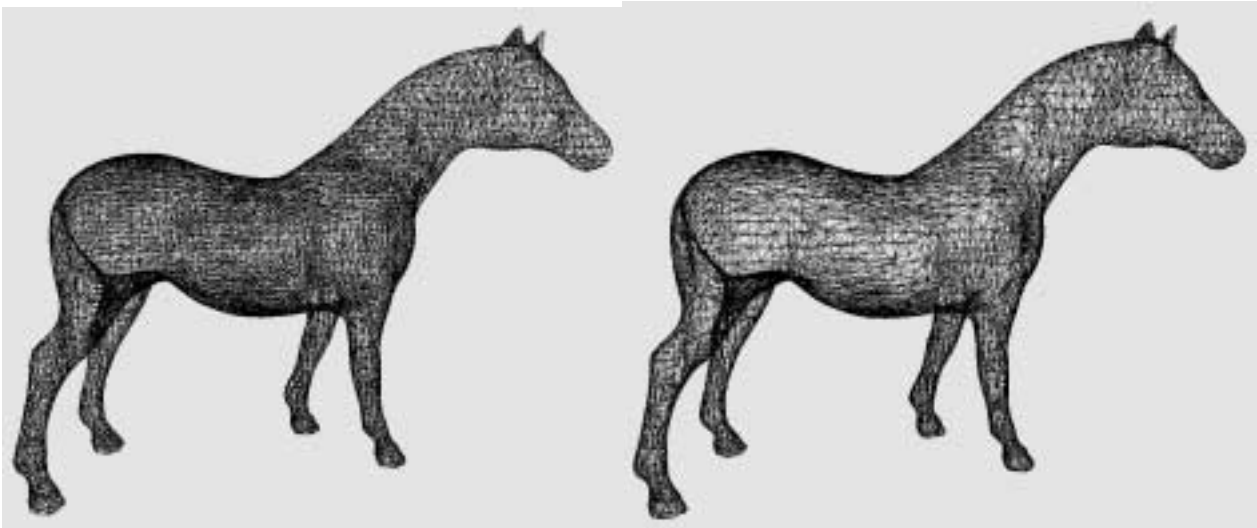


Figure 9: Perceptually driven best-effort simplification. Both images show the horse model (originally 96,966 faces) reduced to 18,000 faces using triangle budget rendering in VDSlib. Left, the default VDSlib error metric uses screenspace node size, leading to unnecessarily even tesselation. Right, our perceptually driven metric uses fewer polygons in interior and low-contrast regions.

perceptibility of fold operations. Rather than a binary perceptible/not perceptible decision, we need a scalar to express *how* perceptible a fold operation could be. We chose to cast the question in terms of distance: how far would the viewer have to be from the screen before the node could be folded imperceptibly? The answer can be computed from our current perceptual model, in effect by inverting our lookup tables. Rather than computing the spatial frequency of a node and looking up the threshold contrast at which folding is perceptible, we use the precomputed contrast induced by folding and look up the threshold spatial frequency. From this we can compute the distance at which folding the node would be perceptible, and sort nodes in the priority queue by this distance.

To recap, we order folds based on the viewing distance at which they become perceptible. This provides a convenient framework for best-effort triangle budget simplification, and an intuitive physical measure of the fidelity achieved: after simplifying to the user-specified number of triangles, the system can report the distance at which that simplification should be imperceptible.

5.4 Results

All results given are on an 866 MHz Pentium III computer with NVidia GeForce^2 graphics. Figures 8 and 9 show models simplified with our perceptually driven algorithm. Since we are guaranteeing imperceptible simplification, the reductions in polygon count may seem modest. However, these results and the user study below clearly show that perceptually driven simplification can reduce model complexity without visual effect.

Perceptually driven best-effort rendering may be of more use to many 3-D applications. Figure 9 compares our results to VDSlib's built-in triangle budget rendering, which orders fold operations only by size of the node. Note that the perceptually driven algorithm preserves more triangles near silhouettes, and simplifies more aggressively in regions of low contrast.

6 Implementation and Evaluation

6.1 Calibration: Building a Perceptual Model

The goal of our calibration step is to build a simple empirical model of threshold contrast across different values of spatial frequency and eccentricity. Building this model at runtime allows us to account for factors that affect perception but are not likely to vary over the course of a viewing session, such as room illumination and visual acuity of the individual user. During the calibration, the user fixates on a target while a grating fades in, slowly increasing in contrast. The user is instructed to press the mouse button when something becomes visible. To verify that the user actually saw the grating, he then clicks on it (without looking away from the target). By varying the spatial frequency and eccentricity of the gratings presented to the user, we find the threshold contrast across these parameters [Figure 6]. From the data sampled during calibration we build the threshold contrast lookup table used at runtime.

Recall that our perceptual model assumes low frequencies are more perceptible than high frequencies, so that we can use node spatial extent to determine a worst-case contrast. This assumption holds true for most simplification operations we are concerned with, which extend a few pixels at most. To ensure that we make a conservative choice, we modify our lookup table to effectively clamp the threshold contrast below the most sensitive frequencies.

We must also calibrate the monitor, in order to translate 24-bit OpenGL color values into luminance values (cd/m^2) used by our definition of contrast. This step involves measuring with a photometer the light levels produced by different red, green, and blue OpenGL intensities, and building a lookup table from the results. Fortunately, this part of the calibration need be repeated only as often as monitor drift warrants. If full precision were not necessary, simple gamma correction would suffice.

6.2 User Study: Evaluating Imperceptibility

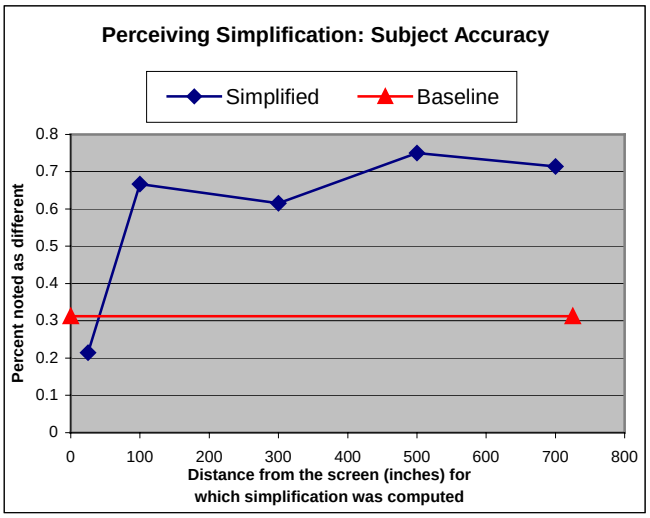
We performed a user study to evaluate our system more formally, determining whether our algorithm can indeed produce a simplification imperceptible from the original model. The study tested whether subjects could perceive the difference between a rendering of a full-resolution model and a rendering of a model simplified with our algorithm. If our hypothesis holds, a subject's ability to discern the simplification will be no better than chance.

As a control, we also evaluated the ability of subjects to discern simplifications that our model predicts could be visible. To this end, we randomly interspersed trials in which the "imperceptible" simplification was calculated for an incorrect field of view. Since the field of view varies with the distance δ of the user to the monitor, this amounts to calculating simplifications for incorrect viewing distances. For example, a simplification that assumes the user is ten times further from the screen than in reality will probably be visibily different than a simplification computed for the correct distance. The viewing distance δ in these trials was chosen so that simplifications should range from imperceptible (δ = correct distance to screen) to clearly perceptible.

The study consisted of 4 subjects, each of whom performed 200 trials. During each trial, the subjects fixated on a target (a short line segment) in the center of the screen. They were then shown the same 3-D object twice in succession, identical in all parameters except resolution. 25% of the trials displayed the object twice at full resolution, while another 25% displayed once at full resolution and once using our imperceptible simplification. For the remaining trials, one was presented at full resolution and the other was presented at a reduced resolution computed using "imperceptible" simplification for a random viewing distance.

Objects were displayed for 1 second and separated by 500 milliseconds, with a neutral grey background before, after, and between scenes. When the second object finished displaying, the subject pressed a key to indicate whether they thought the models differed. To avoid subject fatigue, the next trial did not begin until 500 milliseconds after the subject had pressed a key.

After a practice session of 20 trials, each subject performed 200 trials in a continuous session. Subjects viewed 3 models (bunny, horse, rhino) from 40 random viewpoints for each viewing distance used by the simplification algorithm. Viewing parameters were chosen so that the subject viewed the object at randomly distributed orientations from randomly distributed directions. Object size was randomly chosen such that the visual angle subtended by the object was uniformly distributed from 5° to 60°. The screen occupied approximately 46° of the subject's field of view, so that in some views the object filled most of the screen. When adjusting viewing distance, the value was chosen from 1-50 times the actual distance of the viewer from the screen. We picked these values to span the range between clearly perceptible and completely imperceptible simplification.



Each subject reported normal eyesight, some with corrective lenses. Subject accuracy is plotted against distance δ. *Baseline* represents the willingness of subjects to report a difference between the models when none existed. As the graph shows, our system does produce imperceptible simplification: at the correct viewing distance (25 inches), subject accuracy, defined as ability to perceive simplification, is no better than baseline.

7 Previous Work

7.1 Perceptually Based Offline Rendering

Many researchers have worked on perceptually based rendering algorithms; Bolin and Meyer [3] and Ramasubramanian et al [23] provide nice surveys of the field. These algorithms take advantage of the limitations of human vision to avoid rendering computation where the result will be imperceptible. Unlike our work, which seeks to accelerate interactive rendering, almost all previous perceptually based rendering approaches have addressed realistic offline rendering approaches such as ray and path tracing. Since image creation times in such approaches are typically measured in seconds or minutes, these algorithms are able to use very sophisticated perceptual models.

The state of the art in perceptually guided realistic rendering is well exemplified by the perceptually based physical error metric of Ramasubramanian et al [23]. Their perceptual model combines threshold sensitivity for varying illumination, contrast sensitivity with multi-scale spatial frequency sensitivity, and visual masking to predict the maximum imperceptible change in luminance at a given pixel in the rendered image. This advanced model, which incorporates the latest advances in our understanding of the visual system, is costly to evaluate by interactive rendering terms. Despite a novel framework that does not require reevaluation at every stage of the progressive illumination computation, the authors report that evaluating the model for a 512x512 image required about 12 seconds on a 400 MHz CPU.

7.2 Perceptually Based LOD Selection

Comparatively few systems have attempted to guide interactive rendering with explicit perceptual metrics. Funkhouser and Sequin's system for dynamic LOD selection uses a cost-benefit estimate to pick the best levels of detail within a specified time budget [10]. LOD benefit is assigned heuristically, based primarily on an object's screen-space size. Their system also takes into account eccentricity and *velocity*, the speed at which the image of an object moves across the retina. Lacking eye or head tracking, the user's gaze is assumed to lie in the center of the screen; lacking accurate perceptual models, the effects of velocity and eccentricity are controlled with sliders set by the user. Though the use of these factors is ad hoc, this important work

introduced the notion of perceptually guided metrics to drive LOD selection.

Ohshima et al described a system for gaze-directed stereoscopic rendering [21]. Although the paper mentions an eye-tracking system in progress, the results were gathered using head-tracked viewing direction to approximate gaze direction. Their system uses eccentricity, velocity, and convergence to guide selection of precomputed LODs. The equations used to model the three perceptual effects, and the method for combining all three effects to choose an LOD, appear to have been determined empirically. Thus their algorithm, while clearly demonstrating the potential of a gaze-directed approach, still employs a fundamentally heuristic model of the visual system.

Reddy was the first to attempt an LOD selection system guided throughout by a principled perceptual model [24]. Using images rendered from multiple viewpoints, Reddy analyzes the frequency content of objects and their LODs. A model of the visual acuity, defined as highest perceptible spatial frequency, guides LOD selection. If a high-resolution and a low-resolution LOD differ only at frequencies beyond the visual acuity of the viewer, those differences are imperceptible and the low-resolution LOD may be used. Reddy analyzes the frequency content of LODs, rendering each from several directions, and models the decrease in visual acuity with eccentricity and velocity to decide which LOD to use.

8 Summary And Discussion

Perceptually guided interactive rendering is a broad and difficult topic. Our system shows the feasibility and potential of imperceptible view-dependent simplification, but many avenues for further research remain. Below we summarize our contribution and results, and address what we see as the most pressing and interesting directions for future work.

8.1 Summary

We have demonstrated a novel approach to accelerating interactive rendering that is directly driven by perceptual criteria. Our principle contribution is a practical framework for perceptually guided interactive rendering that equates local simplification operations to worst-case contrast gratings whose perceptibility we can evaluate. We have shown an application of the framework to the QSplat point-rendering system, and demonstrated it in depth in the context of view-dependent polygonal simplification. Our approach addresses several interesting problems, including silhouette preservation and imperceptible simplification. An optional gaze-directed component uses eye tracking to obtain further simplification by reducing fidelity in the viewer's peripheral vision.

8.2 Improving the Current System

We see several opportunities to improve the current system. Incorporating dynamic lighting into the contrast calculation is an obvious extension. The difficulty lies in determining the possible range of intensities across the original surface without having to render that surface. Since lighting calculations depend on the surface normal, we might use the current normal masks, which bound the space of normals subtended by each node's region of support, to compute the minimum and maximum intensities of that region under the given lighting conditions. Accounting for non-directional light sources complicates the problem, but a solution certainly appears feasible.

Our estimate of the spatial frequency induced by a fold is overly conservative. We currently treat the worst case by assuming the entire screenspace extent of the node changes; the resulting spatial frequency bounds the minimum frequency change induced. In practice, however, the actual frequencies affected by a fold are typically higher (and hence less perceptible) than this worst-case estimate. A tighter estimate of spatial frequency would, we suspect, greatly improve the amount of simplification possible at a given contrast. For example, we currently use spheres to bound each node's region of support, which can overestimate the change caused by folding. A more direct measure of surface distortion, such as those used by Hoppe [12] or Cohen [7], might allow considerably more aggressive imperceptible simplification.

8.3 Extending the Perceptual Model

Like other perceptually based rendering algorithms to date (e.g., that of Ramasubramanian [23]), we base our decisions on the viewer's ability to perceive static stimuli. In animated or interactive rendering, we should also account for *temporal contrast sensitivity*: the ability to see a sudden change or flicker when a more gradual change would be invisible. In practice, our algorithms do not introduce visible flicker, but we could gaurantee this with a more sophisticated perceptual model that accounted for temporal contrast sensitivity. Such a model could prevent folds that would cause a visible "pop", or regulate a transition-softening technique such as alpha blending or Hoppe's *geomorphs* [12]. Unfortunately, temporal contrast sensitivity is less well studied than the static case, and incorporating current psychophysical models into our framework seems difficult. More research is needed in this area.

One significant factor affecting perceptibility of features is *contrast masking*, in which the presence of a high-contrast pattern can decrease sensitivity to features at other frequencies. For example, subtle discolorations on a brick wall may be less noticeable than on a uniformly colored brick-red wall. In polygonal models high-frequency patterns such as bricks are often represented using texture maps, whose frequency components could be computed in advance. In future work we hope to exploit contrast masking by patterns in texture maps for more aggressive simplification.

The velocity of a feature across the visual field also affects perception of detail. Consider, for example, a foreground object moving rapidly across a static background. If the user's eye tracks the moving object, the background will have high velocity and be simplified more aggressively. However, if the user's gaze remains fixed on the background, the moving object has high velocity and may be simplified. The perceptibility of moving contrast gratings may be related to that of static contrast gratings by a scaling function [15], so it should be straightforward to incorporate node velocity with respect to gaze into our folding criteria. Experiments by Reddy indicate the potential of using eccentricity and velocity together to guide LOD management [24], and integrating retinal velocity into our perceptual metrics seems an obvious and very promising avenue for future work.

8.4 Applicability of Gaze-Directed Rendering

Gaze-directed rendering is a powerful concept with some clear limitations. Accurately monitoring the user's gaze requires tracking the eye, but eye tracking is still emerging as a commodity technology: some current systems are fast enough, accurate enough, robust enough, and posses low enough latency for our application, but no existing eye tracker meets all of these needs at once. For example, our current system is quite fast and accurate, but restricts the user's head to a small volume, requires a room without sunlight, and involves a short calibration step before use.

It seems likely that eye-tracking technology will improve, eliminating these limitations. However, even without eye tracking gaze-directed rendering may still be a viable option. When allowed free range of head motion, user gaze is almost always restricted to ±15º of head direction [1]. We can thus substitute head direction for eccentricity in our system simply by subtracting a 15º error term. For multi-screen wide-angle displays, such as video wall or CAVE™ systems, head-tracked gaze-directed rendering may be a very attractive option.

Our system could easily handle multiple viewers by calculating the eccentricity of a node as the minimum distance to any viewer's gaze direction. Obviously, multiple viewers can reduce the impact of gaze-directed rendering, since viewers might examine different parts of the display at once. Such a scenario increases the demand on the eye-tracking system and limits the degree of simplification possible. In a multi-screen wide-angle display scenario, however, most of the scene will still be outside any viewer's fovea and therefore still eligible for aggressive simplification. Even with head tracking, which forces a more conservative estimate of eccentricity, we suspect that gaze-directed rendering will prove a powerful technique for managing rendering complexity in such situations.

9 Acknowledgements

Many of our models are provided courtesy of the Stanford 3-D Scanning Repository. The authors would like to thank the maintainers of the repository for making these excellent models public.

10 References

[1] Barnes, G. "Vestibulo-ocular function during coordinated head and eye movements to acquire visual targets", *Journal of Physiology*, 287, (1979).

[2] Barten, Peter. "The Square-Root Integral (SQRI): A new Metric to Describe the Effect of Various Display Parameters on Perceived Image Quality", In *Human Vision, Visual Processing, and Digital Display*, vol. 1077, Proceedings SPIE (1989)

[3] Bolin, Mark. and G. Meyer. "A Perceptually Based Adaptive Sampling Algorithm", *Computer Graphics*, Vol. 32 (SIGGRAPH 98).

[4] Campbell, F. and Gubisch, R. "Optical Quality of the Human Eye", *Journal of Pysiology*, 186 (1966)

[5] Campbell, F.W. and Robson, J.G. "An Application of Fourier Analysis to the Visibility of Contrast Gratings", *Journal of Physiology*, 187 (1968)

[6] Clark, James H. "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM*, Vol. 19, No 10, pp 547-554.

[7] Cohen, J, M. Olano, and D. Manocha. "Appearance-Preserving Simplification," *Computer Graphics*, Vol. 32 (SIGGRAPH 98).

[8] Cosman, M., and R. Schumacker. "System Strategies to Optimize CIG Image Content". *Proceedings Image II Conference* (Scotsdale, Arizona), 1981.

[9] Ferdwada, James, S. Pattanaik, P. Shirley, and D. Greenberg. "A Model of Visual Masking for Realistic Image Synthesis", *Computer Graphics*, Vol. 30 (SIGGRAPH 96).

[10] Funkhouser, Tom, and C. Sequin. "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments", *Computer Graphics*, Vol. 27 (SIGGRAPH 93).

[11] Heckbert, Paul, and M. Garland. "Survey of Polygonal Surface Simplification Algorithms", SIGGRAPH 97 course notes (1997).

[12] Hoppe, Hughes. "View-Dependent Refinement of Progressive Meshes", *Computer Graphics*, Vol. 31 (SIGGRAPH 97).

[13] Hutchinson, Thomas E. "Human-Computer Interaction Using Eye-Gaze Input", *IEEE Transactions on Systems, Man, and Cybernetics* Vol 19, No. 6 (Novermber 1989).

[14] Kelly, D.H. Spatial Frequency Selectivity in the Retina, *Vision Research*, 15 (1975)

[15] Koenderink, J.J. *et al.* "Perimetry of contrast detection thresholds of moving spatial sine wave patterns", *Journal of the Optical Society of America*, 68 (1978)

[16] Lindstrom, P. and Turk, G. "Image-Based Simplification", To appear in *ACM Transactions on Graphics*. Available as technical report GIT-GVU-99-49, Georgia Institute of Technology (1999).

[17] Lubin, Jeffery. "A Visual Discrimination Model for Imaging System Design and Evaluation", *Vision Models for Target Detection and Recognition*, E. Peli, ed. World Scientific (1995).

[18] Luebke, David, and C. Erikson. "View-Dependent Simplification of Arbitrary Polygonal Environments", *Computer Graphics*, Vol. 31 (SIGGRAPH 97).

[19] Luebke, David. "A Developer's Survey of Polygonal Simplification Algorithms", To appear in *IEEE Computer Graphics & Applications* (March 2001). Available as technical report CS-99-07, University of Virginia.

[20] Luebke, David. See *http://vdslib.virginia.edu*.

[21] Oshima, Toshikazu, H. Yamammoto, and H. Tamura. "Gaze-Directed Adaptive Rendering for Interacting with Virtual Space", *Proceedings of VRAIS 96* (1996).

[22] Puppo, Enrico, and R. Scopigno. "Simplification, LOD and Multiresolution—Principles and Applications", *Eurographics '97 Tutorial Notes*, PS97 TN4 (1997).

[23] Ramasubramanian, Mahesh, S. Pattanaik, and D. Greenberg. "A Perceptually Based Physical Error Metric for Realistic Image Synthesis", *Computer Graphics*, Vol. 33 (SIGGRAPH 99).

[24] Reddy, Martin. "Perceptually-Modulated Level of Detail for Virtual Environments", Ph.D. thesis, University of Edinburgh, 1997.

[25] Rovamo, J. and Virsu, V. "An Estimation and Application of the Human Cortical Magnification Factor", *Experimental Brain Research*, 37 (1979)

[26] Rusinkiewicz, S. and Levoy, M. "QSplat: A Multiresolution Point Rendering System for Large Meshes", *Computer Graphics*, Vol. 34 (SIGGRAPH 2000).

[27] Savoy, R.L. and McCann, J.J. "Visibility of low-spatial-frequency sine-wave targets: Dependence on number of cycles", *Journal of the Optical Society of America*, 65 (1975)

[28] Shirman, L., and Abi-Ezzi, S. "The Cone of Normals Technique for Fast Processing of Curved Patches", Computer Graphics Forum (*Proc. Eurographics '93*) Vol 12, No 3, (1993), pp 261-272.

[29] Xia, Julie and Amitabh Varshney. "Dynamic View-Dependent Simplification for Polygonal Models", *Visualization 96*.

[30] Zhang, Hansong, and K. Hoff. "Fast Backface Culling Using Normal Masks", *Proceedings of ACM Symposium on Interactive 3D Graphics* (1997).

University of Virginia Technical Report CS-99-33:

ROBUST VIEW-DEPENDENT SIMPLIFICATION FOR VERY LARGE-SCALE CAD VISUALIZATION

David P. Luebke

Department of Computer Science

University of Virginia, Thornton Hall

Charlottesville, VA 22903 (U.S.A.)

804-924-1021 (Work) 804-982-2214 (Fax)

ABSTRACT

View-dependent simplification (VDS) is a novel polygonal simplification algorithm uniquely suited to the interactive visualization of very large-scale CAD datasets. VDS adjusts the simplification continually according to view-dependent parameters such as the viewpoint position and orientation. As a result, objects can span several levels of detail, degrading smoothly from high fidelity where necessary to low fidelity where possible. VDS is also global, able to process the entire database without first decomposing the environment into individual objects. The resulting system enables interactive display of very complex polygonal CAD models consisting of thousands of parts and millions of polygons. VDS supports various preprocessing algorithms and various view-dependent criteria, providing a general framework for dynamic view-dependent simplification.

Keywords: *polygonal simplification, multiresolution methods, interactive visualization*

1 INTRODUCTION

Interactive visualization of very large-scale CAD models is an increasingly crucial problem. More and more enterprises and industries are embracing full-system CAD processes, producing and employing CAD models of unprecedented detail and completeness. These models span the spectrum of products from submarines to power plants, from airplanes to offshore oilrigs. Such large-scale CAD databases can serve as a single unified resource for simulation and design, cutting costs, streamlining the design process, and speeding up production. Interactive computer graphics is an integral component of the full-system CAD process, enabling the design, visualization, and manipulation of these datasets. Despite tremendous strides in computer graphics hardware, however, the growth in complexity of large-scale models continues to outstrip our capability to render them interactively. When converted to polygonal form for interactive rendering, today's large-scale models easily reach 100 million polygons, two to three orders of magnitude beyond what a high-end commercial graphics platform can render interactively. Tomorrow's models will undoubtedly measure billions of polygons. To achieve interactive rendering rates on such large-scale datasets clearly requires some algorithmic means of managing geometric complexity.

Polygonal simplification provides a powerful tool for managing this complexity. These techniques simplify the polygonal geometry of small, distant, or otherwise unimportant portions of the model, reducing the rendering cost without a significant loss of visual detail. This article presents a novel polygonal simplification approach uniquely suited to interactive rendering of very large-scale CAD databases.

1.1 Traditional Polygonal Simplification

Polygonal simplification is at once a very current and a very old topic in computer graphics. As early as 1976 James Clark described the benefits of representing objects within a scene at several resolutions, and flight simulators have long used hand-crafted multi-resolution models of airplanes to guarantee a constant frame rate [1,3]. Recent years have seen a flurry of research into generating such multi-resolution representations of objects automatically by simplifying the polygonal geometry of the object. Figure 1 illustrates

traditional polygonal simplification. Multiple versions of each object are created at progressively coarser *levels of detail* or *LODs* in a preprocess. At run time the system picks which LOD will represent the object based on criteria such as distance from the viewer.

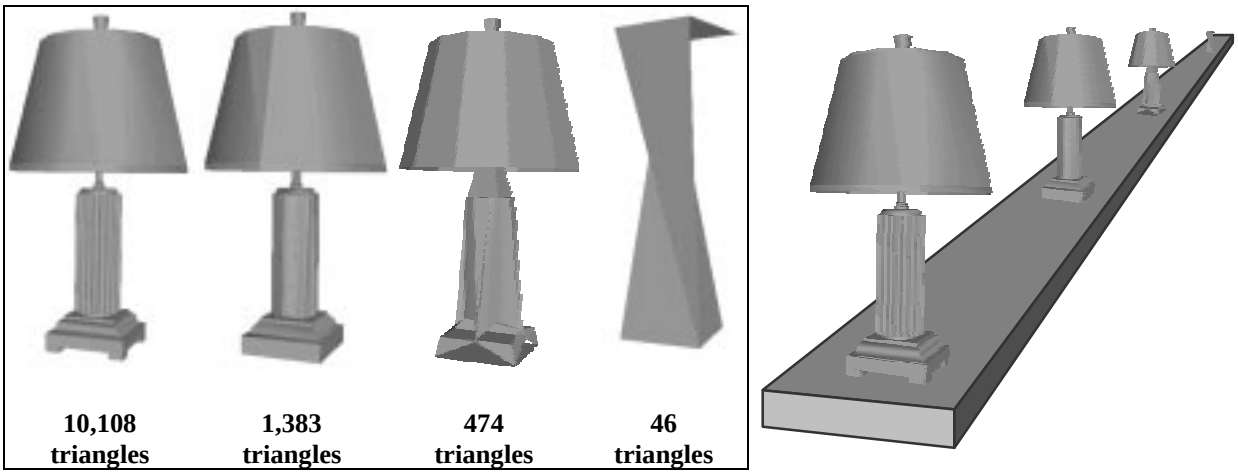


Figure 1: The traditional approach to polygonal simplification creates multiple levels of detail (LODs) of each object in a preprocess, and picks an LOD at run time based on distance.

1.2 Motivation for a Different Approach

This article describes an algorithm conceived for very large-scale CAD databases, a class of models for which earlier simplification methods often prove inadequate. Several features of such models make simplification a difficult task. To begin with, large-scale CAD models are by their nature handcrafted, often by many designers with different styles and levels of expertise. As a result, the models tend to be messy, often containing topological degeneracies of every sort. The sheer complexity of these models can also be daunting. Massive models consisting of thousands of parts and millions of polygons are not uncommon. Such massive CAD models often represent entire scenes rather than objects, and typically exhibit a high dynamic range, containing structural elements as large as the model as well as small, complex parts and assemblies.

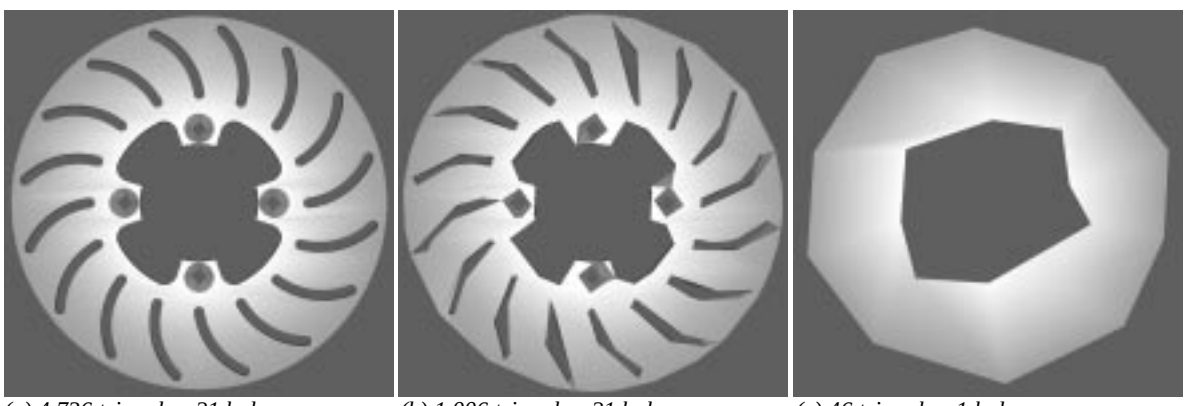
An ideal polygonal simplification algorithm for large-scale CAD models should possess a few key attributes. Such an algorithm should be:

Fast. Many traditional algorithms are quite slow, taking minutes or even hours to create LODs for a complex object. For models containing thousands of parts and millions of polygons, creating LODs becomes a batch process that can take hours or days to complete.

Depending on the application, such long preprocessing times may be a slight inconvenience or a fundamental handicap. In a design-review setting, for instance, CAD users may want to visualize their revisions in the context of the entire model several times a day. Preprocessing times of hours prevent the rapid turnaround desirable in this scenario.

Capable of topology reduction. Most traditional LOD algorithms both require and preserve manifold topology in the polygonal mesh. Requiring clean mesh topology hinders the usefulness of such algorithms on handcrafted CAD models, which as noted above often contain topological degeneracies. In addition, preserving mesh topology implies preserving the overall genus, which as Figure 2 shows, can limit the amount of simplification possible.

Capable of drastic simplification. Visualizing truly large-scale CAD datasets requires reducing the polygonal complexity of those datasets by three to four orders of magnitude. Traditional methods work on a per-object basis, creating separate levels of detail for each object in the model. As the next sections argue, this limits the amount of drastic simplification possible.



(a) 4,736 triangles, 21 holes *(b) 1,006 triangles, 21 holes* *(c) 46 triangles, 1 hole*

Figure 2: Preserving genus limits drastic simplification. The original model of a brake rotor (a) is shown simplified with a topology-preserving algorithm (b) and a topology-modifying algorithm (c). Rotor model courtesy Alpha_1 Project, University of Utah.

1.3 Drastic Simplification: the Problem With Large Objects

Creating multiple LODs per object is useless for very large objects. Such an object—for example, the terrain landform in an civil engineering project, or the hull of a ship in maritime CAD—presents a fundamental problem: parts of the object will always be near the viewer, while other parts will always be distant. If the object contains many polygons, using a highly detailed LOD would mean high fidelity but low frame rates and jerky motion; using a low

detail would provide smooth motion but terrible fidelity. The usual solution is to break up large objects by hand into smaller objects that can be simplified separately. This can entail a great deal of work, however, and gives rise to the problem of *cracks* between adjacent objects simplified to different levels of detail.

The solution presented here involves *view-dependent simplification*, in which the level of detail varies across the object according to interactive viewing parameters such as viewer position and orientation. The terrain landform and ship hull present no problem to a view-dependent simplification algorithm because only the portions of the object near the viewpoint need to be rendered in high detail. The bulk of the object can still be simplified drastically, rescuing frame rates while preserving visual fidelity.

1.4 Drastic Simplification: the Problem With Small Objects

Complex assemblies of many small objects present another problem for traditional per-object LOD. The diesel engine model shown in Figure 3, contains over two hundred small parts (and at that is not particularly detailed). Assume an excellent LOD algorithm, which can reduce with good fidelity each of these parts to a single cube: the entire assembly still requires over 2,400 triangles to render. From a distance, the whole engine may cover only a few pixels on the viewer's screen. In this situation a single, fifty-polygon, roughly engine-shaped block makes a better approximation than two hundred small cubes.

Our solution is to use a *global simplification* algorithm that treats the entire scene rather than individual objects within the scene. With knowledge about the entire scene, the algorithm can decide when to start combining the various parts of the diesel engine. At a low enough level of detail, the whole engine (and perhaps nearby portions of the walls and floor) can be merged and represented by that fifty-polygon block. Note that the idea of global simplification dovetails nicely with a view-dependent approach. Since view-dependence allows different portions of an object to be represented at different levels of detail, *the entire scene can be treated as a single all-inclusive object for view-dependent simplification.*

These points are important and bear repeating: for drastic simplification using per-object LOD algorithms, large objects must be subdivided and small objects must be combined. Doing this manually can mean a great deal of work. A global, view-dependent algorithm is

better suited to drastic simplification, and thus better suited for very large-scale CAD models, than the traditional approach of creating separate LODs for each object.

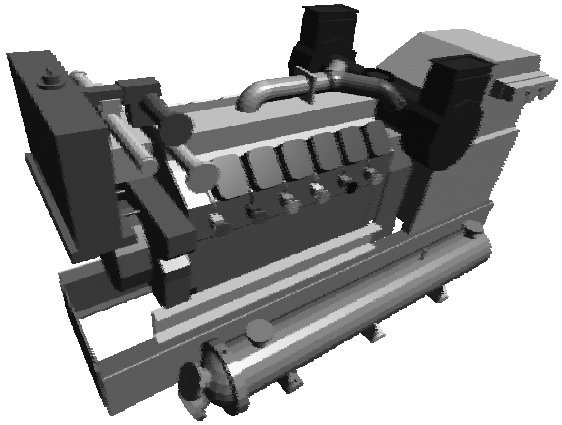


Figure 3: A diesel engine model with over 200 parts.
Courtesy Electric Boat Division, General Dynamics Corp.

1.5 The VDS algorithm

These considerations led to the algorithm presented in this article, called simply *view-dependent simplification* or *VDS*. VDS provides a framework for polygonal simplification via *vertex merging*. This operation, in which several polygon vertices are collapsed together into a single vertex, provides the fundamental mechanism for removing polygonal detail. Merging vertices that share an edge of a triangle makes that triangle redundant, allowing it to be removed. Note the use of *triangle* rather than *polygon*. The constant size and guaranteed planarity of triangles make them preferable to generic polygons, and like most simplification algorithms, VDS assumes that polygonal models have been fully triangulated.

VDS was designed explicitly to address the particular demands of large-scale CAD visualization. The algorithm permits global simplification, with a single large data structure comprising (if the user so desires) the entire model. This structure is the *vertex tree*, a hierarchy of vertex merge operations that encodes a continuum of possible levels of detail across the whole model. Applying a node's vertex merge operation collapses all of the vertices within the node together to a single vertex, eliminating triangles whose corners have been collapsed together. This is called *folding* the node. Likewise, a node may be *unfolded* by splitting that single vertex into the vertices of the node's children. Triangles filtered out when the node was folded become visible again when the node is unfolded, increasing the

triangle count. Figure 4 illustrates a simple two-dimensional example mesh and vertex tree, along with a sequence of folding operations.

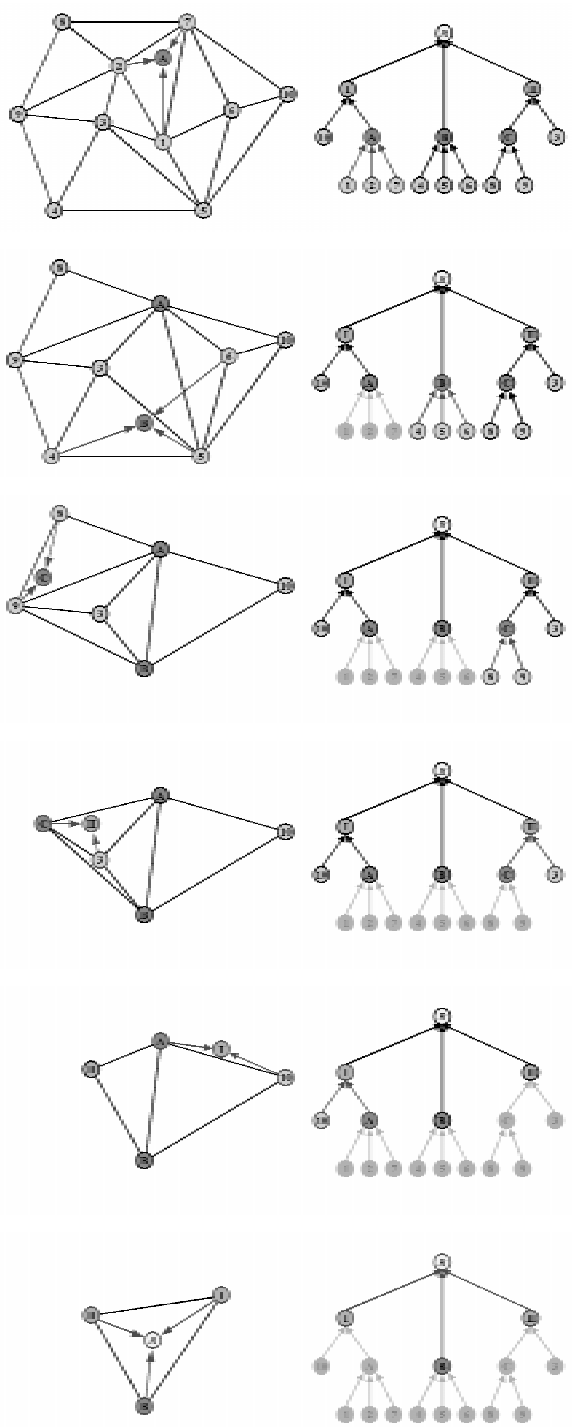


Figure 4: A sequence of fold operations. Folding each node removes some triangles from the scene, reducing the scene to a single triangle and finally to the root node *R*.

Note that the vertex tree contains information only about the vertices and triangles of the model. The algorithm makes no assumptions about the connectivity of those primitives. In particular, the triangles are not assumed to form a manifold mesh or approximate a smooth surface. This is another important feature of the VDS framework: because the simplification operates on the level of triangles and vertices rather than meshes and surfaces, manifold topology is not required and need not be preserved.

The entire system is dynamic and view-dependent. Nodes to be folded or unfolded are continually chosen at run-time based according to user-specified criteria. For example, a common criterion is to set a threshold on projected node screen size. In this mode, the user sets a screenspace-size threshold—say two pixels—before flying the viewpoint interactively around the model. The screenspace extent of each node is monitored: as the viewpoint shifts, certain nodes in the vertex tree will shrink in apparent size, falling below the two-pixel threshold. These nodes will be folded and redundant triangles removed from the scene. Other nodes will increase in apparent size and will be unfolded into their constituent child nodes, introducing new vertices and new triangles into the display list. Adjusting the threshold lets the user interactively control the degree of simplification and select the right balance of fidelity and performance.

Other criteria can be used to drive the system; VDS is less a particular algorithm than a general framework from which algorithms can be constructed. The only essential invariants of the VDSlib framework are the vertex tree and its associated methods (e.g., folding and unfolding nodes). Decisions such as how the vertex tree is constructed and which view-dependent criteria are used to fold and unfold nodes flesh out the framework into a specific algorithm.

2 STRUCTURES AND METHODS

2.1 The Vertex Tree

The VDS vertex tree spans the entire model, organizing every vertex of every polygon into one global hierarchy encoding all simplifications VDS can produce. Leaf nodes each represent a single vertex of the original model; internal nodes represent the merging of multiple vertices from the original model into a single vertex called the *proxy*. A proxy is

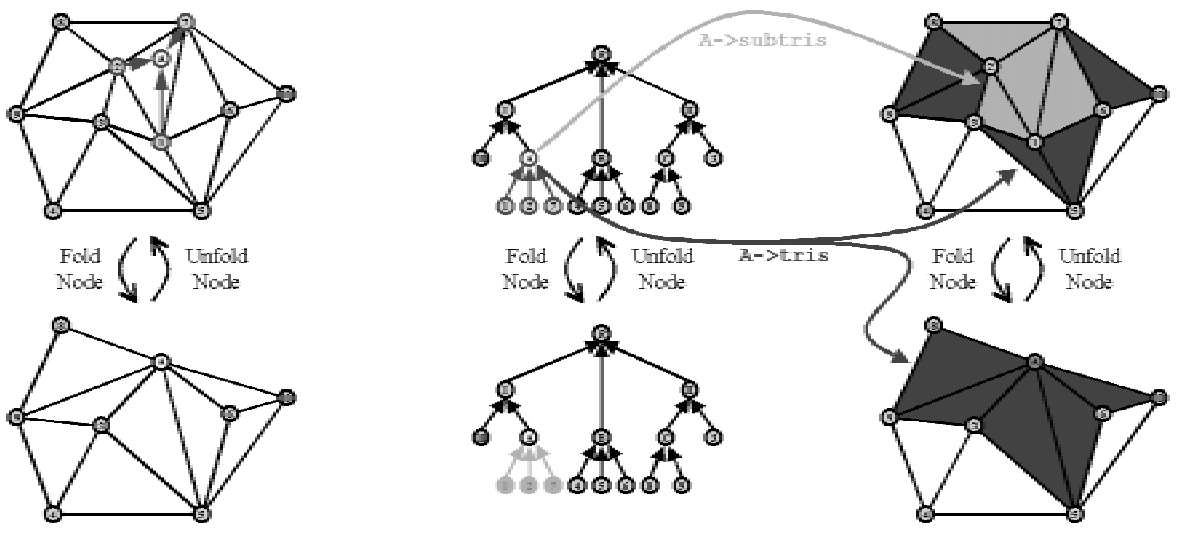
associated with each node in the vertex tree. We say a node **N** *supports* a vertex **V** of the original model if the leaf node representing **V** is a descendent of **N**. Each node in the vertex tree, then, supports a subset of the vertices in the original model; the root node supports every vertex of the entire model. A node supports a triangle **T** of the original model if it supports one or more of the vertices that form **T**'s corners.

Folding a node merges the vertices supported by the node into its proxy, and *unfolding* a node reverses the process. To define these terms more carefully, assume for simplicity that a node's children must all be folded before the node can be folded (since those children can first be folded recursively if necessary, this assumption does not limit the power of the fold operation). This requirement reduces the fold process from merging all vertices supported by a node to merging the proxies of that node's children. Similarly, unfolding a node assumes that the node's parent is unfolded, and splits a node's representative vertex into just the few representative vertices of the node's folded children. Defined this way, fold and unfold are local operations that make only incremental changes to the vertex tree.

Folding and unfolding a node always affects certain triangles. One set of triangles, called the node's relevant triangles or *reltris*, will change in shape as a corner shifts during fold and unfold operations. Another set of triangles, called the node's *subtris*, will disappear when the node is folded and reappear when the node is unfolded (Figure 5). Since reltris and subtris do not depend on the state of other nodes in the vertex tree, they can be computed offline and accessed very quickly at runtime.[1] This is the key observation behind VDS.

---

1 In fact, calculating reltris can be postponed until run-time or avoided altogether; these optimizations are discussed later.



(a) Nodes 1, 2, 7 merge to form A *(b) The local vertex tree* *(c) Reltris and subtris of node A*

Figure 5: Reltris and subtris of a node in the vertex tree. The highlighted node A represents the clustering of nodes 1, 2, and 7.

Unfolded nodes are labeled *active*; folded nodes are labeled *inactive*. During operation the active nodes constitute a cut of the vertex tree, rooted at the root node, called the *active tree*. Folded nodes with active parents are a special case; these nodes form the boundary of the active tree and are labeled *boundary* (Figure 6). Since the location of the boundary nodes determines which vertices in the original model have been collapsed together, the path of the boundary nodes across the vertex tree completely determines the current simplification. Notice that by definition, only boundary nodes can be unfolded and only active nodes whose children are all boundary nodes can be folded.

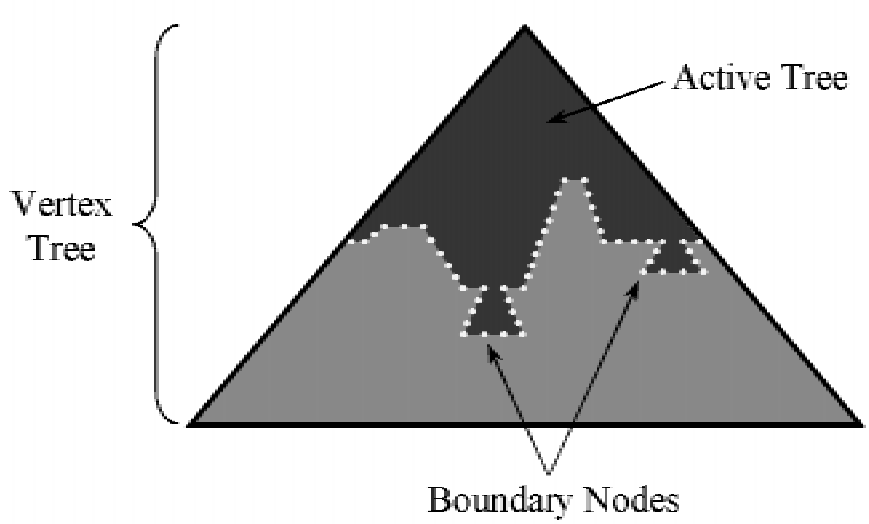


Figure 6: The vertex tree, active tree, and boundary nodes.

Each node in the vertex tree includes the basic structure described below; explanations of the individual fields follow.

```
struct Node {
   Int         depth;      // depth of the node in vertex tree
   NodeStatus  label;      // status: active, boundary, or inactive
   Coordinate  proxy;      // node's representative vertex
   BoundingVol bound;      // bounding volume of all tris supported
   Node *      parent;     // parent node
   Node *      children[]; // child nodes
   Tri *       reltris[];  // triangles that change shape upon folding
   Tri         subtris[];  // triangles that disappear completely
};
```

- `depth`: the depth of the node in the vertex tree.
- `label`: the node's status: *active, inactive, or boundary*.
- `proxy`: the coordinates of the node's representative vertex, to which all vertices represented by the subtree rooted at this node are collapsed.

`bound`: a bounding volume that contains all triangles supported by this node. For simplicity the current implementation uses spheres.

- `parent`: the parent of the node in the vertex tree
- `children`: a list of the node's children in the vertex tree.

`reltris`: a list of triangles with exactly one corner supported by the node. These are the triangles whose corners must be adjusted when the node is folded or unfolded.

`subtris`: a list of triangles with two or three corners supported by the node, but no more than one corner supported by each child of the node. These triangles will be filtered out if the node is folded, and re-introduced if the node is unfolded.

Note that for memory efficiency, fields such as `depth` and `label` can in fact be combined into a single field, and that lazy evaluation of triangle corners (described below) can be used to eliminate the `reltris` field entirely.

2.2 The Active Triangle List

While the vertex tree represents every simplification of the model possible in the VDS system, the *active triangle list* represents the current simplification being rendered. The chief

purpose of the active triangle list is to take advantage of temporal coherence. Frames in an interactive viewing session typically exhibit only incremental shifts in viewpoint, so the set of visible triangles remains largely constant. In its simplest form, the active triangle list is just a sequence of those visible triangles. Unfolding a node adds its subtris to the active triangle list; folding the node removes them. The active triangle list is maintained in the current implementation as a doubly-linked list of triangle structures, each with the following basic structure:

```
struct Tri {
   NodePath    corners[3];
   Node        *proxies[3];
   Tri         *prev, *next;
};
```

The `corners` field encodes the triangle at its highest resolution, referencing the three leaf nodes representing the original corners of the triangle. The `proxies` field represents the triangle in the current simplification, pointing to the *boundary ancestor* of each corner node. If a node **N** is inactive, its boundary ancestor is the boundary node on the path from **N** to the root. The `proxies` of a triangle in the active triangle list therefore encode the three nodes whose proxies currently represent the `corners` of the triangle.

Rather than referencing triangle `corners` directly via pointers, VDS uses the `NodePath` structure, a bit vector which stores the path to a node from the root of the vertex tree. In a binary vertex tree, for instance, each bit would represent a single branch; in an octree, each three-bit sequence would represent an 8-way branch. The *n*th element of the vector specifies which branch to take at level *n*. Referencing nodes in this fashion has advantages over simple pointers. First, simple bitwise operations can be used to compute the first common ancestor of two nodes, or to determine whether one node is a direct ancestor of another. Equivalent tests without the NodePath bit vector would involve hopping through the vertex tree traversing parent pointers. This will typically exhibit very poor memory coherence and correspondingly poor cache behavior.

A related—and perhaps more important—consideration comes into play when the vertex tree as a whole does not fit in memory, and must be paged in as necessary from disk. In this case accessing a triangle's original full-resolution corner nodes may be an extremely

expensive operation, to be avoided at all costs during runtime. Storing the path to each corner node enables the triangle `proxies` field to be updated during fold and unfold operations using purely local information, never referencing the original corner nodes.

2.3 Methods

The fundamental methods associated with the active triangle list are **`addTri()`**, **`removeTri()`**, and **`renderTriList()`**. A simple implementation uses a doubly-linked list with sentinels:

```
// Dummy sentinel structures start and end active triangle list:
Tri *startTriList, *endTriList;

addTri (Tri *T)
   // append to end of list
   T->next = endTriList;
   T->prev = endTriList->prev;
T->prev->next = T;

removeTri (Tri *T)
   // sentinels ensure prev & next fields won't be NULL
   T->next->prev = T->prev;
   T->prev->next = T->next;

RenderTriList ()
   Tri *T = startTriList->next;
   while (T != endTriList)
         renderTri(T);
         T = T->next;
```

Note that this scheme maintains the active triangle list entirely in place. All triangles in the model are kept in an array; as **`addTri()`** and **`removeTri()`** are called, they thread the doubly-linked list through the array. Though simple, this approach exhibits poor memory coherence: after a long series of **`addTri()`** and **`removeTri()`** calls, the linked list is likely to hop around the array of triangles seemingly at random. If the entire array does not fit into cache (or even into main memory), this can greatly degrade performance. Later we will discuss possible optimizations to avoid this problem.

The fundamental methods of the vertex tree are **`foldNode()`** and **`unfoldNode()`**. These functions add or remove the subtris of the specified node from the active triangle list, update the active boundary, and update the proxies of the node's reltris:

```
foldNode (Node *N)
   N->label = boundary;
   // all children should be labeled boundary; change to inactive
   foreach child C of N
         assert(C->label == boundary)
         C->label = inactive;
   // update tri proxies
   foreach triangle T in N->tris
         // which corner of T does N support?
         foreach corner c of {1,2,3}
               if (T->proxies[c]->parent == N) break;
         T->proxies[c] = N;
   // remove subtris from active triangle list
   foreach triangle T in N->subtris
         removeTri(T);


unfoldNode (Node *N)
   assert (N->label == boundary)
   foreach child C of N
         C->label = boundary;
   N->label = active;
   // update tri proxies
   foreach triangle T in N->tris
         // which corner of T is currently represented by N?
         foreach corner c of {1,2,3}
               if (T->proxies[c] == N) break;
         // which child of N supports T? Check NodePath in T->corners[]
         whichchild = T->corners[c][N->depth];
         T->proxies[c] = T->proxies[c]->children[whichchild];
   // add subtris to active triangle list
   foreach triangle T in N->subtris
         addTri(T);
```

3 View-Dependent Simplification Criteria

These structures and methods enable nodes to be folded or unfolded and triangles added or removed fast enough to respond to run-time events. Criteria for view-dependent simplification in this framework take the form of a function to choose which nodes are folded and unfolded each frame. This section describes three such criteria in our current implementation.

3.1 Screenspace Error Threshold

Our first goal was to remove small and distant triangles from the scene. To formulate this approach more precisely, consider a node in the vertex tree. Folding this node, which represents multiple vertices in the original model, clusters those vertices together into the node's proxy. The error introduced by collapsing the vertices can be thought of as the

maximum distance that a vertex can be shifted during the fold operation. This distance equals the length of the vector between the node's proxy and the clustered vertex farthest from the proxy. The extent of this vector when projected onto the screen is the *screenspace error* of the node. By unfolding exactly those nodes whose screenspace error exceeds a user-specified threshold *t*, VDS enforces a quality constraint on the simplification: no vertex shall move by more than *t* pixels on the screen.

Determining the exact screenspace extent of a vertex cluster can be a time-consuming task, but a conservative estimate can be efficiently obtained by associating a bounding volume with each node in the vertex tree. Our current implementation uses bounding spheres, which allow an extremely fast screenspace extent test but often provide a poor fit to the vertex cluster. The function **`nodeSize()`** tests the bounding sphere of a node and returns its extent on the screen as a fraction of viewport size. The recursive procedure **`adjustTree()`** uses **`nodeSize()`** in a top-down fashion, evaluating which nodes to fold and unfold. Nodes with extent greater than the threshold are unfolded and smaller nodes are folded:

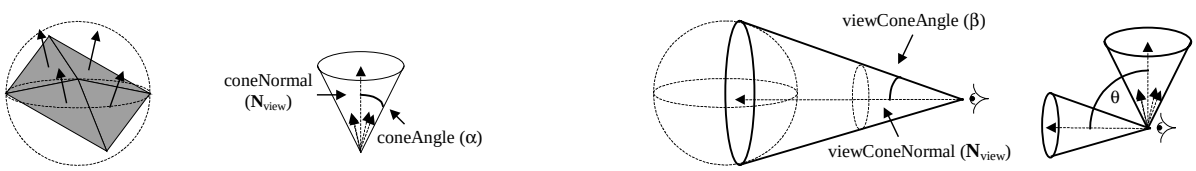
```
adjustTree (Node *N)
   size = nodeSize(N);
   if (size >= threshold)
         if (N->label == active)
               foreach child C of N
                     adjustTree(C);
         else  // N->label == boundary
               unfoldNode(N);
   else  // size < threshold
         foldSubtree(N);
```

The recursive function **`foldSubtree()`**, as the name suggests, folds the entire subtree rooted at node N:

```
foldSubtree (Node *N)
   if (node->label == active)
         foreach child C of N
               foldSubtree(C);
               foldNode(C);
```

3.2 Silhouette Preservation

Silhouettes and contours are particularly important visual cues for object recognition. Detecting nodes along object silhouettes and allocating more detail to those regions can therefore disproportionately increase the perceived quality of a simplification [15]. A conservative but efficient silhouette test can be plugged into the VDS framework by adding two fields to the `Node` structure: `coneNormal` is a vector and `coneAngle` is a floating-point scalar. These fields together specify a *cone of normals* [14] for the node, which bounds all the normals of all the triangles supported by the node. At run time a viewing cone is created that originates from the viewer position and tightly encloses the bounding sphere of the node (Figure 7). Testing the viewing cone against the cone of normals determines whether the node is completely frontfacing, completely backfacing, or potentially on the silhouette.



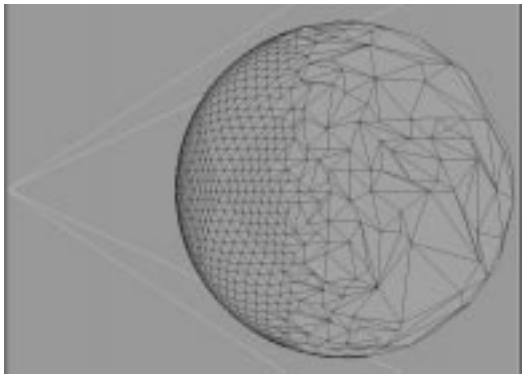
(a) A node containing four triangles, shown with its bounding sphere, and the node's cone of normals.
(b) The viewing cone originates from the viewer and tightly encloses the node's bounding sphere. The angle between N_{cone} and N_{view} is denoted θ.

Figure 7: Silhouette preservation. If any vector within the viewing cone is at right angles to any vector within the cone of normals, the node may be on the silhouette.

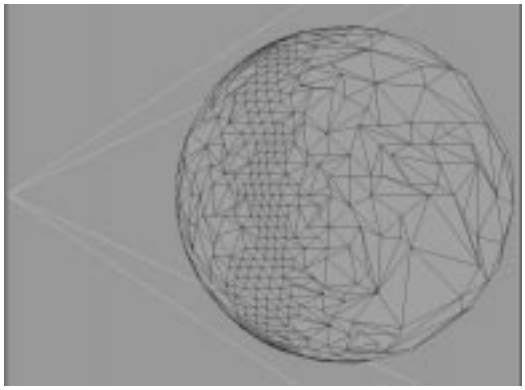
```
testSilhouette(Node *node, Coord eyePt)
   α = node->coneAngle;
   N_cone = node->coneNormal;
   β = calcViewConeAngle(eyePt, node);
   N_view = calcViewConeNormal(eyePt, node);
   θ = cos^-1(N_view • N_cone);
   if (θ - α - β > π/2)
         return FrontFacing;
   if (θ + α + β < π/2)
         return BackFacing;
   return OnSilhouette;
```

Silhouette preservation fits easily into the screenspace error metric approach presented above: the silhouette test determines which nodes may be on the silhouette, and these nodes are then tested against a tighter screenspace error threshold (Ts) than interior nodes (T_I).

Following Hoppe [8], we fold nodes that **`testSilhouette()`** evaluates as backfacing, aggressively simplifying portions of the model oriented away from the viewer. This is called *backface simplification*. The **`adjustTree()`** operation is easily modified to incorporate these tests; Figure 8 illustrates silhouette preservation and backface simplification.



(a) 1% error threshold, backface simplification enabled (3,388 faces)



(b) 1% silhouette error threshold, 20% interior error threshold (1,950 faces)

Figure 8: Silhouette preservation and backface simplification. The original model contains 8,192 faces.

3.3 Triangle Budget Simplification

The screenspace error threshold and silhouette test allow the user to set a bound on the fidelity of the simplified scene, but often a bound on the complexity (and thus rendering time) is desired instead. *Triangle budget simplification* allows the user to specify how many triangles the scene should contain. VDS then minimizes the maximum screenspace error of all boundary nodes within this triangle budget constraint. The intuitive meaning of this process is easily put into words: "Vertices on the screen can move as far as *t* pixels from their original position. Using no more than *n* triangles, minimize *t*."

We perform triangle budget simplification using a priority queue of boundary nodes, sorted by screenspace error. The node **N** with the greatest error is unfolded, removing **N** from the top of the queue and inserting the children of **N** back into the queue. This process iterates until unfolding the top node of the queue would exceed the triangle budget, at which point the maximum error has been minimized. Pseudocode for this procedure is straightforward, using a standard heap to implement the priority queue:

```
budgetSimplify(Node *rootnode)
// Initialize priority queue Q to contain just the rootnode
Heap *Q(rootnode);

while (Q->topnode->nsubtris < tribudget)
   unfoldNode(Q->topnode);
   // insert children, sorted by screenspace error:
   foreach child C of Q->topnode
         Q->insert(C);
   tribudget = tribudget - Q->topnode->nsubtris;
   Q->removeTopnode();
```

3.4 Related Work

View-dependent simplification criteria have been proposed by several researchers. Xia and Varshney [15], for instance, describe a novel criterion that uses local illumination information to preserve detail around regions such as specular highlights and shadow boundaries. The screen-space error threshold described above was initially presented in [10]. It is similar in principle to the uniform component of Hoppe's *deviation space* [8]. Deviation space is an ingenious error metric, evaluated at each node, that by its shape also incorporates a form of silhouette preservation. Deviation space thus provides an elegant solution to two view-dependent simplification criteria at once. The advantages of the simpler schemes presented here over Hoppe's deviation space are their speed and robustness, enabling fast preprocessing times and simplification of non-manifold meshes. For the specific domain of large-scale CAD datasets, with messy models and rapid turnaround times, these can be significant advantages indeed.

4 OPTIMIZING THE ALGORITHM

Our initial implementation ran at interactive rates on small models, on the order of 20,000 triangles. The current system has been demonstrated on models more than two orders of magnitude larger. Four sorts of optimization made this possible: exploiting temporal coherence, using visibility information, streamlining the math, and parallelizing the algorithm.

4.1 Exploiting Temporal Coherence

Interactive viewing sessions exhibit a high degree of frame-to-frame coherence, and VDS exploits this coherence throughout. The active triangle list, for example, is based on the assumption that relatively few triangles will be added or removed each frame. As Figure 9 shows, less than 2% of the triangles were added, deleted, or adjusted each frame during a typical path through the Torp model at a 5-pixel screenspace error threshold. The active triangle list exploits this temporal coherence by storing the unchanged triangles from frame to frame, and by supporting efficient add, delete, and update operations for the rest.

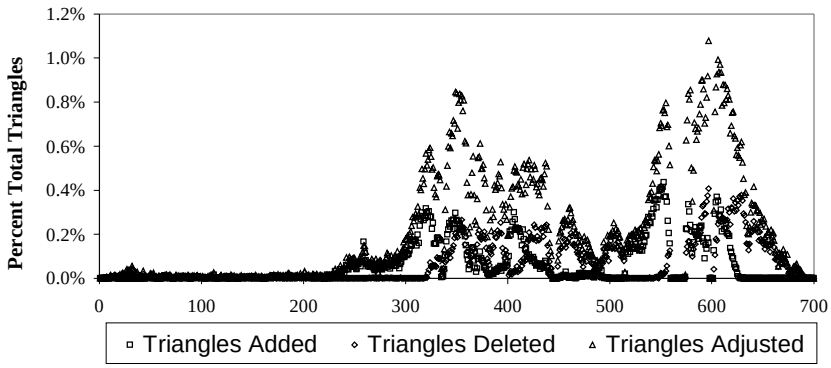


Figure 9: Triangles added, deleted, and adjusted during a 700-frame path through the 699,000-triangle Torp model, using a screenspace error thresholds of 5 pixels.

Vertex tree traversal can also profit from temporal coherence. Just as few triangles change status from frame to frame, few nodes in the vertex tree change status from frame to frame. Under these conditions the **`adjustTree()`** function is inefficient, visiting many nodes unnecessarily. A better scheme is to traverse, not down the vertex tree as **`adjustTree()`** does, but across the vertex tree along the path formed by boundary nodes. This path, called the *boundary path*, is maintained as a doubly-linked list by adding `prev` and `next` fields to the `Node` structure. In this way nodes far from the boundary are never considered and need not even be resident in memory. The function **`adjustPath()`** traverses the boundary path, folding and unfolding nodes as necessary:

```
adjustPath (Node *root)
   Node *currentNode;                  // node currently being tested
   Node *parentNode, *lastParent;      // parent of current & last node

   currentNode = root->next;
   parentNode = lastParent = NULL;
   repeat
         parentNode = currentNode->parent;
         if (parentNode != lastParent)
               lastParent = parentNode;
               // check parent's size first
               if (nodeSize(parentNode) < threshold)
                     // parent falls below threshold; fold
                     foldSubtree(parentNode);
                     currentNode = parentNode;
                     continue;
         // parent is fine, check current node
         if (nodeSize(current) >= threshold)
               // current node too large; unfold
               unfoldNode(current);
         current = current->next;
   until (current == root)
```

Note that `currentNode` is initialized to the root node. The boundary path actually forms a circular linked list, going through the root. This simplifies the maintenance of the boundary path in the **`foldNode()`** and **`unfoldNode()`** functions:

```
foldNode (Node *N)
   Node *pred, *succ;            // predecessor and successor nodes

   N->label = boundary;
   pred = N->children[0]->prev;  // set pred to N's first child's prev
   // all children should be labeled boundary; change to inactive
   foreach child C of N
         C->label = inactive;
         succ = C->next;
   // update tri proxies as before
   // remove subtris from active triangle list as before
   // adjust active boundary
   N->prev = pred;
   N->next = succ;
   N->prev->next = N;
   N->next->prev = N;
```

```
unfoldNode (Node *N)
   Node *pred = N->prev;
   Node *succ = N->next;

   foreach child C of N
         C->label = boundary;
         C->prev = pred;
         pred->next = C;
         pred= C;
   prev->next = succ;
   next->prev = pred;
   N->label = active;
   // update tri proxies as before
   // add subtris to active triangle list as before
```

4.2 Using Visibility Information

For many applications, most of the model is invisible most of the time. VDS can use this visibility information to reduce simplification as well as rendering time. For example, the process of quickly identifying and rejecting objects outside the visible field of view is called *view-frustum culling*. In applications such as architectural walkthroughs, view-frustum culling can greatly decrease rendering time by not rendering invisible portions of the model.

Efficient view-frustum culling in VDS requires modifying the active triangle list. Triangles are added to and removed from the list in a haphazard fashion as nodes are folded and unfolded, so triangles near each other in the model are unlikely to be near each other in the active triangle list. The solution is to impose *spatial coherence* by splitting the active triangle list into a hierarchy of lists, each representing a region of the complete model. Each triangle created by **`unfoldNode()`** is added to the appropriate list. View-frustum culling is applied to the lists themselves; the rendering process tests a bounding volume associated with each list, and skips any lists determined to be invisible.

The vertex tree provides a ready-made hierarchy in which to organize these multiple active triangle lists. Each triangle can be associated with a *cull node* that supports all three corners of the triangle. The cull node is invisible if its bounding volume lies outside the view frustum. Since that volume includes all three corners of the triangle, the triangle need not be rendered if its cull node is invisible. This property holds hierarchically: the descendants of an invisible node are themselves invisible. View-frustum culling, then, can be easily incorporated into VDS by creating an active triangle list for each node in the first few levels

of the vertex tree. When **`unfoldNode()`** creates a triangle, that triangle is added to the list of the appropriate node, which can be quickly calculated via bitwise operations on the triangle's corners.

Not rendering triangles contained by invisible nodes speeds rendering, but an invisible node may still support visible triangles (Figure 10). This fact gives rise to a stronger condition: some nodes are not only invisible but *irrelevant*, meaning that they support no visible triangles. An irrelevant node therefore cannot affect the visible scene, and the simplification traversal can choose to fold the node or simply ignore it. The large majority of invisible nodes are typically irrelevant, so testing for irrelevance provides a significant speedup. A simple test is to extend the node's bounding sphere to include all triangles supported by the node, storing an additional radius.

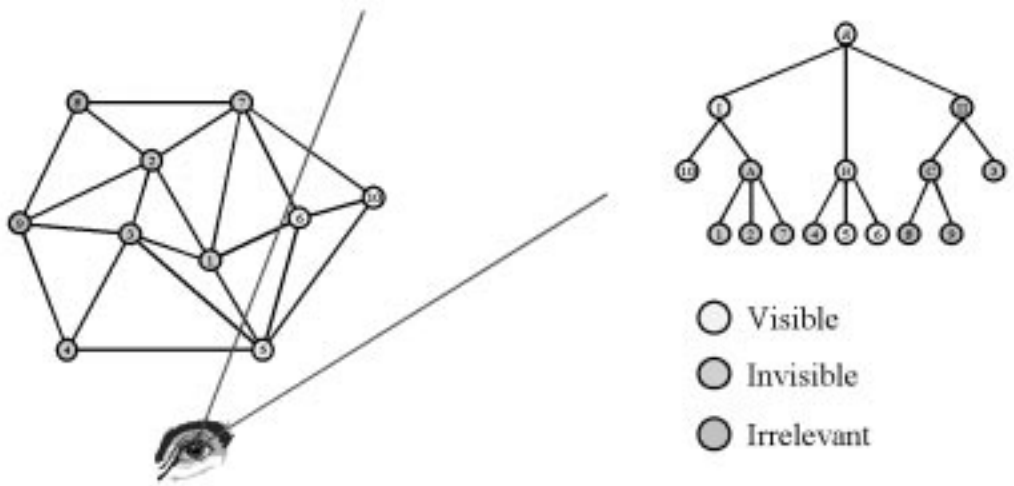


Figure 10: Visible, invisible, and irrelevant nodes. Invisible nodes lie outside the view frustum. Irrelevant nodes are invisible and support no vertices of visible triangles.

4.3 Streamlining the Math

Appropriate use of approximations and careful implementation can greatly streamline the computation involved in evaluating view-dependent criteria. For example, the **`nodeSize()`** function for evaluating screenspace error finds the extent of a cluster of vertices when projected onto the screen. An exact solution would presumably involve projecting the vertices (or their convex hull) and comparing the resulting screen coordinates, a dauntingly expensive operation. Since **`nodeSize()`** is typically called thousands of times per frame, an approximate solution based on bounding spheres is used instead. For a sphere with center ***c***

and radius r, seen from the eyepoint $\boldsymbol{e}$ with field-of-view angle φ, the fraction of viewport F occupied by the sphere is estimated by:

$$F = \frac{r}{\|\boldsymbol{c} - \boldsymbol{e}\| \tan(\varphi/2)} \tag{1}$$

Note that this approximation assumes that the sphere lies in the center of the field of view, and slightly underestimates F for nodes near the edges of the viewport. This fairly terse expression can be optimized still further in context. The function **`adjustTree()`**, for instance, compares each node's screenspace extent F to a user-specified threshold t. This amounts to evaluating the inequality:

$$F \geq t \tag{2}$$

which reduces to:

$$r \geq t\|\boldsymbol{c} - \boldsymbol{e}\| \tan(\varphi/2) \tag{3}$$

Squaring both sides and dividing by $\tan^2(\varphi/2)$ yields:

$$r^2 \cot^2(\varphi/2) \geq t^2 \|\boldsymbol{c} - \boldsymbol{e}\|^2 \tag{4}$$

The $\cot^2(\varphi/2)$ term is precalculated at the beginning of each frame. This expression is well suited for rapid evaluation, with no division or square root operations. Fixing the field-of-view angle throughout the viewing session enables further optimization by computing and storing the entire $r^2\cot^2(\varphi/2)$ term for each node, instead of just the radius r of the node's bounding sphere. Though these rearrangements may seem minor, this optimization alone more than tripled the speed of the simplification process in practice.

Here is the modified **`adjustTree()`** function, with silhouette tests omitted for clarity. The `threshold2` term, as the name suggests, holds the user-specified threshold, squared, and the new `r2cot2` field of the `Node` structure stores $r^2\cot^2(\varphi/2)$ for the node. Modifying the **`adjustPath()`** function along the same lines is straightforward.

```
adjustTree (Node *N)
   distance2 = (N->center[X] – eyept[X])² +
               (N->center[Y] – eyept[Y])² +
               (N->center[Z] – eyept[Z])²
   if (N->r2cot2 >= threshold2 * distance2)
         if (N->label == active)
               foreach child C of N
                     adjustTree(C);
         else  // N->label == boundary
               unfoldNode(N);
   else  // node size is below threshold
         foldSubtree(N);
```

4.4 Parallelization: Asynchronous Simplification

Computer graphics applications commonly parallelize by performing the major rendering stages concurrently in pipeline fashion. A traditional level-of-detail system might be divided into SELECT and RENDER stages: the SELECT stage decides which LOD of which objects to render in frame *n* and compiles them into a display list, while the RENDER process renders frame *n*-1 [4]. If *S* is the time taken to select LODs and *R* is the time taken to render a frame, performing the two processes as a pipeline reduces the total time per frame from *R*+*S* to ***max****(R,S)*.

VDS divides naturally into two basic tasks, SIMPLIFY and RENDER. The SIMPLIFY task traverses the vertex tree, folding and unfolding nodes as needed. The RENDER task loops over the active triangle list rendering each triangle. Let the time taken by SIMPLIFY to traverse the entire tree be *S* and the time taken by RENDER to draw the entire active list be *R*. The frame time of a uniprocessor implementation will then be *R*+*S*, and the frame time of a pipelined implementation will again be ***max****(R,S)*. The rendering task usually dominates the simplification task, so the effective frame time often reduces to *R*. The exception is during large shifts of viewpoint, when the usual assumption of temporal coherence fails and many triangles must be added and deleted from the active triangle list. This can have the distracting effect of slowing down the frame rate exactly when the user speeds up the rate of motion.

Asynchronous simplification provides a solution: let the SIMPLIFY and RENDER tasks run asynchronously, with the SIMPLIFY process writing to the active triangle list and the RENDER process reading it. This decouples the tasks for a total frame time of *R*,

eliminating the slowdown artifact associated with large viewpoint changes. When the viewer's velocity outpaces the simplification rate in asynchronous mode, the SIMPLIFY process simply falls behind. Typically, this results in a temporary coarsening of the scene quality. Under VDS, the portions of the scene near the viewer are refined to high detail whereas distant portions are simplified to coarse detail. If the user moves forward too quickly for the SIMPLIFY process to keep up, the viewpoint will leave the highly detailed region behind and move into a coarsely represented region. The scene rendered for the viewer remains coarse in quality until the SIMPLIFY process catches up, at which point the scene gradually refines back to the expected quality. This graceful degradation of fidelity is much less distracting than sudden drops in frame rate.

4.5 Lazy Evaluation of Triangle Corners

A final optimization worth mentioning reduces the space, rather than the time, required by VDS. Recall the `reltris` field of the VDS `Node` structure, which stores a list of triangles that must be adjusted when the node is folded or unfolded. We have discovered that this list can be eliminated by moving the update of triangle corners to the RENDER task, just before the triangle is rendered. Since this is the last possible moment for update, and since only triangles that must in fact be rendered are updated, we refer to this as *lazy evaluation of triangle corners*. In our experiments, evaluating triangle corners in lazy fashion slowed the rendering process by around 5%, while decreasing the memory requirements of the vertex tree by around 15%. The worth of this tradeoff depends on the bottlenecks of the application.

5 CONSTRUCTING THE VERTEX TREE

Thus far we have described the VDS vertex tree and its role in dynamic simplification, but have left open the question of how to construct the vertex tree in the first place. The vertex tree is completely determined by the order in which vertices are grouped. Once the hierarchical grouping of vertices is established, the matter of calculating subtris, bounding volumes, and so on becomes a purely mechanical process. How then do we perform this hierarchical vertex clustering?

The possible algorithms form a spectrum, ranging from fast, simple approaches whose resulting simplifications have moderate fidelity to slower, more sophisticated methods with

superb fidelity. The choice of algorithm for constructing the vertex tree is heavily application-dependent. In a design-review setting, CAD users may want to visualize their revisions in the context of the entire model several times a day. Preprocessing times of hours are unacceptable in this scenario. On the other hand, a walkthrough of the completed model might be desired for marketing purposes. Here it makes sense to use a slower, more careful algorithm to optimize the quality of simplifications and prevent any distracting artifacts. Since our goal is interactive visualization of very large, potentially messy CAD datasets with rapid turnaround, the clustering scheme we present here emphasizes speed and robustness above all.

5.1 Tight-Octree Vertex Clustering

The *octree* provides a simple top-down approach to vertex clustering. An octree is an 8-way tree in which each node represents an axis-aligned cube; the root node cube is created large enough to contain every vertex in the model. The root node is divided in half along the X, Y, and Z axes into 8 cubical subnodes, the vertices are partitioned among these eight children, and the process is recursively repeated for any subnode with more than one vertex. In this way, vertices are clustered roughly according to proximity. Neighboring vertices are likely to get clustered near the leaves of the tree, whereas distant vertices merge only at higher levels of the tree.

CAD models are often locally dense but globally sparse, consisting of highly detailed components separated by large areas of low detail or empty space. In this situation, a more adaptive partitioning structure is desired. The *tight octree* is a modified octree in which each node is tightened to the smallest axis-aligned cube that encloses the relevant vertices before the node is subdivided. This tightening ensures that every subdivision partitions the vertices, leading to more balanced trees with fewer nodes to traverse and store, and works very well in practice on all the CAD datasets we have tested.

Tight-octree clustering possesses many advantages. Its simplicity makes an efficient, robust implementation relatively easy to code. In addition, the spatial partitioning of vertices is very fast, bringing the preprocess time of even large models down to manageable levels. Preprocessing the 700,000-polygon torpedo room model, for example, takes only 108 seconds using a tight-octree clustering scheme. Finally, spatial-subdivision vertex clustering

is inherently very general. No knowledge of the polygon mesh is used; manifold topology is neither assumed nor preserved. In the CAD domain, meshes with degeneracies such as cracks, T-junctions, and missing polygons are regrettably common, but tight-octree vertex clustering can operate despite the presence of degeneracies incompatible with many schemes.

5.2 Related Work

Hoppe's work on view-dependent refinement of progressive meshes [8, 9] resembles the VDS algorithm presented here in many ways. A *progressive mesh* is a hierarchy of edge collapse operations similar in principle to a binary VDS vertex tree. The view-dependent criteria introduced by Hoppe to simplify the progressive mesh at run time have already been discussed. For construction of the progressive mesh, Hoppe uses a careful optimization approach that sorts possible edge collapse operations into a priority queue based on the error they add to the mesh [7]. This error is estimated by measuring the deviation of the simplified mesh from the original mesh at multiple points scattered across the local neighborhood.

The most fundamental difference between VDS and view-dependent refinement of progressive meshes is their underlying mechanism of merging vertices. VDS clusters arbitrarily many vertices at once to a single representative vertex, whereas progressive meshes use edge collapse operations that merge exactly two vertices sharing an edge in the mesh. As a result, the vertex hierarchy in a progressive mesh will always be binary, whereas the VDS vertex tree may in principle be *n*-ary[2]. Similarly, applying an edge collapse removes exactly two triangles from the mesh, whereas folding a VDS node may remove many triangles from the scene. Which scheme is better in general is unclear. The binary tree of a progressive mesh will be deeper than the corresponding VDS vertex tree, with many more nodes to traverse and store. On the other hand, the simple and regular structure of the edge collapse operation, which can be represented by a small, constant-size structure, lends itself to efficient storage and traversal. The finer granularity of the edge collapse could be an advantage, since a triangle budget can be specified very precisely, or a disadvantage, since more nodes must be processed to reach a desired level of simplification.

---

[2] In practice, of course, using a tight octree guarantees a vertex tree of maximum degree 8.

In the context of large CAD datasets, for which VDS was designed, the edge collapse operation has some definite disadvantages. Since edge collapses that create non-manifold regions are disallowed, holes in the mesh are not simplified and the genus of the object remains fixed. As argued above, this can limit the potential for drastic simplification of high-genus objects. Moreover, since only vertices that share an edge are merged, each object must be simplified separately, limiting the potential for drastic simplification of complex assemblies of objects. Furthermore, particularly CAD models may contain inherently non-manifold features, such as three triangles meeting at a single shared edge; algorithms based on edge collapses simply cannot represent such models. By clustering based on proximity, without regard to topology or source object, VDS deals well with all of these problem cases.

Once the tight octree has produced a vertex clustering, we use *quadric error metrics* (*QEMs*) to optimize the placement of the representative vertex. QEMs, introduced by Garland and Heckbert [6], provide a simple and fast technique for measuring the sum of the squared distances from a node's proxy to the planes of all of the triangles that node supports. Quadric error metrics require relatively little storage, handle non-manifold surfaces robustly, and produce simplifications of excellent fidelity. We use refinements to the basic QEM algorithm, in particular the surface-area preservation technique used by Erikson and Manocha in the GAPS algorithm [4].

6 RESULTS

All results reported here were obtained on a four-processor SGI Onyx^2 computer with 195 MHz R10K processors, 1152 megabytes of main memory, 4 megabytes of secondary cache, and InfiniteReality graphics.

6.1 Performance

Five sample models were chosen to span several CAD categories and a large range of polygon counts. The models and results are summarized in Table 1. *Engine* is a detailed model of an automobile engine containing over 140,000 triangles. *Cassini* is an aerospace CAD model of the Cassini space probe, provided courtesy of the Jet Propulsion Laboratory. It contains over 415,000 triangles. *AMR* depicts the auxiliary machine room of a notional nuclear submarine, containing approximately 505,000 triangles. *Torp* is another maritime

CAD dataset, representing the torpedo room of the same submarine with approximately 699,000 triangles. The Electric Boat Division of General Dynamics Corporation provided both submarine models. The smallest model, *bunny*, contains 70,000 triangles and comes from the Stanford 3-D Scanning Repository. Though it hardly qualifies as a CAD model, the bunny has become an unofficial benchmark for the polygonal simplification field.

Model	Category	Triangles	Preprocessing time	Vertex tree nodes	Vertex tree storage (w/ gzip)
Bunny	Scanned	69,451	7.2 seconds	50,856	5 Mb
Engine	Mechanical CAD	140,696	25 seconds	102,577	8 Mb
Cassini	Aerospace CAD	415,257	75 seconds	278,329	30 Mb
AMR	Maritime CAD	504,969	86 seconds	394,253	32 Mb
Torp	Maritime CAD	698,872	108 seconds	816,833	48 Mb

Table 1: Names, categories, and complexity of models in the VDS test suite.

6.2 Artifacts

Implementing asynchronous simplification is relatively straightforward, but care must be taken to avoid *dropouts*. Characterized by triangles that disappear for a frame, these transient artifacts occur when the RENDER process sweeps through a region of the active list being affected by the SIMPLIFY process. For example, the **`foldNode()`** operation removes triangles and fills the resulting holes by merging the corners of surrounding triangles. If those neighboring triangles have already been rendered during the frame when **`foldNode()`** adjusts their corners, but the triangle to be removed has not yet been rendered, a hole will appear in the mesh for that frame.

Dropouts prove difficult to eradicate using simple locking schemes without a significant performance penalty. One solution that works well is the *update queue*. Rather than performing the **`foldNode()`** and **`unfoldNode()`** operations, the SIMPLIFY process accumulates these updates into the update queue, marking the node Dirty and placing a Fold or Unfold entry in the queue. At the beginning of every frame, the RENDER process performs the updates in the queue, folding or unfolding each node before marking it Clean again. All changes to the active triangle list take place as a batch before any triangles are rendered; the shared database is thus kept consistent and dropouts are eliminated.

Another visual artifact that VDS can introduce is *mesh folding*. Mesh folding occurs when shifting the position of a vertex causes an attached triangle to flip in orientation (Figure 11). These artifacts are inherent to any vertex-merging or edge-collapse scheme that does not take care to avoid them. The visual effect of mesh folding depends on the rendering parameters. Folding a triangle flips its orientation, so such triangles may not be drawn if backface culling is enabled. If two-sided lighting is enabled, the triangle will be drawn, but since flipping a triangle negates its normal vector, the folded triangle may be shaded differently from the surrounding mesh.

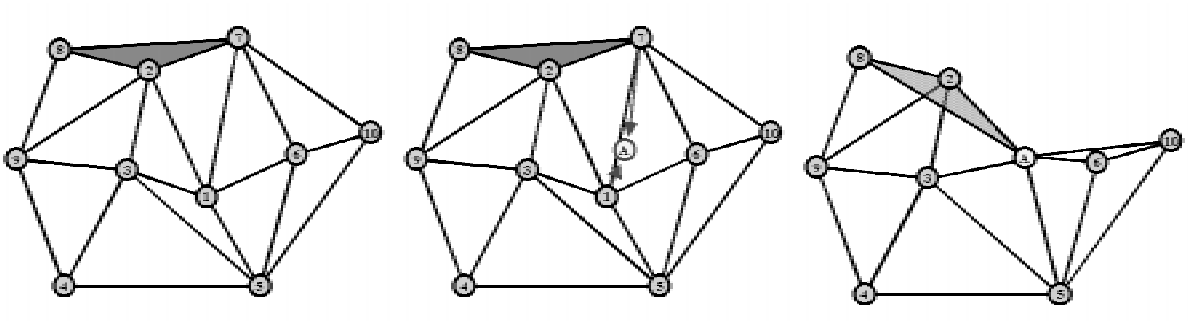


Figure 11: An example of mesh folding. When vertices 1 and 7 are merged to form vertex A, the shaded triangle folds over neighboring triangles, flipping in orientation.

Careful construction of the vertex tree can reduce the likelihood of mesh folding, but to completely eliminate folding artifacts requires additional view-dependent criteria. One possibility is to check the normal of each affected triangle; if a triangle normal is flipped, the fold operation is disallowed. Less expensive tests can be had by enforcing dependency constraints on the mesh, so that certain conditions must be met before a node may be folded or unfolded [8, 15]. Since the artifacts are small, and since additional view-dependent criteria might overly restrict simplification, the current VDS implementation does not attempt to prevent mesh folding. When high fidelity is a concern, however, adding code to prevent these artifacts would certainly be worthwhile.

6.3 Visual Results

Figure 12 shows an examples of the VDS system in action, comparing an original model to run-time simplifications created with various screenspace error tolerances.

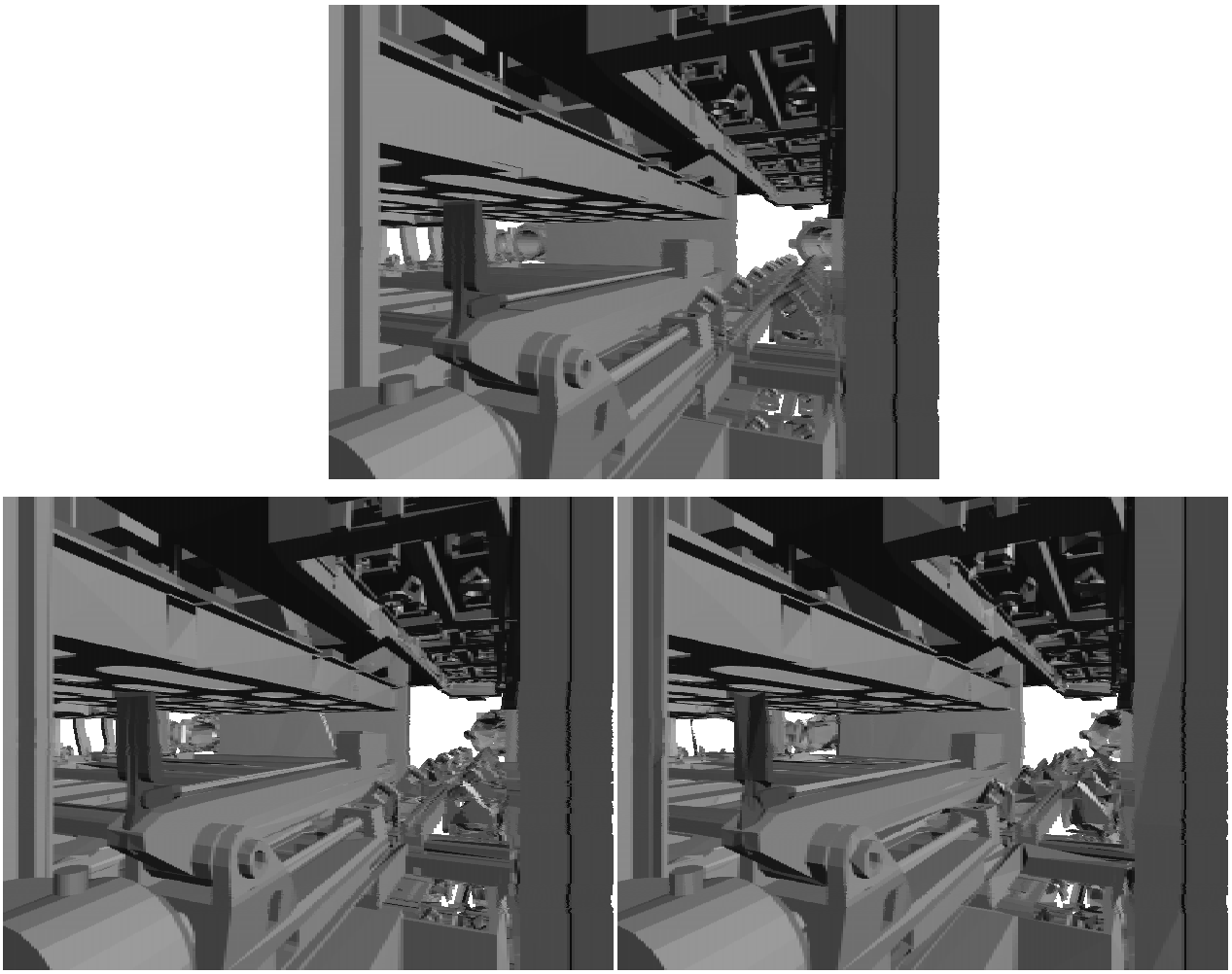


Figure 12: Top: The Torp model at original resolution comprises 698,872 faces.
Left: At 0.8% screenspace error (129,446 faces), visual artifacts are fairly subtle.
Right: At 1.5% screenspace error (76,404 faces), distant objects are simplified to almost schematic levels, while nearby features still possess reasonable fidelity.

7 SUMMARY AND FUTURE WORK

View-dependent simplification can provide a powerful, general framework for visualizing complex polygonal environments. We have described VDS, a view-dependent polygonal simplification algorithm particularly well suited to very large-scale CAD datasets. Such datasets are notoriously difficult for simplification; they are by nature extremely large, complex, and messy models. Key advantages of VDS include the ability to perform drastic simplification—despite the presence of very large objects, very small and numerous objects, or objects of high genus—and the ability to simplify arbitrary polygonal models despite non-manifold mesh topology.

VDS is also flexible, letting the user tailor the vertex tree construction algorithm and view-dependent simplification criteria to the application at hand. We have focused on the tight-octree vertex-clustering algorithm for constructing the vertex tree. The tight-octree is fast, robust, and completely automatic, making it well suited for large-scale CAD datasets with rapid turnaround. We have presented multiple view-dependent criteria for simplification, and discussed many optimizations to the basic VDS system.

The chief disadvantages of VDS are those of any view-dependent polygonal simplification scheme: an increased computational load on the CPU, and a mismatch to current graphics hardware, which is largely oriented towards retained-mode rendering.

Many interesting avenues of future work remain to be explored. One promising criterion for view-dependent simplification would adapt the work of Cohen [2] on *appearance preserving simplification*. Using successive mappings, this approach is able to bound, in screenspace, the distortion of texture maps during simplification. A similar technique can be applied to a *normal map*, which represents surface curvature just as a texture map captures surface color. Appearance preserving simplification thus provides a screenspace bound, not only on geometric deviation, but on deviation in coloration as well. View-dependent appearance-preserving simplification seems a promising area for future research.

Oshima [12] and Reddy [13] describe *gaze-directed simplification* systems, in which level of detail is regulated by the direction of a user's gaze. For example, an object in the center of a user's field of view would be allocated more detail than the same object in the periphery of the user's vision. Both Oshima and Reddy apply these criteria to selection of static LODs. We have begun to experiment with gaze-directed view-dependent simplification, and our initial results are encouraging.

The memory access patterns of VDS could be improved. For example, the current system implements the active triangle list as a doubly linked list to support efficient insert and delete operations. This list is maintained in place, threaded through an array of all triangles in the model. As described above, however, this tends to create a haphazard path that ruins cache coherence as it hops back and forth through the array. A better approach would collect all the active triangles into a single coherent array where they could be rendered with a simple linear pass.

These sort of memory-management issues touch on the larger topic of *out-of-core simplification*, in which the model to be rendered is far larger than main memory. VDS seems well suited to out-of-core simplification, since only the boundary path and active triangle list appear crucial to keep resident in memory. Research problems to be addressed include the out-of-core generation of a vertex tree. Hoppe [9] describes an interesting solution to this problem for the specific domain of terrain rendering; perhaps his approach could be generalized to arbitrary polygonal models.

One important area of future research is the question of how to simplify dynamic polygonal environments. Every simplification algorithm to date assumes that the models to be simplified are static, and must be run from scratch if the model changes. In an active CAD session, however, a designer builds a complex model with a series of incremental, often local changes. For example, many systems are based on constructive solid geometry (CSG) modeling, in which solids are defined by a series of Boolean operations upon simpler solids. Supported operations include union, intersection, and subtraction. A simplification system that supported efficient union, intersection, and subtraction operations upon VDS-style vertex trees could maintain a simplified representation of the model through incremental updates to the design. This should enable the designer to view a larger, more complex portion of the model interactively, which might provide more helpful context. Simplification of dynamic scenes is a challenging problem, and seems likely to be one of the next frontiers of polygonal simplification.

8 REFERENCES

1. Clark, James. "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM*, Vol. 19, No 10.

2. Cohen, Jon, and D. Manocha. "Appearance Preserving Simplification", *Computer Graphics*, Vol. 32 (SIGGRAPH 98).

3. Cosman, M., and R. Schumacker. "System Strategies to Optimize CIG Image Content". *Proceedings Image II Conference* (Scotsdale, Arizona), 1981.

4. Erikson, Carl, and D. Manocha. "GAPS: General and Automatic Polygonal Simplification", *Proceedings 1999 Symposium on Interactive 3-D Graphics* (Atlanta, Georgia), 1999.

5. Funkhouser, Tom, and C. Sequin. "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments", *Computer Graphics*, Vol. 27 (SIGGRAPH 93).

6. Garland, Michael, and P. Heckbert, "Simplification Using Quadric Error Metrics", Computer Graphics, Vol. 31 (SIGGRAPH 97).

7. Hoppe, Hughes. "Progressive Meshes", *Computer Graphics*, Vol. 30 (SIGGRAPH 96).

8. Hoppe, Hughes. "View-Dependent Refinement of Progressive Meshes", *Computer Graphics*, Vol. 31 (SIGGRAPH 97).

9. Hoppe, Hughes. “Smooth view-dependent level-of-detail control and its application to terrain rendering”, *IEEE Visualization `98*, October 1998.

10. Luebke, David, and C. Erikson. “View-Dependent Simplification of Arbitrary Polygonal Environments”, *Computer Graphics*, Vol. 31 (SIGGRAPH 97).

11. Luebke, David. “View-Dependent Simplification of Arbitrary Polygonal Environments”, Ph.D. thesis, Department of Computer Science, University of North Carolina, 1998.

12. Oshima, Toshikazu, H. Yamammoto, and H. Tamura. “Gaze-Directed Adaptive Rendering for Interacting with Virtual Space”, *Proceedings of VRAIS 96*.

13. Reddy, Martin. “Perceptually-Modulated Level of Detail for Virtual Environments”, Ph.D. thesis, University of Edinburgh, 1997.

14. Shirman, L., and Abi-Ezzi, S. "The Cone of Normals Technique for Fast Processing of Curved Patches", *Computer Graphics Forum*, Vol. 12, No 3, 1993.

15. Xia, Julie, and A. Varshney. “Dynamic View-Dependent Simplification for Polygonal Models”, *IEEE Visualization 96*, October 1996.

A Developer's Survey of Polygonal Simplification Algorithms

David P. Luebke
University of Virginia

This article surveys polygonal simplification algorithms, identifies the issues in picking an algorithm, relates the strengths and weaknesses of different approaches, and describes several published algorithms.

Polygonal models currently dominate interactive computer graphics. This is chiefly because of their mathematical simplicity: polygonal models lend themselves to simple, regular rendering algorithms that embed well in hardware, which has in turn led to widely available polygon rendering accelerators for every platform. Unfortunately, the complexity of these models—measured by the number of polygons—seems to grow faster than the ability of our graphics hardware to render them interactively. Put another way, the number of polygons we want always seems to exceed the number of polygons we can afford.

Polygonal simplification techniques (see Figure 1) offer one solution for developers grappling with complex models. These methods simplify the polygonal geometry of small, distant, or otherwise unimportant portions of the model, seeking to reduce the rendering cost without a significant loss in the scene's visual content. This is at once a very current and a very old idea in computer graphics. As early as 1976, James Clark described the benefits of representing objects within a scene at several resolutions,[1] and flight simulators have long used hand-crafted multiresolution airplane models to guarantee a constant frame rate. Recently, a flurry of research has targeted generating such models automatically. If you're considering using polygonal simplification to speed up your 3D application, this article should help you choose among the bewildering array of published algorithms.

The first questions

The first step in picking the right simplification algorithm is defining the problem. Ask yourself the following questions. (Note that Table 1 gives some informal and highly subjective recommendations for developers, organized according to the criteria I present in this section.)

Why do I need to simplify polygonal objects?

What's your goal? Are you trying to eliminate redundant geometry? For example, the volumetric isosurfaces generated by the marching cubes algorithm[10] tile the model's flat regions with many small, coplanar triangles. Merging these triangles into larger polygons can often decrease the model complexity drastically without introducing any geometric error. Similarly, you may need to subdivide a model for finite-element analysis; afterwards a simplification algorithm could remove unnecessary geometry.

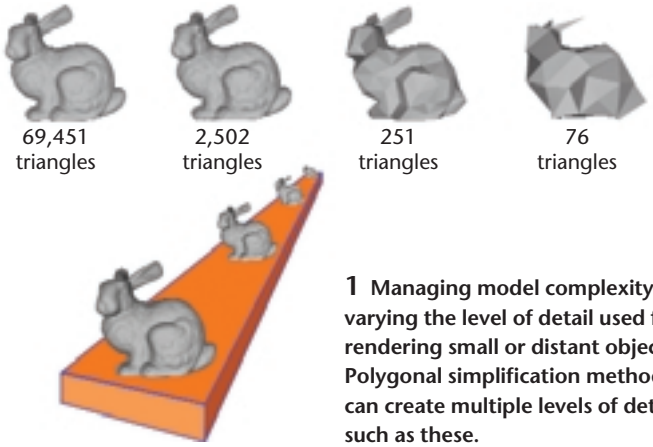
Or are you trying to reduce model size, perhaps creating downloadable models for a Web site? Here the primary concern becomes optimizing bandwidth, which means minimizing storage requirements. A simplification algorithm can take the original highly detailed model—whether created by a CAD program, laser scanner, or other source—and reduce it to a bandwidth-friendly level of complexity. If reducing the size required to store or transmit your 3D models is important, you should also investigate algorithms for geometric compression (see the "Further Reading" sidebar on p. 26).

Or are you trying to improve runtime performance by simplifying the polygonal scene being rendered? The most common use of polygonal simplification is to generate levels of detail (LODs) of the objects in a scene. By representing distant objects with a lower LOD and nearby objects with a higher LOD, applications from video games to CAD visualization packages can accelerate rendering and increase interactivity. Similar techniques let applications manage the rendering complexity of flying over a large terrain database. This leads naturally to the next important question.

0272-1716/01/$10.00 © 2001 IEEE

What are my models like?

No algorithm today excels at simplifying all models. Some approaches best suit curved, organic forms, while others work best at preserving mechanical objects with sharp corners, flat faces, and regular curves. Many models, such as radiositized scenes or scientific visualization data sets, have precomputed colors or lighting that must be considered. Some scenes, such as terrain data sets and volumetric isosurfaces from medical or scientific visualization, comprise a few large, high-complexity, individual objects. The monsters in a video game, on the other hand, might consist of multiple objects of moderate complexity, mostly in isolation. As a final example, an automobile engine CAD model involves large assemblies of many small objects. Which simplification algorithm you choose depends on which of these descriptions applies to your models.



1 Managing model complexity by varying the level of detail used for rendering small or distant objects. Polygonal simplification methods can create multiple levels of detail such as these.

What matters to me most?

Ask yourself what you care about in a simplification algorithm. Do you need to preserve and regulate geometric accuracy in the simplified models? According to what criterion? Some algorithms control the *Hausdorff distance* of the simplified vertices or surface to the original. (Informally, two point sets A and B are within Hausdorff distance *d* of each other if every point in A is within distance *d* of a point in B, and vice versa.) Other algorithms bound the volumetric deviation of the simplified mesh from the original. Some algorithms preserve the model's topological genus; others attempt to reduce the genus in a controlled fashion.[11,12]

Do you simply want high visual fidelity? This unfortunately is much harder to pin down; perception is more difficult to quantify than geometry. Nonetheless, some algorithms empirically provide higher visual fidelity than others do: one measures the simplification against rendered images[4] and another bounds, in pixels, the

Table 1. Assorted recommendations for "The First Questions" section.

Questions and Answers	Recommendation
Why do I need to simplify polygonal objects?	
Eliminate redundant geometry	Decimation[2] excels at this.
Reduce model size	For a one-shot application, use a high-fidelity algorithm like appearance-preserving simplification (APS)[3] or image-driven simplification (IDS).[4] Also consider geometry compression techniques (see the "Further Reading" sidebar).
Improve runtime performance (by managing levels of detail)	Depends, see below.
What are my models like?	
Complex organic forms	Decimation is often used, for example, for medical datasets, but quadric error metrics (QEM)[5] provide better fidelity at drastic rates.
Mechanical objects	Progressive meshes[6] for fidelity; vertex clustering for speed and simplicity.
Lots of textures or precomputed lighting	APS preserves fidelity best, with guaranteed bounds on deviation.
A few high-complexity objects	Use a view-dependent algorithm such as progressive meshes or hierarchical dynamic simplification (HDS).[7]
Multiple moderately complex objects	Use LODs. QEM is the best overall algorithm for producing LODs.
Large assemblies of many small objects	Merge objects into hierarchical assemblies using a topology-tolerant algorithm such as QEM or HDS.
So complex they don't fit in memory	Out-of-core simplification[8] is your best bet.
What matters to me most?	
Geometric accuracy	Use simplification envelopes (SE)[9] for manifold models, otherwise use QEM.
Visual fidelity	APS provides strong fidelity guarantees but is limited on most current hardware. IDS is driven by rendered images and has high visual fidelity.
Preprocess time	QEM provides high fidelity at high speed.
Drastic simplification	QEM if view-independent simplification suffices, otherwise use HDS.
Completely automatic	HDS works well for this.
Simple to code	Use publicly available code if possible, otherwise code up vertex clustering.

Further Reading

A number of excellent surveys review the field of polygonal simplification. For example, Cignoni et al.[1] supplied some comparative performance statistics, while Heckbert and Garland[2] and Puppo and Scopigno[3] summarized many of the methods not mentioned here. A tutorial by De Floriani et al.[4] provides a nice overview of LOD techniques in both surface and volume modeling.

Since polygonal simplification methods reduce the amount of geometry required to represent a model, they can are a form of geometry compression. Simplification provides lossy compression: simplified LODs require less memory to store and less bandwidth to deliver across a network, but at the cost of lower fidelity. Since 1995, researchers have proposed many geometry-compression techniques—both lossy and lossless—and the state of the art is rapidly advancing. Developers interested in reducing their 3D models' storage or bandwidth requirements may wish to investigate this burgeoning field. A full survey lies beyond the scope of this article, but a brief list of important papers follows (for more detail, see Taubin and Rossignac's excellent overview[5]):

- *Geometry compression*.[6] This seminal paper introduced the generalized triangle mesh representation, which caches the most recent *n* vertices for reference by triangles. The algorithm also optimizes the encoding of normals using a table-based approach and applies standard compression techniques to colors and coordinates.
- *Geometric compression through topological surgery*.[7] This work focuses on compressing connectivity and coordinate information using a vertex spanning tree. It forms the basis for 3D geometry compression in the MPEG-4 standard.
- *Progressive forest split compression*.[8] This algorithm combines aspects of progressive meshes and topological surgery, providing a highly efficient encoding of models that may be transmitted progressively across a network. Each stage of decompression doubles the number of vertices by splitting each vertex into two and stitching together the split regions according to an encoded triangulation.
- *Edgebreaker*.[9] This algorithm uses a finite-state machine to traverse and label triangles in a manifold mesh, compressing the triangle connectivity of zero-genus (that is, no holes) objects to less than 2 bits per triangle.
- *Progressive geometry compression*.[10] This recent and sophisticated progressive coding algorithm uses semiregular meshes, wavelet transforms, and zero-tree coding to achieve extremely high compression rates with excellent visual fidelity. The algorithm, designed for densely sampled meshes produced by geometry scanning, doesn't attempt to recreate the connectivity or vertex locations of the original mesh, focusing instead on the underlying surface's shape.

References

1. P. Cignoni, C. Montani, and R. Scopigno, "A Comparison of Mesh Simplification Algorithms," *Computers & Graphics*, vol. 22, no. 1, 1998, pp. 37-54.
2. P. Heckbert and M. Garland, "Survey of Polygonal Surface Simplification Algorithms," *Siggraph 97 Course Notes,* no. 25, ACM Press, New York, 1997.
3. E. Puppo and R. Scopigno, *Simplification, LOD, and Multiresolution-Principles and Applications*, tech. report C97-12, National Research Council of Italy, Pisa, Italy, 1997.
4. L. DeFlorani, E. Puppo, and R. Scopigno, "Level-of-Detail in Surface and Volume Modeling," *IEEE Visualization 98 Tutorials,* no. 6, IEEE CS Press, Los Alamitos, Calif., 1998.
5. G. Taubin and J. Rossignac, "3D Geometry Compression," *Siggraph 99 Course Notes,* no. 21, ACM Press, New York, 1999.
6. M. Deering, "Geometry Compression," *Computer Graphics* (Proc. Siggraph 95), vol. 29, ACM Press, New York, 1995.
7. G. Taubin and J. Rossignac, "Geometric Compression through Topological Surgery," *ACM Trans. Graphics*, vol. 17, no. 2, 1998, pp. 84-115.
8. G. Taubin et al.,"Progressive Forest Split Compression," *Computer Graphics* (Proc. Siggraph 98), vol. 32, ACM Press, New York, 1998, pp. 123-132.
9. J. Rossignac, "Edgebreaker: Connectivity Compression for Triangle Meshes," *IEEE Trans. Visualization and Computer Graphics*, vol. 5, no. 1, 1999, pp. 47-61.
10. A. Khodakovsky, P. Schroder, and W. Sweldens, "Progressive Geometry Compression," *Computer Graphics* (Proc. Siggraph 2000), vol. 34, ACM Press, New York, 2000, pp. 271-278.

visual disparity between an object and its simplification.[3]

Is preprocess time an issue? For models containing thousands of parts and millions of polygons, creating LODs becomes a batch process that can take hours or days to complete. Depending on the application, such long preprocessing times may be a slight inconvenience or a fundamental handicap. In a design-review setting, for instance, CAD users may want to visualize their revisions in the context of the entire model several times a day. Hours of preprocessing prevent the rapid turnaround desirable in this scenario. On the other hand, when creating LODs for a video game or a product demonstration, it makes sense to take the time necessary to get the highest quality simplifications.

If runtime performance is crucial or your models are extremely complex, you may need an algorithm capable of drastic simplification. As the following sections explain, drastic simplification may require drastic measures such as view-dependent simplification and topology-reducing simplification. If you need to simplify many different models, a completely automatic algorithm may be most important. Perhaps—let's face it—you just want something simple to code. Once you decide what matters to you most, you're ready to pick an algorithm.

Different kinds of algorithms, and why you'd want to use them

The computer graphics literature is replete with excellent simplification algorithms. Researchers have proposed dozens of approaches, each with strengths and weaknesses. Here, I attempt a useful taxonomy for simplification algorithms, listing some important ways algorithms can differ or resemble each other and describing what these mean to the developer.

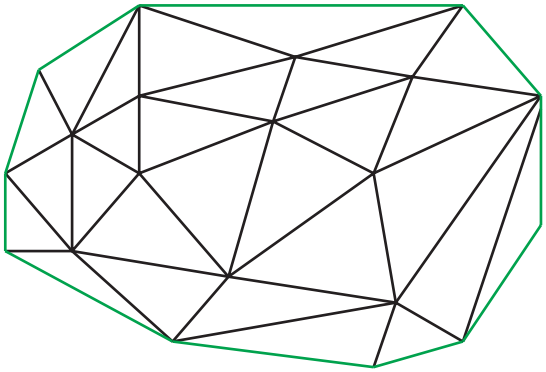
Topology

The treatment of mesh topology during simplification provides an important distinction among algorithms. First, let me introduce a few terms. In the context of polygonal simplification, *topology* refers to the connected polygonal mesh's structure. The *genus* is the number of holes in the mesh surface. For example, a sphere and a cube have a genus of zero, while a doughnut and a coffee cup have a genus of one. The *local topology* of a face, edge, or vertex refers to the connectivity of that feature's immediate neighborhood. The mesh forms a *2D manifold* if the local topology is everywhere equivalent to a disc—that is, if the neighborhood of every feature consists of a connected ring of polygons forming a single surface (see Figure 2). In a triangulated mesh displaying manifold topology, exactly two triangles share every edge, and every triangle shares an edge with exactly three neighboring triangles. A *2D manifold with boundary* permits boundary edges, which belong to only one triangle.

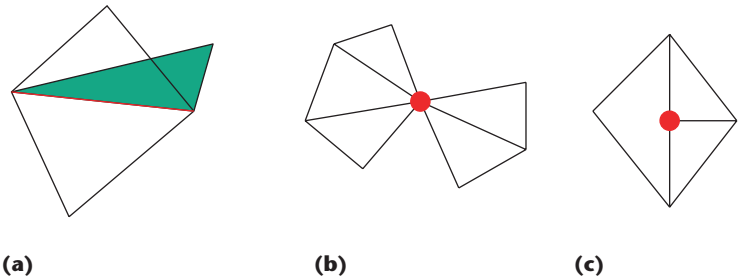
Manifold meshes result in well-behaved models. Virtually any simplification algorithm can successfully operate on any manifold object. Manifold meshes are also desirable for many other applications, such as finite-element analysis and radiosity. Some algorithms and modeling packages guarantee manifold output. For example, the marching-cubes algorithm constructs manifold volumetric isosurfaces. Unfortunately, in actual practice many models aren't perfectly manifold, with topological flaws such as cracks, T-junctions, and nonmanifold points or edges (see Figure 3). Such defects are particularly problematic in CAD, which by definition involves handmade models.

A *topology-preserving* simplification algorithm preserves manifold connectivity at every step. Such algorithms don't close holes in the mesh and therefore preserve the overall genus. Because no holes appear or disappear during simplification, the simplified object's visual fidelity tends to be relatively good. This constraint limits the simplification possible, however, since objects of a high genus can't be simplified below a certain number of polygons without closing holes in the model (see Figure 4). Moreover, a topology-preserving approach requires beginning with a mesh with manifold topology. Some algorithms are *topology tolerant*—they ignore regions in the mesh with nonmanifold local topology, leaving those regions unsimplified. Other algorithms, faced with nonmanifold regions, may simply fail.

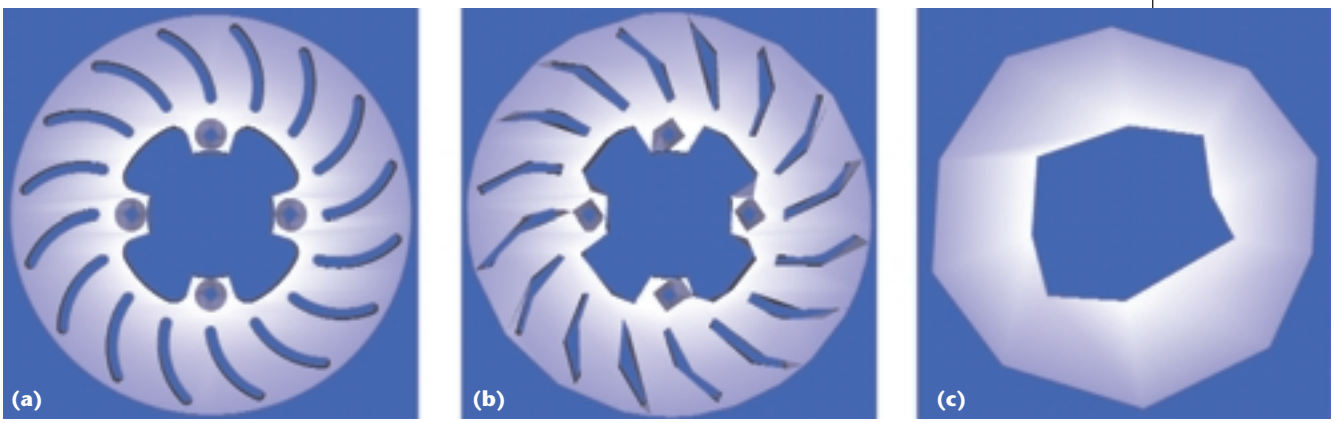
Topology-modifying algorithms don't necessarily preserve manifold topology. The algorithms can therefore close up holes in the model and aggregate separate objects into assemblies as simplification progresses, permitting drastic simplification beyond the scope of topology-preserving schemes. This drastic simplification



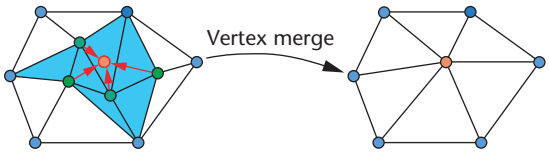
2 A 2D manifold with a boundary (boundary edges in green). One or two triangles share each edge and a connected ring of triangles shares each vertex.



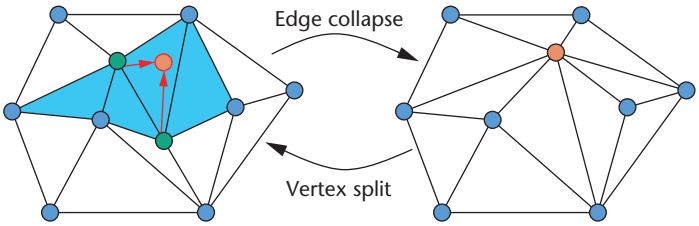
3 Examples of nonmanifold meshes: (a) An edge shared by three triangles, (b) a vertex shared by two otherwise unconnected sets of triangles, and (c) a T-vertex.



4 Preserving genus limits drastic simplification. The original model of a brake rotor with (a) 4,736 triangles and 21 holes is simplified with a topology-preserving algorithm using (b) 1,006 triangles and 21 holes and a topology-modifying algorithm with (c) 46 triangles and one hole. Model courtesy of the Alpha_1 Project, University of Utah.



5 The vertex-merge operation clusters multiple vertices (green) together into a single representative vertex (orange), eliminating those triangles (aqua) whose corner vertices are merged.



6 The edge-collapse operation merges exactly two vertices that share an edge. This eliminates two triangles from the mesh (one if the edge lies on a boundary). A vertex split is the dual of an edge collapse, introducing two triangles.

often comes at the price of poor visual fidelity, with distracting popping artifacts caused by holes appearing and disappearing from one LOD to the next. Some topology-modifying algorithms don't require valid topology in the initial mesh, which greatly increases their utility in real-world CAD applications. Some topology-modifying algorithms attempt to regulate the change in topology but most are *topology insensitive*, paying no heed to the initial mesh connectivity.

As a rule, topology-preserving algorithms work best when visual fidelity is crucial or with an application such as finite-element analysis, in which surface topology can affect results. Preserving topology also simplifies some applications, such as multiresolution surface editing, which require a correspondence between an object's high- and low-detail representations. Real-time visualization of complex scenes, however, requires drastic simplification and here topology-modifying algorithms have the edge. Either way, pick a topology-tolerant algorithm unless you're certain that your models will always have valid manifold topology.

Mechanism

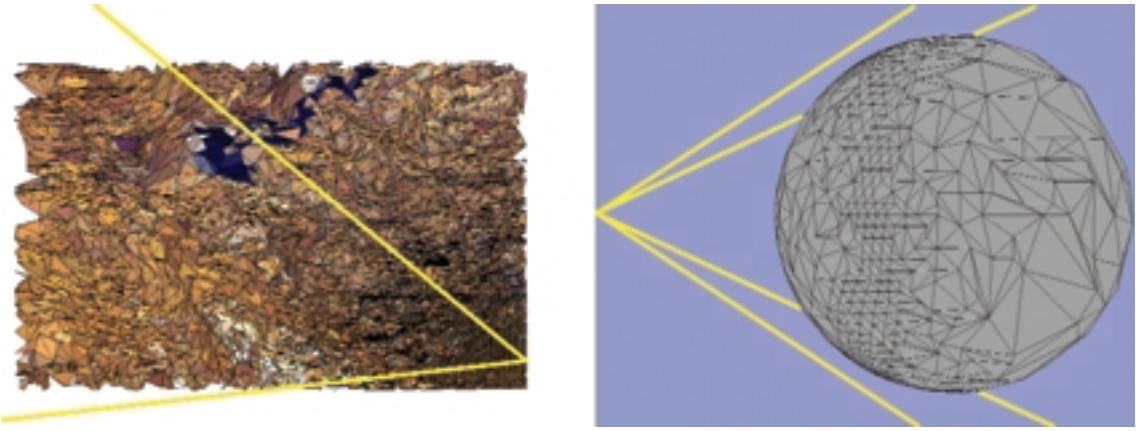
Nearly every simplification technique in the literature uses some variation or combination of four basic polygon removal mechanisms: sampling, adaptive subdivision, decimation, and vertex merging. Because the mechanism you use may affect an algorithm's characteristics, these are worth a few comments.

- Sampling algorithms sample the initial model's geometry, either with points on the model's surface or voxels superimposed on the model in a 3D grid. These are among the more elaborate and difficult to code approaches. They may have trouble achieving high fidelity since high-frequency features are inherently difficult to sample accurately. These algorithms usually work best on smooth organic forms with no sharp corners.
- Adaptive subdivision algorithms find a simple *base mesh* that can be recursively subdivided to closely approximate the initial model. This approach works best when the base model is easily found. For example, the base model for a terrain is typically a rectangle. Achieving high fidelity on general polygonal models requires creating a base model that captures the original model's important features, which can be tricky. Adaptive subdivision methods preserve the surface topology, which may limit their capacity for drastic simplification. On the other hand, they suit multiresolution surface editing well, because changes made at low levels of subdivision propagate naturally to higher levels.
- Decimation techniques iteratively remove vertices or faces from the mesh, retriangulating the resulting hole after each step. These algorithms are relatively simple to code and can be very fast. Most use strictly local changes that tend to preserve the genus, which again could restrict drastic simplification ability, but these algorithms excel at removing redundant geometry such as coplanar polygons.
- Vertex-merging schemes operate by collapsing two or more vertices of a triangulated model together into a single vertex, which in turn can be merged with other vertices. Merging a triangle's corners eliminates it, decreasing the total polygon count (see Figure 5). Vertex merging is a simple and easy-to-code mechanism, but algorithms use techniques of varying sophistication to determine which vertices to merge in what order. Accordingly, vertex-merging algorithms range from simple, fast, and crude to complex, slow, and accurate. Edge-collapse algorithms (see Figure 6), which always merge two vertices sharing an edge, tend to preserve local topology, but algorithms permitting general vertex-merge operations can modify topology and aggregate objects, enabling drastic simplification of complex objects and assemblies of objects

Static, dynamic, and view-dependent simplification

The traditional approach to accelerating rendering with polygonal simplification creates several discrete versions of each object in a preprocess, each at a different level of detail. At runtime, rendering algorithms choose the appropriate LOD to represent the object. Because distant objects use much coarser LODs, the total number of polygons is reduced and rendering speed increased. Because we compute LODs offline during preprocessing, this approach can be called *static polygonal simplification*.

Static simplification has many advantages. Decoupling simplification and rendering makes this the simplest model to program. The simplification algorithm generates LODs without regard to real-time rendering constraints, and the rendering algorithm simply chooses which LODs to render. Furthermore, modern graphics hardware lends itself to the multiple model versions created by static simplification, because each LOD can be converted during preprocessing to triangle strips and compiled as a separate display list. Rendering such dis-



7 Two examples of view-dependent simplification, with the view frustum in yellow. A high-resolution terrain near the viewer is simplified aggressively as distance increases (left). A sphere is simplified aggressively in interior and backfacing regions, while high fidelity is preserved along the silhouette (right).

play lists will usually be much faster than rendering the LODs as an unordered list of polygons.

Dynamic polygonal simplification departs from the traditional static approach. Whereas a static simplification algorithm creates individual LODs during the preprocessing stage, a dynamic simplification system creates a data structure encoding a continuous spectrum of detail. The desired LOD can be extracted from this structure at runtime. A major advantage of this approach is better granularity. Since the algorithm specifies the LOD for each object exactly, rather than choosing from a few precreated options, it uses no more polygons than necessary. This frees up more polygons for rendering other objects. Better granularity thus leads to better use of resources and higher overall fidelity for a given polygon count. Dynamic simplification also supports progressive transmission of polygonal models, in which a base model is transmitted followed by a stream of refinements to be integrated dynamically.[6]

View-dependent simplification extends dynamic simplification by using view-dependent criteria to select the most appropriate LOD for the current view. In a view-dependent system, a single object can span multiple levels of simplification. For instance, nearby portions of the object may appear at a higher resolution than distant portions, or silhouette regions of the object may appear at a higher resolution than interior regions (see Figure 7). By allocating polygons where they're most needed, view-dependent simplification optimizes the distribution of this scarce resource.

Indeed, complex models representing physically large objects often can't be adequately simplified without view-dependent techniques. Terrain models are a classic example. Large terrain databases are well beyond the interactive rendering abilities of even high-end graphics hardware, but creating traditional LODs doesn't help. The viewpoint is typically quite close to part of the terrain and distant from other parts, so a high LOD will provide good fidelity at unacceptable frame rates, while a low LOD will provide good frame rates but terrible fidelity. Breaking up the terrain into smaller chunks, each comprising multiple LODs, addresses both problems but introduces discontinuities between chunks. These discontinuities appear as cracks when two adjacent chunks are represented at different LODs. A view-dependent simplification system, however, can use a high LOD to represent the terrain near the viewpoint and a low LOD for parts distant, with a smooth degradation of detail between. Not surprisingly, early work on view-dependent simplification focused on terrains.[13]

The static simplification approach of creating multiple discrete LODs in a preprocess is simple and works best with most current graphics hardware. Dynamic simplification supports the progressive transmission of polygonal models and provides better granularity, which in turn can provide better fidelity. View-dependent simplification can provide even better fidelity for a given polygon count and can handle models (such as terrains) containing complex individual objects that are physically large with respect to the viewer. An obvious disadvantage of view-dependent systems is the increased runtime load of choosing and extracting an appropriate LOD. If the rendering system is CPU bound, this additional load will decrease the frame rate, cutting into the speedup provided by regulating LODs.

A brief catalog of algorithms

Several published algorithms follow, each classified according to their underlying mechanism; how they treat topology; and whether they use static, dynamic, or view-dependent simplification. In some cases, I describe a family of algorithms, with reference to papers that improve or extend the original published algorithm. The intent of this section isn't to provide an exhaustive list of work in the field of polygonal simplification, nor to select the best papers, but to briefly describe a few important algorithms that span the taxonomy presented. You may choose to implement one of the algorithms here (or download the code, if available), choose another algorithm from the literature, or come up with your own. Hopefully, this article will help you make an informed decision.

The decimation algorithm, designed to reduce isosurfaces containing millions of polygons, is quite fast. It's also topology tolerant, accepting models with nonmanifold vertices but not attempting to simplify around those vertices.

Triangle mesh decimation

Schroeder, Zarge, and Lorenson published one of the first algorithms to simplify general polygonal models and coined the term *decimation* for iterative removal of vertices.[2] Schroeder's decimation scheme is designed to operate on the output of the marching cubes algorithm for extracting isosurfaces from volumetric data and is still commonly used for this purpose. Marching cubes output is often heavily overtessellated, with coplanar regions divided into many more polygons than necessary, and Schroeder's algorithm excels at removing this redundant geometry.

The algorithm operates by making multiple passes over all the vertices in the model. During a pass, the algorithm considers deleting each vertex. If the vertex can be removed without violating the neighborhood's local topology, and if the resulting surface would lie within a user-specified distance of the unsimplified geometry, the algorithm deletes the vertex and all its associated triangles. This leaves a hole in the mesh, which is then retriangulated. The algorithm continues to iterate over the vertices in the model until it can't remove any more vertices.

The vertices of a model simplified by the decimation algorithm are a subset of the original model's vertices. This property is convenient for reusing normals and texture coordinates at the vertices, but it can limit the fidelity of the simplifications since minimizing the geometric error introduced by the simplified approximation to the original surface can require changing the vertices' positions.[5] The decimation algorithm, designed to reduce isosurfaces containing millions of polygons, is quite fast. It's also topology tolerant, accepting models with nonmanifold vertices but not attempting to simplify around those vertices. Schroeder, Zarge, and Lorenson have since developed a topology-modifying algorithm.[14] Public-domain decimation code is available as part of the Visualization Tool Kit at http://www.kitware.com/vtk.html.

Vertex clustering

This vertex-merging algorithm, first proposed by Rossignac and Borrel,[15] is topology insensitive, neither requiring nor preserving valid topology. The algorithm can therefore deal robustly with degenerate models with which other approaches have little or no success. The Rossignac–Borrel algorithm begins by assigning an importance to each vertex. Vertices attached to large faces and vertices of high curvature are considered more important than vertices attached to small faces and vertices of low curvature. Next, the algorithm overlays a 3D grid on the model and collapses all vertices within each cell of the grid to the single most important vertex within the cell. The grid's resolution determines the quality of the resulting simplification; a coarse grid will aggressively simplify the model whereas a fine grid will perform only minimal reduction. During the clustering process, triangles whose corners are collapsed together become degenerate and disappear.

Low and Tan[16] introduced a different clustering approach called *floating-cell clustering*. This technique ranks the vertices by importance, and a cell of user-specified size is centered on the most important vertex. The algorithm collapses all vertices falling within the cell to the representative vertex and filters out degenerate triangles as in the Rossignac–Borrel scheme. The most important remaining vertex becomes the center of the next cell, and the process repeats. Eliminating the underlying grid greatly reduces the simplification's sensitivity to the model's position and orientation. Low and Tan also improved on the criteria used for calculating vertex importance. Recently, Lindstrom extended the Rossignac–Borrel algorithm in a different direction, describing an out-of-core implementation that requires only enough memory to store the final (simplified) mesh.[8] Lindstrom also used Garland and Heckbert's quadric error metric (described later) to place the representative vertex for each cell.

One unique feature of the Rossignac–Borrel algorithm is the fashion in which it renders triangles with merged corners. Reasoning that a triangle with two corners collapsed is a line and a triangle with three corners collapsed is a point, Rossignac and Borrel chose to render such triangles using the graphics hardware line and point primitives. Thus, a simplification of a polygonal object will generally be a collection of polygons, lines, and points. The resulting simplifications are therefore more accurate from a schematic than a strictly geometric standpoint. For the purposes of drastic simplification, however, the lines and points can contribute significantly to the recognizability of the object. Low and Tan extended this concept, using thick lines and a dynamically assigned normal to give a cylinder-like appearance to degenerate triangles collapsed to lines.

The original Rossignac–Borrel algorithm and Lindstrom's out-of-core extension, which cluster vertices to a 3D grid, are extremely fast and run in $O(n)$ time for n vertices. The Low–Tan variation is also quite fast, though ranking the vertices by importance slows the algorithm to $O(n \lg n)$. The methods do suffer some disadvantages. Since the algorithms don't preserve topology and don't guarantee the amount of error introduced by the simplified surface, the resulting simplifications are often less pleasing visually than those of slower algorithms. Also, it's difficult to specify the output of these algorithms, since the only way to predict how many triangles an LOD will have using a specified grid resolution is to perform the simplification.

Multiresolution analysis of arbitrary meshes

This adaptive subdivision algorithm by Eck et al.[17] uses a compact wavelet representation to guide a recursive subdivision process. Multiresolution analysis, or MRA, adds or subtracts wavelet coefficients to interpolate smoothly between LODs. This process is fast enough to do at runtime, enabling dynamic simplification. By using enough wavelet coefficients, the algorithm can guarantee that the simplified surface lies within a user-specified distance of the original model.

This work's chief contribution is that it provides a method for finding a simple base mesh that exhibits subdivision connectivity, so that recursive subdivision will recover the original mesh. As previously mentioned, finding a base mesh is simple for terrain data sets but difficult for general polygonal models of arbitrary topology. MRA creates the base mesh by growing Voronoi-like regions across the original surface's triangles. When these regions can't grow anymore, the Voronoi sites form a Delauney-like triangulation, and the triangulation forms the base mesh.

This algorithm possesses the disadvantages of strict topology-preserving approaches. Manifold topology is absolutely required in the input model, and the original object's shape and genus limit the potential for drastic simplification. The fidelity of the resulting simplifications is quite high for smooth, organic forms. However, the algorithm is fundamentally a low-pass filtering approach and has difficulty capturing sharp features in the original model unless the features happen to fall along a division in the base mesh.[6]

Voxel-based object simplification

Topology-preserving algorithms must retain the original object's genus, which often limits their ability to perform drastic simplification. Topology-insensitive approaches such as the Rossignac–Borrel algorithm don't suffer from these constraints but reduce the topology of their models haphazardly and unpredictably. Voxel-based object simplification by He et al.[11] is an intriguing attempt to simplify topology in a gradual and controlled manner using a signal processing approach.

The algorithm begins by sampling a volumetric representation of the model, superimposing a 3D grid of voxels over the polygonal geometry. It assigns each voxel a value of 1 or 0, according to whether the sample point of that voxel lies inside or outside the object. Next, the algorithm applies a low-pass filter and resamples the volume. The result is another volumetric representation of the object with a lower resolution. Sampling theory guarantees that small, high-frequency features will be eliminated in the low-pass-filtered volume. The algorithm then applies marching cubes to generate a simplified polygonal model. Because marching cubes can create redundant geometry, He et al. used a standard topology-preserving algorithm as a postprocess.

Unfortunately, high-frequency details such as sharp edges and squared-off corners seem to contribute greatly to the perception of shape. As a result, the voxel-based simplification algorithm performs poorly on models with such features. This greatly restricts its usefulness on mechanical CAD models. Moreover, the algorithm as originally presented isn't topology tolerant, since deciding whether sample points lie inside or outside the object requires well-defined, closed-mesh objects with manifold topology.

Quadric error metrics provide a fast, simple way to guide the simplification process with relatively minor storage costs. The resulting algorithm is extremely fast. The visual fidelity of the resulting simplifications tends to be quite high.

Simplification envelopes

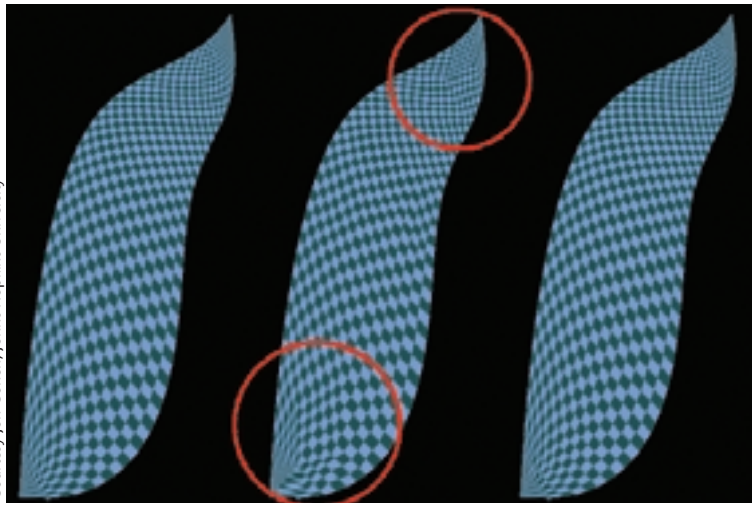
Cohen et al.[9] introduced *simplification envelopes* to guarantee fidelity bounds while enforcing global and local topology. The simplification envelopes of a surface consist of two *offset surfaces*, or copies of the surface offset no more than some distance ε from the original surface. The outer envelope displaces each vertex of the original mesh along its normal by ε, while the inner envelope displaces each vertex by $-\varepsilon$. The envelopes aren't allowed to self-intersect; where the curvature would create such a self-intersection, the algorithm decreases ε in the local neighborhood.

Once created, these envelopes can guide the simplification process. Cohen et al. describe two decimation approaches that iteratively remove triangles or vertices and retriangulate the resulting holes. By keeping the simplified surface within the envelopes, these algorithms can guarantee that the simplified surface never deviates by more than ε from the original surface and that the surface doesn't self-intersect. The resulting simplifications tend to have excellent fidelity.

Where fidelity and topology preservation are crucial, simplification envelopes prove an excellent choice. The ε error bound is also an attractive feature of this approach, providing a natural means for calculating LOD switching distances. However, the strict preservation of topology and the careful avoidance of self-intersections curtail the approach's capability for drastic simplification. The construction of offset surfaces also requires an orientable manifold, and topological imperfections in the initial mesh can hamper or prevent simplification. Finally, the algorithms for simplification envelopes are intricate—writing a robust system based on simplification envelopes seems a substantial undertaking. Fortunately, Cohen et al. posted their implementation at http://www.cs.unc.edu/~geom/envelope.html.

Appearance-preserving simplification

This rigorous algorithm by Cohen, Olano, and Manocha[3] takes the error-bounding approach of simplification envelopes a step further, providing the best



Courtesy Jon Cohen, Johns Hopkins University

8 Texture coordinate deviation. The original model (left, 1740 polygons) has a checkerboard texture applied. Simplifying to a single pixel tolerance without taking texture deviation into effect (middle, 108 polygons) results in an accurate silhouette but noticeable texture distortion. Applying the texture deviation metric (right, 434 polygons) guarantees texture as well as silhouette correctness.

guarantees on fidelity of any simplification algorithm. Fidelity is expressed in terms of maximum screenspace deviation, meaning that the simplification's appearance when rendered should deviate from the original's appearance by no more than a user-specified number of pixels. The authors identified three attributes that affect the simplification's appearance:

- *Surface position*: the coordinates of the vertices.
- *Surface color*: the color field across the mesh.
- *Surface curvature*: the field of normal vectors across the mesh.

Algorithms that guarantee a limit on the deviation of surface position (such as simplification envelopes) may nonetheless introduce errors in surface color and curvature that exceed that limit. For example, simplifying a texture-mapped surface can introduce more distortion in the texture map than in the actual geometry (see Figure 8). Appearance-preserving simplification decouples surface position from color and curvature by storing the latter in texture and normal maps (a normal map resembles a bump map), respectively. The model then reduces to a simple polygonal mesh with texture coordinates, from which the simplification algorithm computes LODs. The simplification process thus filters surface position, while the graphics hardware filters color and curvature information at runtime by MIP mapping the texture and normal maps.

The simplification process uses edge collapses, guided by a texture deviation metric that bounds the deviation of a mapped attribute value from its correct position on the original surface. The algorithm applies this deviation metric to both the texture and normal maps; edge collapses that cause surface color, normals, or position to shift by more than the maximum user-specified distance ε aren't allowed. Of course, this requirement constrains the degree of simplification possible, making appearance-preserving simplification less suitable for drastic simplification.

Although texture-mapping graphics hardware is commonplace, hardware support for normal- or bump-mapping is just beginning to appear in consumer-level systems. Appearance-preserving simplification is thus most useful today on models that don't require dynamic lighting, such as radiositized datasets. In the near future, as more sophisticated shading becomes ubiquitous in hardware, appearance-preserving simplification could become the standard for high-fidelity simplification.

Quadric error metrics

This vertex-merging algorithm by Garland and Heckbert[5] strikes perhaps the best balance yet between speed, fidelity, and robustness. The algorithm proceeds by iteratively merging pairs of vertices, which need not be connected by an edge. Candidate vertex pairs include all vertex pairs connected by an edge, plus all vertex pairs separated by less than a user-specified distance threshold t. The algorithm's major contribution is a new way to represent the error introduced by a sequence of vertex merge operations, called the *quadric error metric*. A vertex's quadric error metric is a 4×4 matrix that represents the sum of the squared distances from the vertex to the planes of neighboring triangles. Because the matrix is symmetric, 10 floating-point numbers suffice to represent the geometric deviation introduced during the course of the simplification.

The error introduced by a vertex-merge operation can be quickly derived from the sum of the quadric error metrics of the vertices being merged and that sum will become the merged vertex's quadric error metric. At the beginning of the algorithm, all candidate vertex pairs are sorted into a priority queue according to the error calculated for merging them. The algorithm removes the vertex pair with the lowest merge error from the top of the queue and merges it. The algorithm then updates the errors of all vertex pairs involving the merged vertices and repeats the process.

Quadric error metrics provide a fast, simple way to guide the simplification process with relatively minor storage costs. The resulting algorithm is extremely fast. The visual fidelity of the resulting simplifications tends to be quite high, even at drastic levels of simplification. Because disconnected vertices closer than t may merge, the algorithm doesn't require manifold topology. This lets holes close and objects merge, enabling more dras-

tic simplification than topology-preserving schemes.

One disadvantage of the algorithm is that the number of candidate vertex pairs, and hence the algorithm's running time, approaches $O(n^2)$ as t approaches the model's size. Erikson and Manocha[18] proposed an adaptive threshold selection scheme that addresses this problem. They also improved fidelity for certain models by incorporating a surface-area metric, and addressed the lack of support for shading attributes (color, normal, and texture coordinates) in the original Garland–Heckbert algorithm. Garland and Heckbert also described an extension for color in a later paper,[19] but Hoppe probably presented the best extension of quadrics to handle attributes.[20] All told, the simple-to-implement quadric-error-metrics algorithm may be the best combination of efficiency, fidelity, and generality currently available for creating LODs. Even better, Garland and Heckbert released their implementation as a software package called QSlim, available at http://graphics.cs.uiuc.edu/~garland/software/qslim.html.

Image-driven simplification

This unique algorithm by Lindstrom and Turk[4] was the first to address simplification directly in terms of how the simplified model will look when rendered. Like the quadric-error-metrics algorithm, simplification occurs through a sequence of edge collapse operations. The unique feature of image-driven simplification is the error metric used to order the edge collapses. While all other simplification algorithms to date use some form of geometric criteria—possibly modified to account for color, normal, and texture coordinate distortion—Lindstrom and Turk used a purely image-based metric.

Put briefly, we can determine the cost of an edge collapse operation by performing the collapse and rendering the model from multiple viewpoints. The algorithm compares the rendered images to images of the original model and sums up the mean-square error in luminance across all pixels of all images. It sorts all edges under consideration into a priority queue according to the total error they induce in the images and chooses the edge collapse that induces the least error. The algorithm then reevaluates and resorts nearby edges into the queue and continues the process.

Evaluating an edge collapse is clearly an expensive step. Rendering the entire model for every edge and from many viewpoints—Lindstrom and Turk used 20 viewpoints in their implementation—would be almost prohibitively expensive, even with hardware-accelerated rendering. To reduce the cost, Lindstrom and Turk exploited the fact that a typical edge collapse affects only a small region of the screen and thus a small fraction of the total triangles. They used a clever scheme for "unrendering" and rerendering the affected triangles, based on spatial hash tables that record the position of each triangle within each image. In this way, only a small portion of the triangles need be rendered per image to evaluate each edge collapse.

Using an image-based metric addresses several thorny problems in polygonal simplification. Most geometry-based algorithms that account for surface attributes such as color and texture coordinates use an arbitrary user-specified weighting to determine the relative importance of preserving these attributes versus preserving geometric fidelity. Evaluating each simplification operation according to its effect on a rendered image provides a direct, natural way to balance geometric and shading properties. Other advantages of the image-based approach include high-fidelity preservation of silhouette regions coupled with drastic simplification of unseen model geometry, and simplification sensitivity to artifacts caused by shading interpolation as well as to the content of texture maps across the surface.

The primary disadvantage of the image-driven approach for many developers will undoubtedly be the algorithm's speed. Despite the optimizations mentioned, reducing a model comprising tens of thousands of polygons to a few hundred polygons could take hours. To address this, Lindstrom and Turk performed two passes, presimplifying the model with a fast geometry-driven algorithm before applying image-driven simplification. Still, the image-driven stage ranges from several minutes to a few hours, much slower than the fastest geometry-driven algorithms. Depending on the application, however, the high visual quality of the resulting simplifications will undoubtedly be worth the wait to some developers.

Progressive meshes

A *progressive mesh* represents polygonal models as a sequence of edge collapses. Hoppe introduced progressive meshes as the first dynamic simplification algorithm for general polygonal manifolds[6] and later extended them to support view-dependent simplification.[21] A progressive mesh consists of a simple base mesh, created by a sequence of edge collapse operations and a series of vertex split operations. A vertex split (vsplit) is the dual of an edge collapse (ecol). Each vsplit replaces a vertex by two edge-connected vertices, creating one additional vertex and two additional triangles. The vsplit operations in a progressive mesh correspond to the edge-collapse operations used to create the base mesh. Applying every vsplit to the base mesh will recapture the original model exactly; applying a subset of the vsplits will create an intermediate simplification. In fact, the stream of vsplit records encodes a continuum of simplifications from the base mesh up to the original model. The vsplit and ecol operations are fast enough to apply at runtime, supporting dynamic and view-dependent simplification.

Along with the new representation, Hoppe described a careful simplification algorithm that explicitly models mesh complexity and fidelity as an energy function to be minimized. The algorithm evaluates all edges that can be collapsed according to their effect on this energy function and sorts them into a priority queue. The energy function can then be minimized in a greedy fashion by performing the ecol operation at the head of the queue, which will decrease the energy function. The algorithm then reevaluates and resorts nearby edges into the queue. This process repeats until topological constraints prevent further simplification. The remaining edges and triangles comprise the base mesh, and the sequence of ecol operations performed becomes (in

reverse order) the hierarchy of vsplit operations.

Hoppe introduced a nice framework for handling surface attributes of a progressive mesh during simplification. He categorized such attributes as discrete attributes—associated with faces in the mesh and scalar attributes—associated with corners of the faces in the mesh. Common discrete attributes include material and texture identifiers; common scalar attributes include color, normal, and texture coordinates. Hoppe also described how to model some of these attributes in the energy function, letting normals, color, and material identifiers guide the simplification process.

Hoppe described three view-dependent simplification criteria. A view frustum test aggressively simplifies regions of the mesh out of view, a backfacing test aggressively simplifies regions of the mesh not facing the viewer, and a screenspace error threshold guarantees that the geometric error in the simplified mesh is never greater than a user-specified screenspace tolerance. Because the algorithm measures deviation tangent to the surface separately from deviation perpendicular to the surface, silhouette preservation falls out of this error test naturally. Clever streamlining of the math involved makes these tests surprisingly efficient. Hoppe reported that evaluating all three criteria, which share several subexpressions, takes only 230 CPU cycles on average.

The assumption of manifold topology is latent in the progressive mesh structure, which may be a disadvantage for some applications. Preserving topology prevents holes from closing and objects from aggregating, which can limit drastic simplification, and representing nonmanifold models as a progressive mesh might present difficulties. Still, the progressive mesh representation provides a powerful and elegant framework for polygonal simplification. Hoppe's energy-minimization approach produces high-fidelity simplifications but is relatively slow and seems somewhat intricate to code. Note, however, that any algorithm based on edge collapses can be used to generate a progressive mesh. For example, the quadric-error-metrics approach would be a fast and simple-to-code alternative. Although the progressive mesh code isn't publicly available, Hoppe published a paper describing its efficient implementation in some detail.[22]

Hierarchical dynamic simplification

This vertex-merging approach by Luebke and Erikson[7] was another of the first to provide a view-dependent simplification of arbitrary polygonal scenes. Hierarchical dynamic simplification (HDS) resembles progressive meshes in many ways, with a hierarchy of vertex merges applied selectively at runtime to effect view-dependent simplification. The approaches differ mostly in emphasis: progressive meshes emphasize fidelity and consistency of the mesh, whereas HDS emphasizes speed and robustness. Rather than representing the scene as a collection of objects, each at several LODs, in the HDS algorithm the entire model comprises a single, large data structure. This structure is the vertex tree, a hierarchy of vertex clusters that HDS queries to generate a simplified scene.

The system is dynamic. For example, clusters to be collapsed or expanded can be chosen continuously based on their projected size. As the viewpoint shifts, the screenspace extent of some nodes in the vertex tree will become small. These nodes can be *folded* into their parent nodes, merging vertices together and removing some now-degenerate triangles. Other nodes will increase in apparent size and will be *unfolded* into their constituent child nodes, introducing new vertices and triangles.

Different folding criteria can be plugged into the HDS framework as callbacks that fold and unfold the appropriate nodes. Demonstrated criteria include a screenspace error threshold, a silhouette test, and a triangle budget. The screenspace error threshold monitors the projected extent of vertex clusters and folds nodes smaller than some number of pixels on the screen. The silhouette test uses a precalculated cone of normals to determine whether a vertex cluster currently lies on the silhouette, then tests clusters on the silhouette against a tighter screenspace threshold than clusters in the interior. Finally, HDS implements triangle-budget simplification by maintaining a priority queue of vertex clusters. The cluster with the largest screenspace error is unfolded and its children placed in the queue. This process repeats until unfolding a cluster would violate the triangle budget.

Because constructing the vertex tree disregards triangle connectivity, HDS neither requires nor preserves manifold topology. Since the vertex tree spans the entire scene, objects may be merged as simplification proceeds. Together, these properties make HDS topology tolerant and suitable for drastic simplification. HDS' structures and methods are also simple to code. On the other hand, the fidelity of the resulting simplifications tends to be lower than the fidelity of more careful algorithms. Again, it's important to emphasize that any algorithm based on vertex merging (including those using edge collapses) can be used to construct the HDS vertex tree.

This algorithm has been implemented as a public-domain library called VDSlib that provides a framework for view-dependent simplification, using user-defined callbacks to control the construction, simplification, culling, and rendering of the vertex tree. VDSlib is available at http://vdslib.virginia.edu.

Issues, trends, and recommendations

The field of polygonal simplification seems to be approaching maturity. For example, researchers are converging on vertex merging as the underlying mechanism for polygon reduction. Hierarchical vertex-merging representations such as progressive meshes and the HDS vertex tree provide general frameworks for experimenting with different simplification strategies, including view-dependent criteria. Settling on this emerging standard will let the simplification field make faster strides in other important issues, such as determining a common error metric.

The lack of an agreed-upon definition of fidelity seriously hampers comparison of results among algorithms. Most simplification schemes use some sort of distance- or volume-based metric in which fidelity of the simplified surface is assumed to vary with the distance of that surface from the original mesh. Cignoni et al.[23]

described a nice tool called Metro, available at http://vcg.iei.pi.cnr.it/metro.html, for measuring and visualizing this sort of geometric fidelity.

Probably the most common use of polygonal simplification, however, is to speed up rendering for visualization of complex databases. For this purpose, the most important measure of fidelity isn't geometric but perceptual: Does the simplification look like the original? To date, only Cohen's appearance-preserving simplification[3] and Lindstrom's image-driven simplification[4] attempt to address this question. Perceptual metrics and perceptually driven simplification seem like crucial topics for further research.

Dozens of simplification algorithms have been published over the last few years and dozens more have undoubtedly been whipped up by developers who were unaware of, or confused by, the plethora of polygonal simplification techniques. Hopefully this article will help developers consider the right issues and pick an algorithm that best suits their needs. ■

References

1. J. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *Comm. ACM*, vol. 19, no. 10, 1976, pp. 547-554.
2. W. Schroeder, J. Zarge, and W. Lorenson, "Decimation of Triangle Meshes," *Computer Graphics* (Proc. Siggraph 92), vol. 26, ACM Press, New York, 1992, pp. 65-70.
3. J. Cohen, M. Olano, and D. Manocha, "Appearance Preserving Simplification," *Computer Graphics* (Proc. Siggraph 98), vol. 32, ACM Press, New York, 1998, pp. 115-122.
4. P. Lindstrom and G. Turk, "Image-Driven Simplification," *ACM Trans. Graphics*. vol. 19, no. 3, July 2000, pp. 204-241.
5. M. Garland and P. Heckbert, "Simplification Using Quadric Error Metrics," *Computer Graphics* (Proc. Siggraph 97), vol. 31, ACM Press, New York, 1997, pp. 209-216.
6. H. Hoppe, "Progressive Meshes," *Computer Graphics* (Proc. Siggraph 96), vol. 30, ACM Press, New York, 1996, pp. 99-108.
7. D. Luebke and C. Erikson, "View-Dependent Simplification of Arbitrary Polygonal Environments," *Computer Graphics* (Proc. Siggraph 97), vol. 31, ACM Press, New York, 1997, pp. 199-208.
8. P. Lindstrom, "Out-of-Core Simplification of Large Polygonal Models," *Computer Graphics* (Proc. Siggraph 2000), vol. 34, ACM Press, New York, 2000, pp. 259-262.
9. J. Cohen et al., "Simplification Envelopes," *Computer Graphics* (Proc. Siggraph 96), vol. 30, ACM Press, New York, 1996, pp. 119-128.
10. W. Lorenson and H. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics* (Proc. Siggraph 87), vol. 21, ACM Press, New York, 1987, pp. 163-169.
11. T. He et al., "Voxel-Based Object Simplification," *Proc. Visualization 95*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 296-303.
12. J. El-Sana and A. Varshney, "Controlled Simplification of Genus for Polygonal Models," *Proc. IEEE Visualization 97*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 403-412.
13. P. Lindstrom et al., "Real-Time, Continuous Level of Detail Rendering of Height Fields," *Computer Graphics* (Proc. Siggraph 96), vol. 30, ACM Press, New York, 1996, pp. 109-118.
14. W. Schroeder, "A Topology-Modifying Progressive Decimation Algorithm," *Proc. IEEE Visualization 97*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 205-212.
15. J. Rossignac and P. Borrel, "Multi-Resolution 3D Approximations for Rendering Complex Scenes," *Geometric Modeling in Computer Graphics*, Springer-Verlag, Berlin, 1993, pp. 455-465.
16. K-L. Low and T.S. Tan, "Model Simplification Using Vertex Clustering," *Proc. 1997 Symp. Interactive 3D Graphics*, ACM Press, New York, 1997, pp. 75-82.
17. M. Eck et al., "Multiresolution Analysis of Arbitrary Meshes," *Computer Graphics* (Proc. Siggraph 95), vol. 29, ACM Press, New York, 1995, pp. 173-182.
18. C. Erikson and D. Manocha, "GAPS: General and Automatic Polygonal Simplification," *Proc. 1999 Symp. Interactive 3D Graphics*, ACM Press, New York, 1999, pp. 79-88.
19. M. Garland and P. Heckbert, "Simplifying Surfaces with Color and Texture using Quadric Error Metrics," *Proc. IEEE Visualization 98*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 263-270.
20. H. Hoppe, "New Quadric Metric for Simplifying Meshes with Appearance Attributes," *Proc. IEEE Visualization 99*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 59-66.
21. H. Hoppe, "View-Dependent Refinement of Progressive Meshes," *Computer Graphics* (Proc. Siggraph 97), vol. 31, ACM Press, New York, 1997, pp. 189-198.
22. H. Hoppe, "Efficient Implementation of Progressive Meshes," *Computers & Graphics*, vol. 22, no. 1, 1998, pp. 27-36.
23. P. Cignoni, C. Rocchini, and R. Scopigno, "Metro: Measuring Error on Simplified Surfaces," *Computer Graphics Forum*, vol. 17, no. 2, 1998, pp. 167-174.

David Luebke *is an assistant professor at the University of Virginia. His research focuses on interactive computer graphics, especially the problem of rendering very complex scenes in real time. His current work includes perceptually guided interactive rendering, automatic acquisition of 3D scenes, occlusion culling, and frameworks for high-speed view-dependent rendering. He received a BA in chemistry from Colorado College in 1993 and a PhD in computer science from the University of North Carolina in 1998.*

Readers may contact Luebke at the Dept. of Computer Science, University of Virginia, Charlottesville, VA 22903-2442, email luebke@cs.virginia.edu, http://www.cs.virginia.edu/~luebke.

©1999 IEEE, Reprinted with permission from Proceedings of the *IEEE Visualization 99*

Skip Strips: Maintaining Triangle Strips for View-dependent Rendering

Jihad El-Sana Elvir Azanli Amitabh Varshney

State University of New York at Stony Brook

Abstract

View-dependent simplification has emerged as a powerful tool for graphics acceleration in visualization of complex environments. However, view-dependent simplification techniques have not been able to take full advantage of the underlying graphics hardware. Specifically, triangle strips are a widely used hardware-supported mechanism to compactly represent and efficiently render static triangle meshes. However, in a view-dependent framework the triangle mesh connectivity changes at every frame making it difficult to use triangle strips. In this paper we present a novel data-structure, Skip Strip, that efficiently maintains triangle strips during such view-dependent changes. A Skip Strip stores the vertex hierarchy nodes in a skip-list-like manner with path compression. We anticipate that Skip Strips will provide a road-map to combine rendering acceleration techniques for static datasets, typical of retained-mode graphics applications, with those for dynamic datasets found in immediate-mode applications.

1 Introduction

Recent advances in three-dimensional acquisition, simulation, and design technologies have led to generation of datasets that are beyond the interactive rendering capabilities of current graphics hardware. Several software and algorithmic solutions have been recently proposed to bridge the increasing gap between hardware capabilities and the complexity of the graphics datasets. These include level-of-detail rendering with multi-resolution hierarchies, occlusion culling, and image-based rendering. Graphics rendering has also been accelerated through compact representations of polygonal meshes using data-structures such as triangle strips and triangle fans.

Recently, view-dependent simplifications have been introduced to enable fine-grained changes to multiresolution hierarchies that depend on parameters such as view location, illumination, and speed of motion. Such simplifications change the mesh structure at every frame to adapt to just the right level of detail necessary for visual realism. One drawback of such schemes is that they fail to take advantage of hardware-supported mechanisms for graphics acceleration, such as triangle strips. Luebke and Erikson [12] point out that view-dependent simplification being an immediate-mode technique has a relative disadvantage since most current graphics hardware takes advantage of retained-mode representations such as display lists that have static geometry and connectivity. To overcome this drawback Hoppe [9] has proposed a solution to compute triangle strips per frame for the view-dependent simplification specific to that frame. In this paper we introduce Skip Strips as a solution to this dichotomy of immediate-mode simplifications and retained-mode hardware-supported acceleration.

A Skip Strip stores the vertex hierarchy nodes in a skip-list-like manner with path compression. Our approach combines the advantages of the two methods – selection of varied level-of-detail at different regions of the surface from view-dependent simplification and faster rendering from triangle strip representations. In addition, Skip Strips do not need to store dependency lists traditionally used in view-dependent simplifications. Hence the test to prevent fold overs at run time is done in constant time, takes advantage of coherence between frames, and incrementally updates the displayed triangle strips. By using triangle strips, our algorithm is able to display the same number of triangles faster and uses less memory to store the active set of triangles.

2 Previous Work

In this section we give an overview of previous work done in the areas of view-dependent simplifications, triangle strip generation, and path compression data-structures.

2.1 View-Dependent Simplifications

Most of the previous work on generating multiresolution hierarchies for level-of-detail-based rendering has concentrated on computing a fixed set of view-independent levels of detail. At runtime an appropriate level of detail is selected based on viewing parameters. Such methods are overly restrictive and do not take into account finer image-space feedback such as light position, visual acuity, silhouettes, and view direction. Recent advances to address some of these issues in a view-dependent manner take advantage of the temporal coherence to adaptively refine or simplify the polygonal environment from one frame to the next. In particular, adaptive levels of detail have been used in terrains by Gross *et al* [6] and Lindstrom *et al* [10]. A number of techniques for conducting view-dependent simplifications of generalized polygonal meshes rely on the primitive operations of vertex-split and edge collapse as shown in Figure 1. The edge (pc) in the mesh on the left collapses to the vertex p and the resulting mesh is shown on the right. Conversely, the vertex p in the mesh on the right can split to the edge (pc) to generate the mesh on the left. Let the vertex p be considered the parent of the vertex c (as c is created from p through a vertex split). The primitves of vertex split and edge collapse were proposed in the context of progressive meshes [8].

View-dependent simplifications using the edge-collapse/vertex-split primitives include work by Xia *et al* [16], Hoppe [9], and Gueziec *et al* [7]. View-dependent

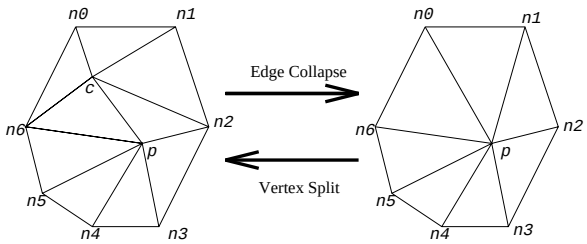


Figure 1: Edge collapse and vertex split

simplifications by Luebke and Erikson [12], and De Floriani *et al* [2] do not rely on the edge-collapse primitive. Our work is most directly applicable to view-dependent simplifications that are based upon the vertex-split/edge-collapse primitive; its extension to more general view-dependent simplifications is a part of our planned future work.

2.2 Triangle Strips

Triangle strips provide a compact representation of triangular meshes and are supported by several graphics APIs including OpenGL. Triangle strips enable fast rendering and transmission of triangular meshes. An example triangle strip in the model of a cow is shown in Figure 2. The set of triangles shown in Figure 3(a) can be compactly represented by a triangle strip $(1, 2, 3, 4, 5, 6)$, where the ith triangle is described by the ith, $(i + 1)$st, and $(i + 2)$nd vertices in this sequence. Such triangle strips are referred to as *sequential triangle strips*. A sequential triangle strip allows rendering of n triangles using only $n + 2$ vertices instead of $3n$ vertices. This results in substantial saving for memory bandwidth and computation of per-vertex operations such as transformations, lighting, and clipping. Sequential triangle strips cannot however represent general sequences of triangles, such as the one shown in Figure 3(b). To represent such triangle sequences, the notion of triangle strips has been extended to *generalized triangle strips* where the two vertices of the previous triangle can be swapped. This can be also simulated by repeating vertices. Thus, the triangle sequence in Figure 3(b) can be represented as $(1, 2, 3, 4, 5, 4, 6, 7)$.

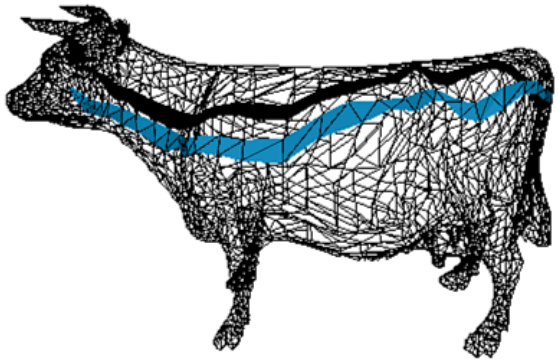


Figure 2: A triangle strip in a cow model

Akeley *et al* [1] have developed a program that constructs generalized triangle strips for a given triangle mesh model [1]. The algorithm tries to generate strips which minimize the number of one-triangle strips. This algorithm chooses the triangle

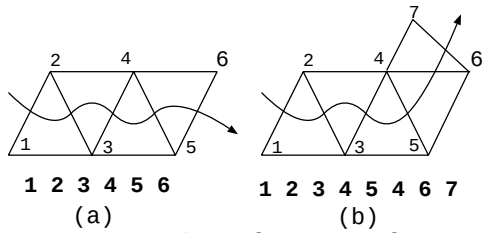


Figure 3: A triangle strip example

which is adjacent to the least number of neighbors as the next triangle in a strip. Evans *et al* [4] use global adjacency information in conjunction with several heuristics such as maximizing the length of each strip, minimizing swaps, and minimizing the number of single-triangle strips. Their experimental results have shown an improvement of about 15% less vertices than [1]. Speckmann and Snoeyink [14] have computed the triangle strips for triangulated irregular networks by creating a spanning tree of the dual graph of the TIN and then traversing the tree in a modified depth-first fashion.

2.3 Efficient Link Traversal

Let us study what happens when an edge collapses in a triangle strip. Figure 4 shows such a situation. As can be seen, the results of an edge collapse can be represented by replacing all occurrences of the child vertex c with the parent vertex p. In this example, $c = 2$ and $p = 4$.

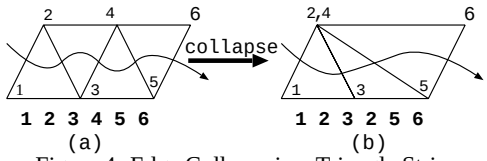


Figure 4: Edge Collapse in a Triangle Strip

The above example illustrates that to maintain triangle strips under view-dependent changes to the triangle mesh connectivity, we should replace each vertex in a triangle strip by its nearest uncollapsed ancestor. In an arbitrarily long sequence of such edge collapses, it is easy to see why efficient traversal of links to a vertex's ancestors becomes important.

Skip list [13] has been proposed as an efficient probabilistic data-structure to store and retrieve data. Skip lists can also be used for efficient compression of pointer paths. Consider a simple linked list as shown in Figure 5(a). Reaching the n-th node on this list requires O(n) pointer hops. Consider next a data-structure that resembles a binary tree and has O(n) additional pointers that connect linked-list nodes that are 2 away, 4 away, ..., $2^{\log n}$ away (refer Figure 5(b)). Using these additional pointers, any node on the linked list can be accessed in O($\log n$) hops. Skip lists generate such additional pointers in a probabilistic manner to provide the same O($\log n$) access time (refer Figure 5(c)), but in practice have been shown to be faster.

In a skip list, a node that has k forward pointers is a level k node. The level of a node is determined in a probabilistic manner. The search for an element is done by traversing forward pointers that do not overshoot the required element. When no more progress is possible, the search moves down to the next level. This is shown by the gray path in Figure 5(c). To accomplish insertion or deletion of an element in a skip list, a search is carried out for that element using the above method.

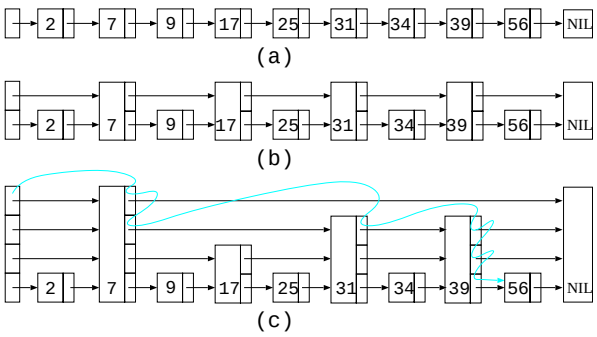


Figure 5: Skip list example

A vector of pointers is set up during this search that represents the set of pointers that are changed to implement the insert or delete operation.

3 Technical Background

In this paper we build upon two previous algorithms – construction of vertex hierarchy for view-dependent simplifications [16, 9] and construction of efficient triangle strips [4]. Let us overview these two algorithms next.

3.1 Construction of Merge Trees

Merge trees have been introduced by Xia *et al* [16] as a data-structure built upon progressive meshes [8] to enable real-time view-dependent rendering of an object. As discussed earlier, let the vertex p in Figure 1 be considered the parent of the vertex c. The *neighborhood* of a vertex v is defined as the set of triangles that are adjacent to v. The neighborhood of an edge (v_a, v_b) is defined as the union of neighborhoods of v_a and v_b. The merge tree is constructed in a bottom-up fashion from a high-detail mesh to a low-detail mesh by storing these parent-child relationships (representing edge collapses) in a hierarchical manner over the surface of an object. At each level l of the tree a maximal set of edge-collapses is selected in the shortest-edge-first order and with the constraint that their neighborhoods do not overlap. The vertices remaining after these edge collapses are promoted to level $l + 1$.

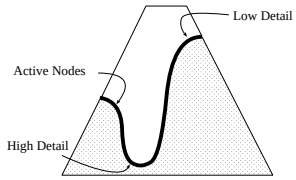


Figure 6: Varying detail in a Merge Tree

View-dependent simplification is achieved by performing edge-collapses and vertex-splits on the triangulation used for display depending upon view-dependent parameters such as lighting (detail is directly proportional to intensity gradient), polygon orientation, (high detail for silhouettes and low detail for backfacing regions) and screen-space projection. This is shown in Figure 6. Since there is a high temporal coherence the selected levels in the merge tree change only gradually from frame to frame. Unconstrained edge-collapses and vertex-splits during runtime can be shown to result in mesh foldovers resulting in visual artifacts such as shading discontinuities. To avoid these artifacts Xia *et al* propose the concept of dependencies or constraints that necessitate the presence of the entire neighborhood of an edge before it is collapsed (or its parent vertex is split). Thus, for the example shown in Figure 1, the neighborhood of edge pc should consist exactly of vertices $n_0 \ldots n_6$ for c to collapse to p. Similarly, for the vertex p to split to c, the vertices adjacent to p should be exactly the set $n_0 \ldots n_6$. Our current implementation of merge trees can construct the merge tree for 69K triangles bunny model in 10.3 seconds on an SGI Onyx 2.

3.2 Generating Triangle Strips

We use the *Stripe* program by Evans *et al* [4] to generate high quality triangle strips. This approach considers the problem of constructing good triangle strips from polygonal models. Often such models are not fully triangulated, and contain quadrilaterals and other non-triangular faces, which must be triangulated prior to rendering. The choice of triangulation can significantly impact the cost of the resulting strips. Evans *et al* have experimented with several variants of local and global algorithms; the details are available in [4]. After comparing the results from 20 different local and global approaches on over 200 datasets, the best option has been empirically observed to use the global row or column strips with a patch cutoff size of 5. In this approach the model is first partitioned into regions that have collections of $m \times n$ quadrilaterals arranged in m rows and n columns, which is referred to as a *patch*. Each patch whose number of quadrilaterals, mn, is greater than a specified cutoff, in this case 5, is converted into one strip, at a cost of three swaps per turn. Further, every such strip is extended backwards from the starting quadrilateral and forwards from the ending quadrilateral of the patch to the extent possible. On triangulated models like the ones we consider in this paper, *Stripe* has been found to work as well as other public-domain triangle strip converters. *Stripe* Version 2.0 [3] converts the 69K triangles bunny model into triangle strips in 6 seconds on an SGI Onyx 2.

4 Our Approach

In our approach we first generate a merge tree file as overviewed above in Section 3.1 and described in [16]. This file contains the parent-child relationships for each node of the tree. Even though our implementation uses merge trees, the concept of Skip Strips is quite general and can be used in conjunction with other vertex-collapse-based simplification schemes as well. We next generate the triangle strip representation of the original polygonal model using the *Stripe* program as overviewed above in Section 3.2 and described in [4]. At run-time we load the merge tree and the triangle strip representations generated during preprocessing and build the *Skip Strip* data-structure on the fly. Then, depending on scene parameters such as eye position, local illumination, front/back-facing regions, we perform vertex split and edge collapse operations directly on the Skip Strips. The information from Skip Strips is then used to generate triangle strips for display.

4.1 Skip Strip data-structure

A Skip Strip is an array of Skip Strip nodes. Each Skip Strip node contains vertex information, a list of child pointers and a parent pointer. We shall see in Section 4.3 how to generalize this data-structure to support a list of parent pointers to accelerate access in a edge-collapse hierarchy. Exactly one of the child pointers is marked as an *active* child pointer. This can be seen in Figure 7 where the parent pointers are shown on the right and the list of child pointers is shown on the left of each Skip Strip node. Also, the parent pointer of the node is marked *active* if this node has collapsed to its parent at a given stage of simplification; otherwise it is marked *inactive*.

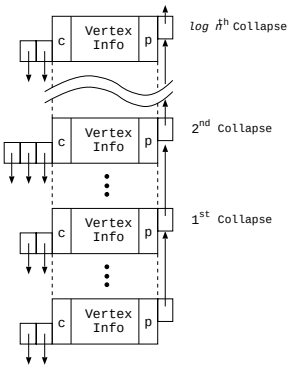


Figure 7: A Skip Strip node

A Skip Strip is constructed at run time from the merge tree and the triangle strip representations. A Skip Strip node is allocated for every merge tree node and then parent-child pointers are set up to mimic the merge tree structure. In our current implementation we are assuming that a child vertex c collapses to a parent vertex p. For this case, a Skip Strip node corresponding to a vertex p will have child pointers to all its children, including c, that collapse to it at different stages of simplification. In general, if there are n vertices then the height of the merge tree is $O(\log n)$. Thus, the length of this child-pointer list for a Skip Strip node could be $O(\log n)$. At a given time only one of these child pointers is flagged *active* and represents the node that will result from the most imminent split. Each Skip Strip node points to its immediate parent via the parent pointer.

To illustrate the Skip Strip data-structure, let us see how it is built from a merge tree. Figure 8(a) shows a hypothetical merge tree over four vertices 1 to 4. As in all the merge tree diagrams in this paper, the right node is the child node and the left node is the parent node (as defined by Figure 1). The equivalent Skip Strip data-structure will have four nodes representing the leaves of the merge tree (the highest detail vertices in the original model). Since according to the merge tree vertex 2 can merge to verex 1, the parent pointer for the Skip Strip node 2 will point to Skip Strip node 1 and the child pointer for the node 1 will point to node 2. Similarly the parent and child pointers of Skip Strip nodes 3 and 4 will be set. This stage is shown in Figure 8(b). Next, let us assume that we are dealing with edge collapses in which one vertex collapses to another (i.e. no new vertices are created). Therefore in our merge tree of Figure 8(a), node 5 is a representation of node and node 6 is a representation of node 3. Since node 6 collapses into node 5, it follows that node 3 will merge to node 1. As a result, node 7 just represents node 1. This structure can be represented in the Skip Strip as a parent pointer from node 3 to node 1 and a child pointer from node 1 to node 3. The completed Skip Strip structure is shown in Figure 8(c).

Figure 8: Building a Simple Skip Strip

The method that we have outlined above assumes that in an edge collapse from c to p, the new vertex is p. However, several other researchers have pointed out the advantage of creating new vertices during edge collapses. These new vertices could be created for accomplishing geomorphs [8] or for better placement of approximating vertices using sophisticated error metrics [5, 11]. For incorporating such simplification metrics into the framework of Skip Strips we suggest storing multiple coordinate sets, once per approximating vertex, in the child pointer of the Skip Strip node.

4.2 Real-Time Adaptive Representation

Once the Skip Strip has been constructed it is easy to construct an adaptive level-of-detail mesh representation during run-time. Real-time adaptive mesh representation involves the determination of the vertices and the triangle strips at the current level of detail. We shall refer to the vertices and triangle strips selected for display at a given frame as *display vertices* and *display strips*.

4.2.1 Determination of display vertices

Determination of display vertices proceeds along the same lines as proposed in earlier work on view-dependent simplification [16, 9] where image-space feedback is used to guide the selection of the level of detail for the mesh. We determine which region of an object to simplify more and which to simplify less using several parameters such as viewer location and orientation, local illumination, and front/back-facing regions of an object. As mentioned earlier, every node of the merge tree stores a *switch value*. If the computed value of the view-dependent error at a given node v is less than the *switch value* stored at node v, then node v splits. If the computed value is larger than the *switch value* stored at the parent of node v, then v merges. In addition to these criteria, each collapse and split also depends on the validity of the operation as determined during the preprocessing to avoid artifacts such as mesh foldovers as explained earlier in Section 3.1. One way to avoid such artifacts is to use dependencies [10, 16]. Until now these dependencies needed to be stored as a set of adjacent vertices. This resulted in high storage as well as verification costs. However, using the Skip Strip data-structures we have found that these checks can be reduced to just one pointer traversal. The Skip Strip structure encodes the sequence of vertex splits and edge collapses in the same order as they were permitted during the pre-processing. Therefore to validate a potential vertex split of v to a child vertex c we verify that c is being pointed to by the currently active child pointer for v. Similarly, a potential collapse of a vertex v to a parent p just needs to verify that the active parent pointer of v is currently pointing to p. We have discovered that these simpler checks have reduced the times for checking and performing a vertex split or edge collapse from around 60μsecs to 6μsecs.

4.2.2 Determination of display strips

Our scene is represented as a set of triangle strips. Each triangle strip has two representations – the original highest resolution triangle strip that was generated using pre-processing and the Skip-Strip-derived run-time representation of it that represents a triangle strip suitable for the current level of detail. We refer to the former as a *original triangle strip* and the latter as a *display strip*. At each frame we first perform view dependent edge collapses/vertex splits as outlined in Section 4.2.1. Each time an edge collapses or vertex splits, all display strips that contain that edge are flagged as modified. At the end of these simplifications if a display strip remains unmodified it is used for rendering. However, if a display strip is modified we discard it and begin generating its replacement by scanning each vertex in the corresponding original triangle strip. Each vertex of the original triangle strip has a pointer to a corresponding node in a Skip Strip. For each vertex's node in the Skip Strip we check if its parent pointer is active or not. If the parent pointer is active we follow the sequence of active parent pointers until we reach a node that has an inactive parent pointer. The vertex information stored with the first node that has an inactive parent pointer is added to the new display strip. After the new display strip has been completely generated it is sent to the graphics system for display.

Let us next illustrate how the Skip Strips are used to split and collapse vertices of a triangle strip to generate the display strips. Figure 9 shows the original mesh with vertices numbered $1..10$. The two triangle strips representing this mesh are labeled a and b. Since no edges have collapsed, the display strips are the same as the original triangle strips. Figure 10 shows the merge tree and the skip strip with one parent pointer per node, constructed for the mesh in Figure 9 at the highest detail. Figure 11 shows the same after two edge collapses ($6 \rightarrow 5$, and $8 \rightarrow 7$) to the mesh of Figure 9. In Figure 10 none of the parent pointers is active (since there have been no edge collapses). In Figure 11, the parent pointers for nodes 6 and 8 that point to 5 and 7 respectively, are active and appear shaded.

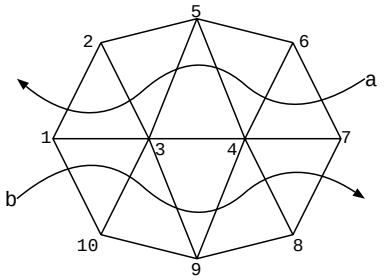


Figure 9: Original triangle mesh

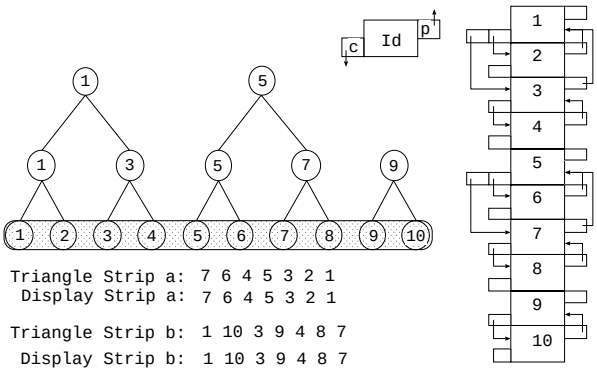


Figure 10: Skip Strip for Triangle Mesh in Figure 9

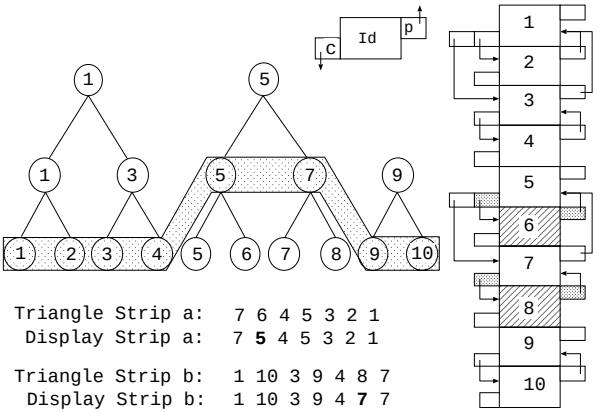


Figure 11: Skip Strip for Triangle Mesh in Figure 9 after two edge collapses

4.3 Efficient Skipping for Parent pointers

As the object moves to a coarser representation, the time spent in following the active parent pointers increases. The maximum number of active parent pointers that one might need to traverse is $O(\log n)$ – the height of the vertex hierarchy. To reduce this time we trade off memory for speed. To accomplish this we use ideas from path compression [15] and skip lists [13] to build a list of parent pointers for each Skip Strip node that point to ancestors of this node that are $1, 2, 4, 8, \ldots, \log n$ away in the edge collapse hierarchy. By using an efficient, skip-list-like pointer hopping scheme we can reduce this to $O(\log \log n)$. Although reducing a $O(\log n)$ factor might seem minor, in practice this results in an appreciable difference, especially when we note that the merge tree height is generally a logarithm to the base $5/4$ [16]. Thus, even if the edge-collapse-based vertex hierarchy tree is balanced (which often is not), the height for a tree over one million vertices (and therefore the worst-case pointer hopping) will be 62 ($\sim \log_{1.25}(10^6)$) while a skip-list-like pointer hopping scheme will only need to traverse 6 ($\sim \log_2 62$) pointers, an order of magnitude improvement for present-day datasets.

In this scheme each Skip Strip node has an active parent field to indicate which pointer in the parent list to follow to get closest to, without overshooting, the first active ancestor. We use a lazy update scheme to modify the active parent field for each Skip Strip node. For this we make use of the fact that the vertex hierarchy nodes are collapsed in an accordion-style fashion from high-detail to low-detail. In other words if a vertex i collapses to vertex j, then it means that *all* vertices that lie in the sub-tree rooted at vertex i have already collapsed to vertex i. If the triangle strips reference one of the vertices in this sub-tree rooted at i and if their active parent pointer overshoots j, then we need to decrement the active parent pointer till it points to a node that is below j (in other words has al-

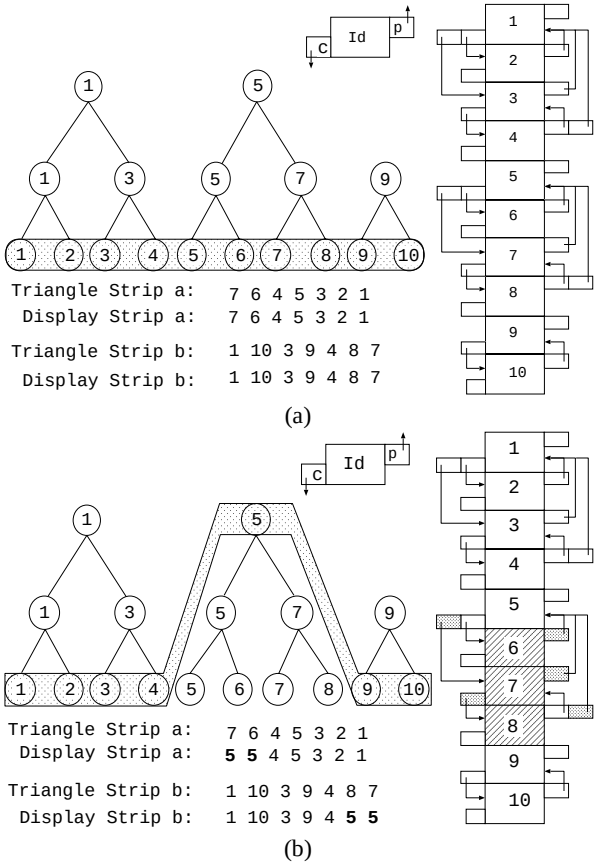


Figure 12: More efficient Skip Strip representations for Figures 10 and 11

ready collapsed). Because of a high temporal coherence, these updates are few and each requires only one or two ancestor checks to find the "correct" ancestor that does not overshoot the first active ancestor. Similarly, when a vertex j splits to vertices i and j we update all pointers from triangle strips that point to j as the first active ancestor to point to a lower level ancestor. We would like to point out that in this application, traversal of triangle strips requires that we access each vertex of the triangle strip and therefore the overhead of such lazy updates of pointers to reflect split and collapse in Skip Strips is minimal. Figure 10 shows the Skip Strip representation with multiple parent pointers, as described above, for the mesh of Figure 9. The active parent pointers appear in bold lines.

4.4 Filtering Triangle Strips

As the model moves to coarser levels the triangle strips begin to accumulate identical vertices. Sending such vertices multiple times is equivalent to sending degenerate triangles that do not contribute to the final scene but add an overhead to the graphics rendering. To address this we filter the triangle strips while sending them to the graphics engine. We have implemented a simple triangle strip scanner that detects and replaces patterns of vertices of the regular expression form $(aa)^+$ and $(ab)^+$ from the sequence of vertices sent for rendering and replaces them with (aa) or (ab) as appropriate.

5 Results

We have implemented Skip Strips and have obtained the results shown in Tables 1 and 2. All of these results have been obtained on an SGI Onyx 2 with four R10000 processors, 1 GB RAM. The video tape has been generated using two of these four processors – one processor was used for display and the second processor was used for view-dependent simplifications and skip-strip maintenance. Timings reported here do not assume parallelization.

Table 1 shows the comparison between rendering datasets using triangle representation as in the conventional view-dependent rendering and rendering using Skip Strips. As can be seen the number of vertices sent is substantially less for Skip Strips since they incrementally maintain triangle strips that are used for final rendering.

Dataset	Frame Tris	Triangles		Skip Strips	
		Display (msec)	Verts Sent	Display (msec)	Verts Sent
Bunny	10.2K	27.1	30.6K	16.0	22K
	50.8K	120.1	152.4K	78.1	83K
	69.4K	166.1	208.2K	101.0	92K
Terrain	71.4K	129.0	214.2K	103.0	160K
	255.5K	651.2	766.5K	401.2	480K
	522.2K	1573.5	1566.6K	991.3	550K
AMR	90.4K	134.1	271.2K	103.1	181K
	180.6K	385.4	541.8K	213.7	320K
	340.2K	573.1	1020.6K	297.4	491K
Dragon	107.8K	256.2	323.4K	204.1	205K
	445.5K	850.3	1336.5K	611.2	800K
	871.3K	1908.2	2613.9K	1184.5	1280K

Table 1: Comparison between Raw Triangles and Skip Strips for Rendering

Table 2 shows the advantage of using Skip Strips over recomputing triangle strips at every frame. As can be seen from this table, recomputing triangle strips results in fewer vertices being sent as compared to those using Skip Strips. However, the total cost of recomputing triangle strips along with the time for their display far exceeds the cost of maintaining, updating, and displaying triangle strips using Skip Strips. All times reported below are wall clock times (not CPU times).

The datasets used for the above results appear in Figures 12, 13, and 14. In these figures, parts (a) show an intermediate level of view-dependent simplification, while parts (b), (c), and (d) show how the triangle strips are maintained across different levels of detail using Skip Strips. Colors in parts (a) depict object colors whereas colors in parts (b), (c), and (d) denote different triangle strips.

6 Conclusions

We have shown how Skip Strips can provide a convenient and simple representation to integrate retained-mode data-structures such as triangle strips with immediate-mode view-dependent simplifications. The Skip Strips offer two main advantages. First, they make pointer hopping along parent links in any hierarchical vertex collapse scheme efficient. Second, they simplify the tests for vertex split and edge collapse to be as simple as verification of their active child and active parent pointers.

Dataset	Triangles in Frame	Strips			Skip Strips		
		Construction (msec)	Display (msec)	Vertices Sent	Triangle Strip Updates (msec)	Display (msec)	Vertices Sent
Bunny	5K	310	4.6	8K	5.3	6.4	10K
	30K	2120	23.1	35K	13.1	30.3	42K
	65K	5310	84.6	90K	25.2	85.2	90K
Terrain	32K	2190	54.3	60K	15.2	78.4	90K
	255K	24630	370.5	411K	150.3	401.2	480K
	522K	131070	450.3	550K	230.0	460.3	550K
AMR	65K	4100	72.4	80K	32.3	101.3	130K
	170K	14300	173.2	268K	60.8	201.4	300K
	340K	35810	291.5	491K	101.2	297.4	491K

Table 2: Computing Triangle Strips per frame versus Skip Strips

Skip Strips provide the advantage of hardware-assisted acceleration to view-dependent simplifications. However, they also suffer from some of the same limitations that afflict triangle strips. Thus, Skip Strip performance will not be very good for datasets that have several discontinuities in surfaces (cracks, holes, T-junctions), normals, colors, and textures. For such datasets the triangle strips that are generated have to be split across such surface attribute discontinuities thereby limiting their efficacy in succinctly representing the polygonal mesh. Although this does affect overall performance, the results will likely still be better than rendering raw triangles.

Another issue to consider is the performance of Skip Strips over genus-reducing simplifications. Our preliminary results indicate that Skip Strips are also applicable to view-dependent genus-reducing simplifications; we need to test this further.

References

[1] K. Akeley, P. Haeberli, and D. Burns. tomesh.c : C Program on SGI Developer's Toolbox CD, 1990.

[2] L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulation. In H. Rushmeier D. Elbert and H. Hagen, editors, *Proceedings Visualization '98*, pages 43–50, October 1998.

[3] F. Evans, E. Azanli, S. Skiena, and A. Varshney. Stripe Version 2.0, http://www.cs.sunysb.edu/∼stripe.

[4] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization '96 Proceedings*, pages 319 – 326. ACM/SIGGRAPH Press, October 1996.

[5] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 209 – 216. ACM SIGGRAPH, ACM Press, August 1997.

[6] M. H. Gross, R. Gatti, and O. Staadt. Fast multiresolution surface meshing. In G. M. Nielson and D. Silver, editors, *IEEE Visualization '95 Proceedings*, pages 135–142, 1995.

[7] A. Gueziec, F. Lazarus, G. Taubin, and W. Horn. Surface partitions for progressive transmission and display, and dynamic simplification of polygonal surfaces. In S. N. Spencer, editor, *Proceedings VRML 98: third Symposium on the Virtual Reality Modeling Language, Monterey, California, February 16–19, 1998*, pages 25–32, New York, NY, USA, 1998. ACM Press.

[8] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 99 – 108. ACM SIGGRAPH, ACM Press, August 1996.

[9] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 189 – 197. ACM SIGGRAPH, ACM Press, August 1997.

[10] P. Lindstrom, D. Koller, W. Ribarsky, L. Hughes, N. Faust, and G. Turner. Real-Time, continuous level of detail rendering of height fields. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 109–118. ACM SIGGRAPH, Addison Wesley, August 1996.

[11] P. Lindstrom and G. Turk. Fast and memory efficient polygonal simplification. In D. Ebert, H. Rushmeier, and H. Hagen, editors, *Proceedings Visualization '98*, pages 279–286, October 1998.

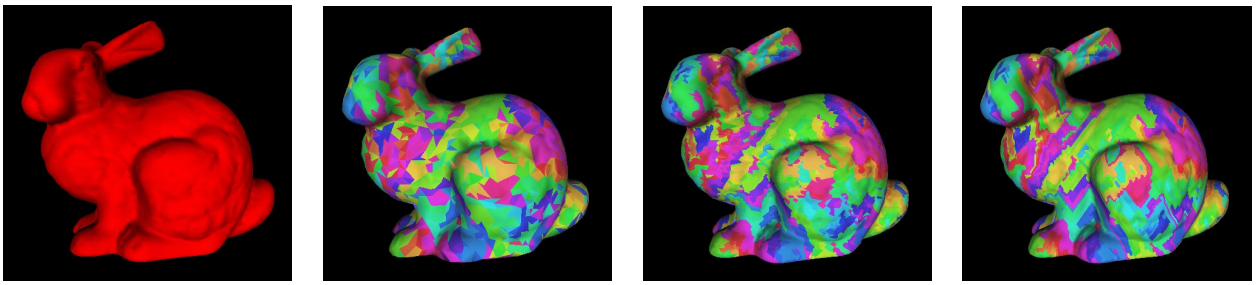
[12] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 198 – 208. ACM SIGGRAPH, ACM Press, August 1997.

[13] W. Pugh. Skip lists: A probabilitics alternative to balanced trees. *Communications of the ACM*, 33(6):668–678, 1990.

[14] B. Speckmann and J. Snoeyink. Easy triangle for TIN terrain models. In *Candaian Conference on Computational Geomerty 97*, 1997.

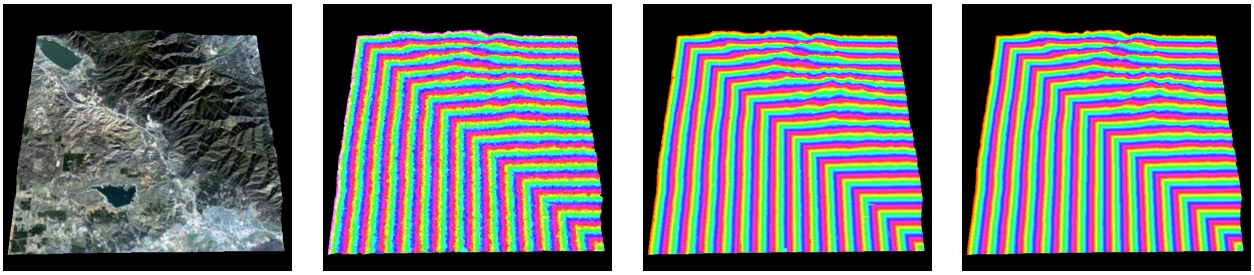
[15] R. E. Tarjan. Data structures and network algorithms. In *Regional Conference Series in Applied Mathematics*, volume 44 of *CBMS-NFS*. SIAM, 1983.

[16] 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*, pages 171 – 183, June 1997.



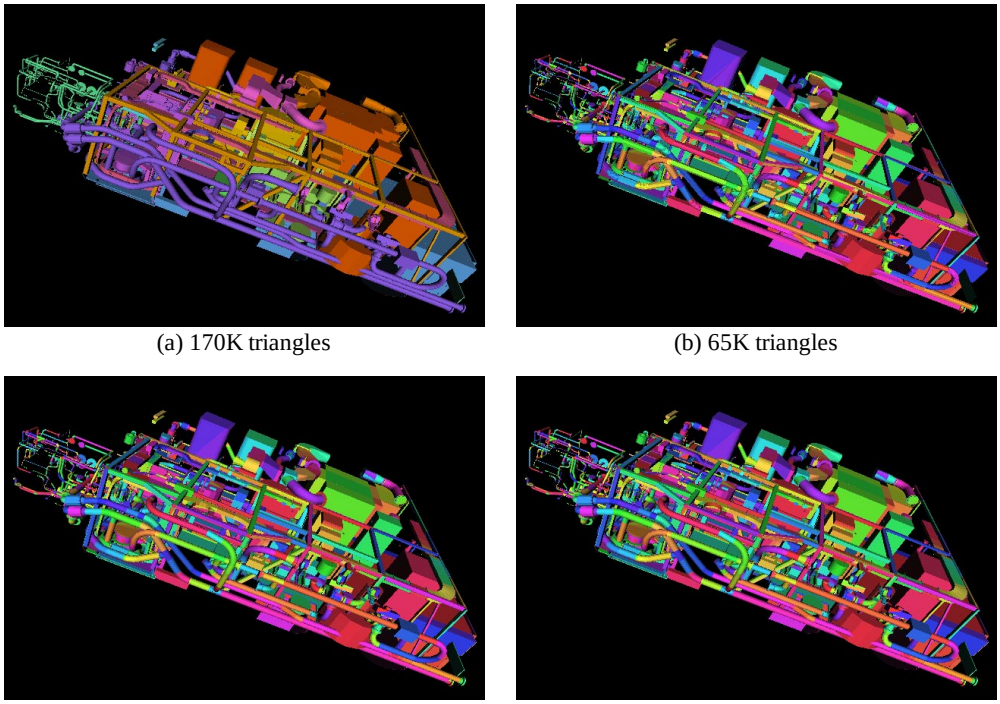
(a) 30K triangles (b) 5K triangles (c) 30K triangles (d) 65K triangles

Figure 12: Skip strips across varying resolutions for the Stanford Bunny model



(a) 255K triangles (b) 32K triangles (c) 255K triangles (d) 522K triangles

Figure 13: Skip strips across varying resolutions for the Terrain dataset



(a) 170K triangles (b) 65K triangles

(c) 170K triangles (d) 340K triangles

Figure 14: Skip strips across varying resolutions for the Auxilliary Machine Room dataset

Adaptive Real-Time Level-of-detail-based Rendering for Polygonal Models

Julie C. Xia Jihad El-Sana Amitabh Varshney

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

We present an algorithm for performing adaptive real-time level-of-detail-based rendering for triangulated polygonal models. The simplifications are dependent on viewing direction, lighting, and visibility and are performed by taking advantage of image-space, object-space, and frame-to-frame coherences. In contrast to the traditional approaches of precomputing a fixed number of level-of-detail representations for a given object our approach involves statically generating a continuous level-of-detail representation for the object. This representation is then used at run-time to guide the selection of appropriate triangles for display. The list of displayed triangles is updated incrementally from one frame to the next. Our approach is more effective than the current level-of-detail-based rendering approaches for most scientific visualization applications where there are a limited number of highly complex objects that stay relatively close to the viewer. Our approach is applicable for scalar (such as distance from the viewer) as well as vector (such as normal direction) attributes.

1 Introduction

The scientific visualization and virtual reality communities have always faced the problem that their "desirable" visualization dataset sizes are one or more orders of magnitude larger than what the hardware can display at interactive rates. Recent research on graphics acceleration for the navigation of such three-dimensional environments has been motivated by attempts to bridge the gap between the desired and the actual hardware performance, through algorithmic and software techniques. This research has involved reducing the geometric and rendering complexities of the scene by using

- statically computed level-of-detail hierarchies [35, 32, 31, 10, 25, 14, 22],
- visibility-based culling that is statically computed [1, 34] and dynamically computed [18, 27, 17],

©1997 IEEE, reprinted with permission from IEEE Transactions on Visualization and Computer Graphics, June 1997

- various levels of complexity in shading and illumination models[4],
- texture mapping [6, 5], and
- image-based rendering [8, 7, 29, 33, 13].

In this paper we will focus on reducing the geometric complexity of a three-dimensional environment by using dynamically computed level-of-detail hierarchies. Research on simplification of general three-dimensional polygonal objects (non-convex, non-terrain, possibly high genus) has spanned the entire gamut of highly local to global algorithms, with several approaches in between that have both local and global steps.

Local algorithms work by applying a set of local rules, which primarily work under some definition of a *local neighborhood*, for simplifying an object. The local rules are iteratively applied under a set of constraints and the algorithm terminates when it is no longer possible to apply the local rule without violating some constraint. The global algorithms optimize the simplification process over the whole object, and are not necessarily limited to the small neighborhood regions on the object. Some of the local approaches have been – vertex deletion by Schroeder *et al* [32], vertex collapsing by Rossignac and Borrel [31], edge collapsing by Hoppe *et al* [26] and Guéziec [20], triangle collapsing by Hamann [21], and polygon merging by Hinker and Hansen [24]. Some of the global approaches have been – redistributing vertices over the surface by Turk [35], minimizing global energy functions by Hoppe *et al* [26], using simplification envelopes by Varshney [36] and Cohen *et al* [10], and wavelets by DeRose *et al* [14]. The issue of preservation or simplification of the genus of the object is independent of whether an algorithm uses local rules, or global rules, or both, to simplify. Recent work by He *et al* [22] provides a method to perform a controlled simplification of the genus of an object.

Simplification algorithms such as those mentioned above are iteratively applied to obtain a hierarchy of successively coarser approximations to the input object. Such multiresolution hierarchies have been used in level-of-detail-based rendering schemes to achieve higher frame update rates while maintaining good visual realism. These hierarchies usually have a number of distinct levels of detail, usually 5 to 10, for a given object. At run time, the perceptual importance of a given object in the scene is used to select its appropriate level of representation from the hierarchy [9, 11, 12, 16, 30, 28]. Thus, higher detail representations are used when the object is perceptually more important and lower detail representations are used when the object is perceptually less significant. Transitions from one level of detail to the next are typically based on simple image-space metrics such as the ratio of the image-space area of the object (usually implemented by using the projected area of the bounding box of the object) to the distance of the object from the viewer.

Previous work, as outlined above, is well-suited for virtual reality walkthroughs and flythroughs of large and complex structures with several thousands of objects. Examples of such environments include architectural buildings, airplane and submarine interiors, and factory layouts. However, for scientific visualization applications where the goal often is to visualize one or two highly detailed objects at close range, most of the previous work is not directly applicable. For instance, consider a biochemist visualizing the surface of a molecule or a physician inspecting the iso-surface of a human head extracted from a volume dataset. It is very likely during such a visualization session, that the object being visualized will not move adequately far away from the viewer to allow the rendering algorithm to switch to a lower level of detail. What is desirable in such a scenario is an

algorithm that can allow several different levels of details to co-exist across different regions of the same object. Such a scheme needs to satisfy the following two important criteria:

- It should be possible to select the appropriate levels of detail across different regions of the same object in real time.
- Different levels of detail in different regions across an object should merge seamlessly with one another without introducing any cracks and other discontinuities.

In this paper we present a general scheme that can construct such seamless and adaptive level-of-detail representations on-the-fly for polygonal objects. Since these representations are view-dependent, they take advantage of view-dependent illumination, visibility, and frame-to-frame coherence to maximize visual realism and minimize the time taken to construct and draw such objects. Our approach shows how one can adaptively define such levels of detail based on (a) scalar attributes such as distance from the viewpoint and (b) vector attributes such as the direction of vertex normals. An example using our approach is shown in Figure 1.

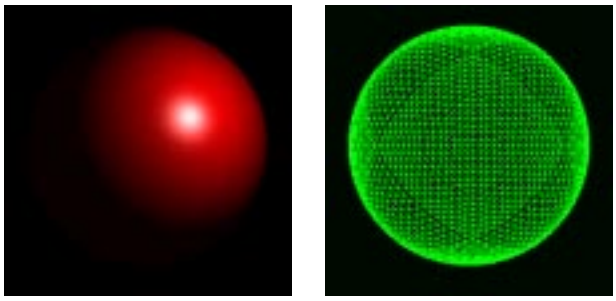
2 Previous Work

Adaptive levels of detail have been used in terrains by Gross *et al* [19] by using a wavelet decomposition of the input data samples. They define wavelet space filters that allow changes to the quality of the surface approximations in locally-defined regions. Thus, the level of detail around any region can adaptively refine in real-time. This work provides a very elegant solution for terrains and other datasets that are defined on a regular grid.

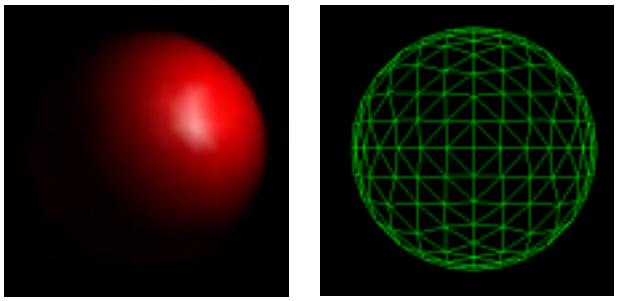
Some of the previous work in the area of general surface simplification has addressed the issue of adaptive approximation of general polygonal objects. Turk [35] and Hamann [21] have proposed curvature-guided adaptive simplification with lesser simplification in the areas of higher surface curvature. In [36, 10], adaptive surface approximation is proposed with different amounts of approximation over different regions of the object. Guéziec [20] proposes adaptive approximation by changing the tolerance volume in different regions of the object. However in all of these cases, once the level of approximation has been fixed for a given region of the object, a discrete level of detail corresponding to such an approximation is statically generated. No methods have been proposed there that allow free intermixing of different levels of detail across an object in real time in response to changing viewing directions.

Work on surface simplification using wavelets [14, 15] and progressive meshes [25] goes a step further. These methods produce a continuous level-of-detail representation for an object in contrast to a set of discrete number of levels of detail. In particular, Hoppe [25] outlines a method for selective refinement – i.e. refinement of a particular region of the object based upon view frustum, silhouette edges, and projected screen-space area of the faces. Since the work on progressive meshes by Hoppe [25] is somewhat similar to our work we overview his method next and discuss how our method extends it.

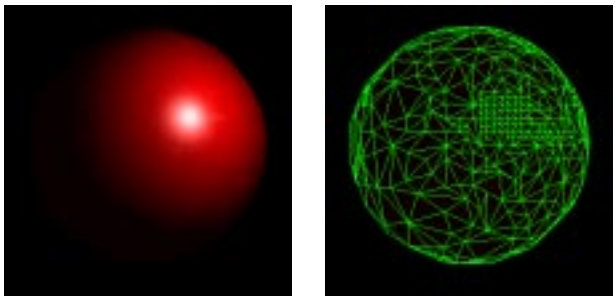
Progressive meshes offer an elegant solution for a continuous resolution representation of polygonal meshes. A polygonal mesh $\hat{M} = M^k$ is simplified into successively coarser meshes M^i by applying a sequence of edge collapses. An edge collapse transformation and its dual, the vertex split transformation, is shown in Figure 2.



(a) Sphere with 8192 triangles (uniform LOD)



(b) Sphere with 512 triangles (uniform LOD)



(c) Sphere with 537 triangles (adaptive LOD)

Figure 1: Uniform and adaptive levels of detail

Thus, a sequence of k successive edge collapse transformations yields a sequence of successively simpler meshes:

$$M^k \overset{collapse_{k-1}}{\rightarrow} M^{k-1} \overset{collapse_{k-2}}{\rightarrow} \ldots M^1 \overset{collapse_0}{\rightarrow} M^0 \tag{1}$$

We can retrieve the successively higher detail meshes from the simplest mesh M^0 by using a sequence of vertex-split transformations that are dual to the corresponding edge collapse transformations:

$$M^0 \overset{split_0}{\rightarrow} M^1 \overset{split_1}{\rightarrow} \ldots M^{k-1} \overset{split_{k-1}}{\rightarrow} (\hat{M} = M^k) \tag{2}$$

Hoppe [25] refers to $(M^0, \{split_0, split_1, \ldots, split_{k-1}\})$ as a *progressive mesh* representation. Progressive meshes present a novel approach to storing, rendering, and transmitting meshes by using a continuous-resolution representation. However we feel that there is some room for improve-

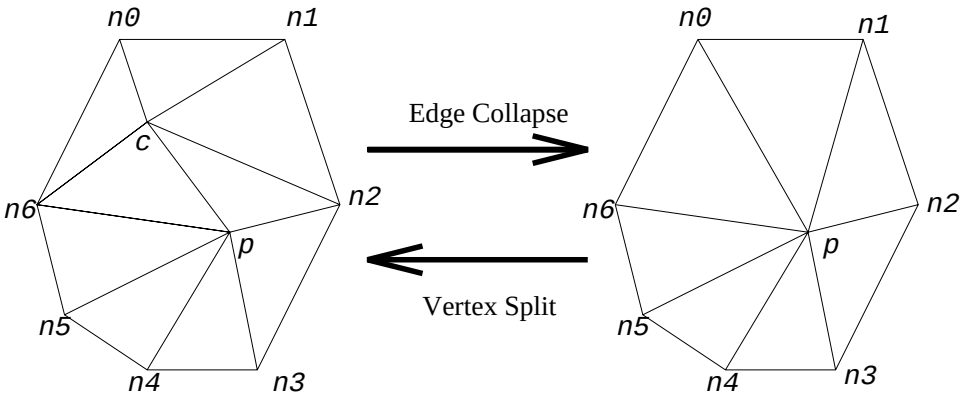


Figure 2: Edge collapse and vertex split

ment in adapting them for performing selective refinement in an efficient manner. In particular, following issues have not yet been addressed by progressive meshes:

- The sequence of edge collapses is aimed at providing good approximations M^i to $(\hat{M} = M^k)$. However, if a sequence of meshes M^i are good approximations to $\hat{M}$ under some distance metric, it does not necessarily mean that they also provide a "good" sequence of edge collapse transformations for selective refinement. Let us consider a two-dimensional analogy of a simple polygon as shown in Figure 3. Assume that vertices $v_0, v_6, v_7,$ and v_8 are "important" vertices (under say some perceptual criteria) and can not be deleted. An approach that generates approximations based on minimizing distances to the original polygon will collapse vertices in the order $v_1 \rightarrow v_2, v_2 \rightarrow v_3, v_3 \rightarrow v_4, v_4 \rightarrow v_5, v_5 \rightarrow v_6$ to get a coarse polygon (v_0, v_6, v_7, v_8). Then if selective refinement is desired around vertex v_1, vertices v_6, v_5, v_4, v_3, v_2 will need to be split in that order before one can get to vertex v_1. An approach that was more oriented towards selective refinement might have collapsed $v_1 \rightarrow v_2, v_3 \rightarrow v_4, v_5 \rightarrow v_6, v_2 \rightarrow v_4, v_4 \rightarrow v_6$ for better adaptive results, even though the successive approximations are not as good as the previous ones under the distance metric.

- Since the edge collapses are defined in a linear sequence, the total number of child links to be traversed before reaching the desired node is $O(n)$.

- No efficient method for incrementally updating the selective refinements from one frame to the next is given. The reverse problem of selective refinement – selective simplification too is not dealt with.

In this paper we provide a solution to the above issues with the aim of performing real-time adaptive simplifications and refinements. We define a criterion for performing edge collapses that permits adaptive refinement around any vertex. Instead of constructing a series of sequential edge collapses we construct a *merge tree* over the vertices of mesh $\hat{M}$ so that one can reach any child vertex in $O(\log n)$ links. We then describe how one can perform incremental updates within this tree to exploit frame-to-frame coherence, view-dependent illumination, and visibility computations using both scalar and vector attributes.

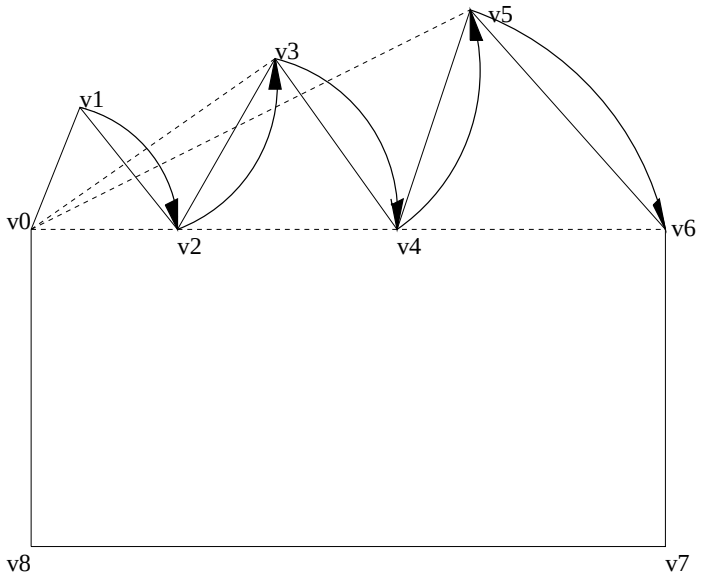


Figure 3: Good versus efficient selective simplification

3 Simplification with Image-Space Feedback

Level-of-detail-based rendering has thus far emphasized object-space simplifications with minimal feedback from the image space. The feedback from the image space has been in the form of very crude heuristics such as the ratio of the screen-space area of the bounding box of the object to the distance of the object from the viewer. As a result, one witnesses coarse image-space artifacts such as the distracting "popping" effect when the object representation changes from one level of detail to the next [23]. Attempts such as alpha-blending between the old and the new levels of detail during such transitions serve to minimize the distraction at the cost of rendering two representations. However alpha blending is not the solution to this problem since it does not address the real cause – lack of sufficient image-space feedback to select the appropriate local level of detail in the object space; it merely tries to cover-up the distracting artifacts.

Increasing the feedback from the image space allows one to make better choices regarding the level of detail selection in the object-space. We next outline some of the ways in which image-space feedback can influence the level of detail selection in the object-space.

3.1 Local Illumination

Increasing detail in a direction perpendicular to, and proportional to, the illumination gradient across the surface is a good heuristic [2]. This allows one to have more detail in the regions where the illumination changes sharply and therefore one can represent the highlights and the sharp shadows well. Since surface normals play an important role in local illumination one can take advantage of the coherence in the surface normals to build a hierarchy over a continuous resolution model that allows one to capture the local illumination effects well. We outline in Section 4.3 how one can build such a hierarchy.

3.2 Screen-Space Projections

Decision to keep or collapse an edge should depend upon the length of its screen-space projection instead of its object-space length. At a first glance this might seem very hard to accomplish in real-time since this could mean checking for the projected lengths of all edges at every frame. However, usually there is a significant coherence in the ratio of the image-space length to the object-space length of edges across the surface of an object and from one frame to the next. This makes it possible to take advantage of a hierarchy built upon the the object-space edge lengths for an object. We use an approximation to the screen-space projected edge length that is computed from the object-space edge length. We outline in Section 4.2 how one can build such a hierarchy.

3.3 Visibility Culling

During interactive display of any model there is usually a significant coherence between the visible regions from one frame to the next. This is especially true of the back-facing polygons that account for almost half the total number of polygons and do not contribute anything to the visual realism. A hierarchy over a continuous resolution representation of an object allows one to significantly simplify the invisible regions of an object, especially the back-facing ones. This view-dependent visibility culling can be implemented in a straightforward manner using the hierarchy on vertex normals discussed in Section 4.3.

3.4 Silhouette boundaries

Silhouettes play a very important role in perception of detail. Screen-space projected lengths of silhouette edges (i.e., edges for which one of the adjacent triangles is visible and the other is invisible), can be used to very precisely quantify the amount of smoothness of the silhouette boundaries. A hierarchy built upon a continuous-resolution representation of a object allows one to do this efficiently.

4 Construction of Merge Tree

We would like to create a hierarchy that provides us a continuous-resolution representation of an object and allows us to perform real-time adaptive simplifications over the surface of an object based upon the image-space feedback mechanisms mentioned in Section 3. Towards this end we implement a *merge tree* over the vertices of the original model. In our current implementation, the merge tree stores the edge collapses in a hierarchical manner. However, as we discuss in Section 7 the concept of a merge tree is a very general one and it can be used with other local simplification approaches as well. Note that the merge tree construction is done as an off-line preprocessing step before the interactive visualization.

4.1 Basic Approach

In Figure 2, the vertex c is merged with the vertex p as a result of collapsing the edge (pc). Conversely, during a vertex split the vertex c is created from the vertex p. We shall henceforth refer to

c as the child vertex of the parent vertex p. The merge tree is constructed upwards from the high-detail mesh $\hat{M}$ to a low-detail mesh M^0 by storing these parent-child relationships in a hierarchical manner over the surface of an object.

At each level l of the tree we determine parent-child relationships amongst as many vertices at level l as possible. In other words, we try to determine all vertices that can be safely merged based on criterion defined in Section 4.4. The vertices that are determined to be the children remain at level l and all the other vertices at level l are promoted to level $l+1$. Note that the vertices promoted to level $l+1$ are a proper superset of the parents of the children left behind at level l. This is because there are vertices at level l that are neither parents nor children. We discuss this in greater detail in the context of *regions of influence* later in this section. We apply the above procedure recursively at every level until either (a) we are left with a user-specified minimum number of vertices, or (b) we cannot establish any parent-child relationships amongst the vertices at a given level. Case (b) can arise because in determining a parent-child relationship we are essentially collapsing an edge and not all edge collapses are considered legal. For a detailed discussion on legality of edge collapses the interested reader can refer to [26]. Since in an edge collapse only one vertex merges with another, our merge tree is currently implemented as a binary tree.

To construct a balanced merge tree we note that the effects of an edge collapse are local. Let us define the *region of influence* of an edge (v_0, v_1) to be the union of triangles that are adjacent to either v_0 or v_1 or both. The region of influence of an edge is the set of triangles that can change as an edge is gradually collapsed to a vertex, for example, in a morphing. Thus, in Figure 2 as vertex c merges to vertex p, (or p splits to c), the changes to the mesh are all limited to within the region of influence of edge (pc) enclosed by $n_0, n_1, \ldots n_6$. Note that all the triangles in region of influence will change if vertices p and c are merged to form an intermediate vertex, say $(p+c)/2$. In our current implementation, the position of the intermediate vertex is the same as the position of the parent vertex p. However our data-structures can support other values of the intermediate vertex too. Such values could be used, for example, in creating intermediate morphs between two level-of-detail representations.

To create a reasonably balanced merge tree we try to collapse as many edges as possible at each level such that there are no common triangles in their respective regions of influence. Since this step involves only local checks, we can accomplish this step in time linear in the number of triangles at this level. If we assume that the average degree (i.e. the number of neighboring triangles) of a vertex is 6, we can expect the number of triangles in an edge's region of influence to be 10. After the collapse this number of triangles reduces to 8. Thus the number of triangles can be expected to reduce roughly by a factor of $4/5$ from a higher-detail level to a lower-detail level. Thus, in an ideal situation, the total time to build the tree will be given by $n + \frac{4n}{5} + \frac{16n}{25} + \ldots = O(n)$. However, this assumes that we arbitrarily choose the edges to be collapsed. A better alternative is to sort the edges by their edge lengths and collapse the shortest edges first. Collapsing an edge causes the neighboring edges to change their lengths. However as mentioned above, since changes are local we can maintain the sorted edge lengths in a heap for efficient updates. With this strategy one can build the merge tree in time $O(n \log n)$.

4.2 Scalar Subtree Attributes

To allow real-time refinement and simplification we can store at every parent node (i.e. a node that splits off a child vertex) of the merge tree, a range of scalar attributes of the children in the subtree below it. Then image-space feedback can be used to determine if this range of scalar attributes merits a refinement of this node or not. We explain this process of incremental refinement and simplification in greater details in Section 5.1.

In our current implementation every merge tree node v stores the Euclidean distances to its child and parent that determine when v's child will merge into v and when v will merge into its parent. The former is called the *downswitch distance* and the latter is called the *upswitch distance*. These distances are built up during the merge tree creation stage. If the maximum possible screen-space projection of the downswitch distance at the vertex v in the object space is greater than some pre-set threshold, we permit refinement at v. However, if the maximum possible screen-space projection of the upswitch distance at v in the object space is less than the threshold, it means that this region occupies very little screen space and can be simplified.

4.3 Vector Subtree Attributes

Our implementation also allows incremental simplification and refinement based upon the coherences of the surface normals. This allows us to implement view-dependent real-time simplifications based on local illumination and visibility. The regions with low intensity gradients are drawn in lower detail, while the regions with high intensity gradients are drawn in higher detail. Similarly, regions of the object that are back-facing are drawn at a much lower detail then the front-facing regions.

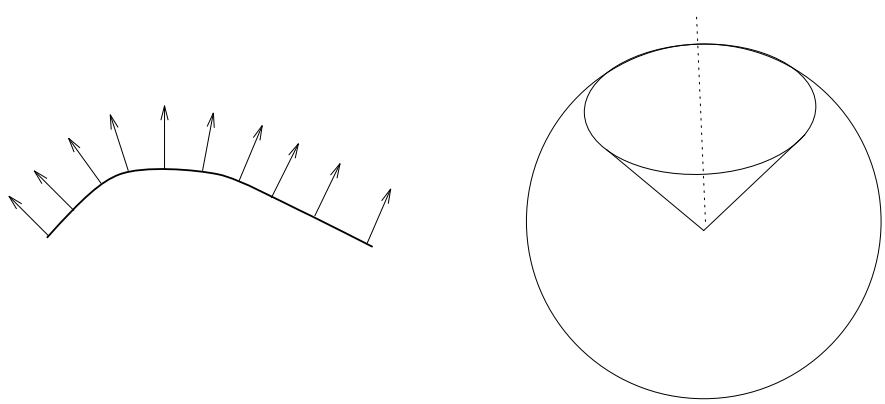
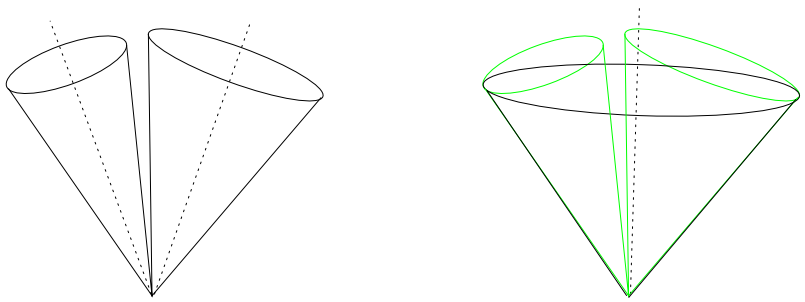


Figure 4: Bounding cone for normal vectors

Since we are using frame-to-frame coherences in computing the levels of detail we need to adopt a data-structure that represents the variation in the normal vectors amongst all the descendents of any given vertex. To identify a possible representation, let us consider the idea behind a Gauss map. A Gauss map is a mapping of the unit normals to the corresponding points on the surface of a unit sphere. Thus, all the normal variations in a subtree will be represented by a closed and connected region on the surface of a sphere using a Gauss map. To simplify the computations involved, we have decided to approximate such regions by circles on the surface of the unit sphere,

i.e. bounding cones containing all the subtree normal vectors. This is demonstrated in Figure 4 where the normal vectors in the surface shown on the left are contained within the cone (i.e. a circle on the Gauss map) on the right.

At the leaf-level, each vertex is associated with a normal-cone whose axis is given by its normal vector and whose angle is zero. As two vertices merge, the cones of the child and parent vertices are combined into a new normal cone that belongs to the parent vertex at the higher level. The idea behind this merging of cones is shown in Figure 5.



a) Child Cones b)Merged Parent Cone

Figure 5: Cone merging

4.4 Merge Tree Dependencies

By using techniques outlined in Section 5.1, one can determine which subset of vertices is sufficient to reconstruct an adaptive level-of-detail for a given object. However, it is not simple to define a triangulation over these vertices and guarantee that the triangulation will not "fold back" on itself or otherwise represent a non-manifold surface (even when the original was not so). Figure 6 shows an example of how an undesirable folding in the adaptive mesh can arise even though all the edge collapses that were determined statically were correct. A shows the initial state of the mesh. While constructing the merge tree, we first collapsed vertex v_2 to v_1 to get mesh B and then collapsed vertex v_3 to v_4 to get mesh C. Now suppose at run-time we determined that we needed to display vertices v_1, v_2, and v_4 and could possibly collapse vertex v_3 to v_4. However, if we collapse v_3 to v_4 directly, as in mesh D, we get a mesh fold where there should have been none. One could devise elaborate procedures for checking and preventing such mesh fold-overs at run-time. However, such checks involve several floating-point operations and are too expensive to be performed on-the-fly.

To solve the above problem we introduce the notion of dependencies amongst the nodes of a merge tree. Thus, the collapse of an edge e is permitted only when all the vertices defining the boundary of the region of influence of the edge e exist and are adjacent to the edge e. As an example, consider Figure 2. Vertex c can merge with vertex p only when the vertices $n_0, n_1, \ldots, n_k$ exist and are adjacent to p and c. From this we determine the following edge collapse dependencies, restricting the level difference between adjacent vertices:

1. c can collapse to p, only when n_0, n_1, ..., n_k are present as neighbors of p and c for display.

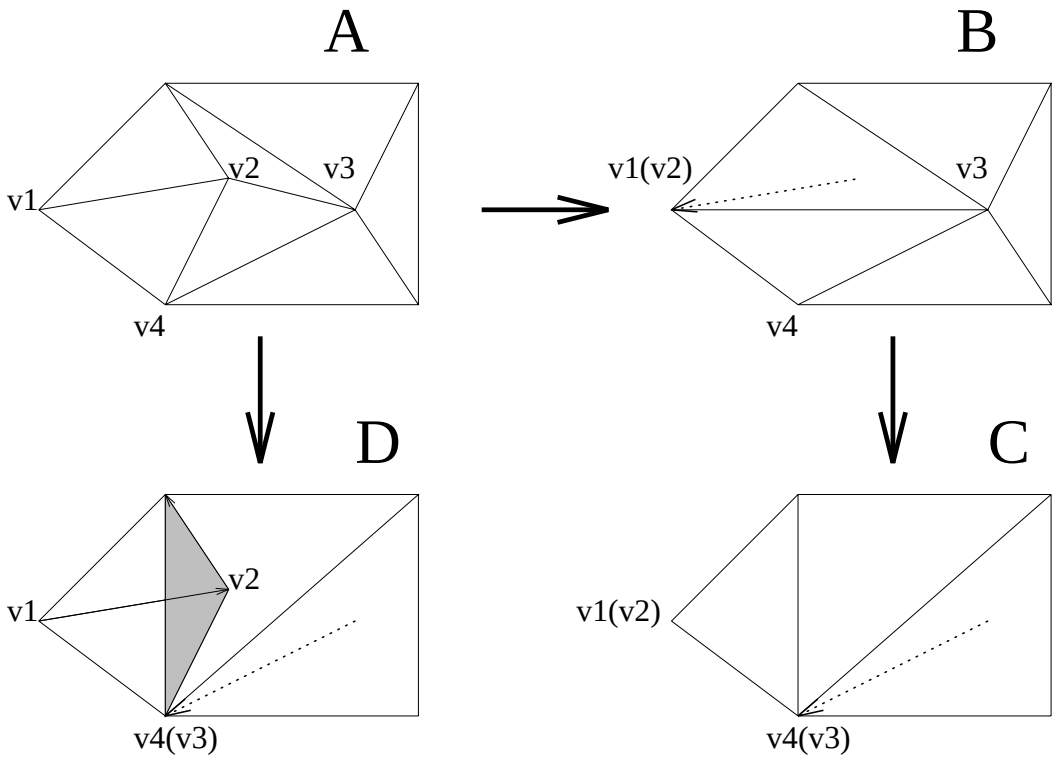


Figure 6: Mesh folding problem

2. $n_0, n_1, \ldots, n_k$ can not merge with other vertices, unless c first merges with p.

Similarly, to make a safe split from p to p and c, we determine the following vertex split dependency:

1. p can split to c and p, only when $n_0, n_1, \ldots, n_k$ are present as neighbors of p for display.

2. $n_0, n_1, \ldots, n_k$ can not split, unless p first splits to p and c.

The above dependencies are followed during each vertex-split or edge collapse during real-time simplification. These dependencies are easily identified and stored in the merge tree during its creation. Considering Figure 6 again, we can now see that collapse of vertex v_3 to v_4 depends upon the adjacency of vertex v_1 to v_3. If vertex v_2 is present then v_1 will not be adjacent to v_3 and therefore v_3 will not collapse to v_4. Although having dependencies might sometimes give lesser simplification than otherwise, it does have the advantage of eliminating the expensive floating-point run-time checks entirely. The basic idea behind merge tree dependencies has a strong resemblance to creating *balanced subdivisions* of quad-trees as presented by Baum *et al* in [3] where only a gradual change is permitted from regions of high simplifications to low simplifications. Details of how these merge tree dependencies are used during run-time are given in Section 5.1.

The pseudocode outlining the data-structure for a merge tree node is given in Figure 7. The pseudocode for building and traversing the merge tree is given in Figure 8. We are representing the triangular mesh by the winged-edge data-structure to maintain the adjacency information.

```
struct NODE {
        struct VERTEX  *vert   ;     /* associated vertex */
        struct NODE    *parent ;     /* parent node for merging   */
        struct NODE    *child[2] ;   /* child nodes for refinement */
        float          upswitch  ;   /* threshold to merge */
        float          downswitch;   /* threshold to refine */
        struct CONE    *cone     ;   /* range of subtree normals */
        struct VERTEX  **adj_vert ;  /* adjacent vertices */
        int            adj_num    ;  /* number of adjacent vertices */
        struct VERTEX  **depend_vert;/* dependency list for merge */
        int            depend_num   ;/* number of vertices in the */
};                                   /* dependency list */
```

Figure 7: Data-structure for a merge tree node

5 Real-Time Triangulation

Once the merge tree with dependencies has been constructed off-line it is easy to construct an adaptive level-of-detail mesh representation at run-time. Real-time adaptive mesh reconstruction involves two phases – determination of vertices that will be needed for reconstruction and determination of the triangulation amongst them. We shall refer to the vertices selected for display at a given frame as *display vertices* and triangles for display as *display triangles*. The phases for determination of display vertices and triangles are discussed next.

5.1 Determination of display vertices

In this section we outline how we determine the display vertices using the scalar and vector attribute ranges stored with the nodes of the merge tree. We first determine the *primary display vertices* using the screen-space projections and the normal vector cones associated with merge tree nodes. These are the only vertices that would be displayed if there were no triangulation constraints or mesh-folding problems. Next, from these primary display vertices we determine the *secondary display vertices* that are the vertices that need to be displayed due to merge tree dependencies to avoid the mesh fold-overs in run-time triangulations.

5.1.1 Primary Display Vertices

Screen-Space Projection

As mentioned earlier, every merge tree node v stores a Euclidean distance for splitting a vertex to its child (downswitch distance) as well as the distance at which it will merge to its parent (upswitch distance). If the maximum possible screen-space projection of the downswitch distance at the vertex v in the object space is greater than some pre-set threshold T, we permit refinement at v and recursively check the children of v. However, if the maximum possible screen-space projection of the upswitch distance at v in the object space is less than the threshold T, it means

```
/* Given a mesh, build_mergetree() constructs a list of merge trees - one
 * for every vertex at the coarsest level of detail.
 */
build_mergetree(struct MESH *mesh, struct NODE **roots)
{ struct HEAP  *current_heap, *next_heap ;
  int level ;

  current_heap = InitHeap(mesh);
  next_heap = InitHeap(nil) ;
  for ( level = 0 ; HeapSize(current_heap) > MIN_RESOLUTION_SIZE; level ++
  { while ( HeapSize(current_heap) > 0 )
       { edge = ExtractMinEdge(current_heap);
         node = CreatNode(edge);
         SetDependencies(node);
         SetCone(node);            /* Set vector attributes */
         SetSwitchDistances(node); /* Set scalar attributes */
         InsertHeap(next_heap, node);
       }
    FreeHeap(current_heap);
    current_heap = next_heap ;
    next_heap = InitHeap(nil);
  }
  FlattenHeap(roots, current_heap);
}

/* Given a list of nodes of the merge tree that were active in the previous
 * frame, traverse_mergetree() constructs a list of new merge tree nodes by
 * either refining or simplifying each of the active merge tree nodes.
 */
traverse_mergetree(struct NODE **current_list,
                   struct VIEW view, struct LIGHTS *lights )
{ int switch ;

  for each node in current_list do
  { switch = EvalSwitch(node, view, lights);
    if ( switch == REFINE )
       RefineNode(node);
    else if ( switch == SIMPLIFY )
       MergeNode(node);
  }
}
```

Figure 8: Pseudocode for building and traversing the merge tree

that this region occupies very little screen space and can be simplified, so we mark v as *inactive* for display.

Normal Vectors

We need to determine the direction and the extent of the normal vector orientation within the subtree rooted at a display vertex, with respect to the viewing direction as well as light source, to accomplish view-dependent local illumination and visibility-based culling.

To determine silhouettes and the back-facing regions of an object, we check to see if the normal vector cone at a vertex lies entirely in a direction away from the viewer. If so, this vertex can be marked inactive for display. If not, this vertex is a display vertex and is a candidate for further refinement based on other criteria such as screen-space projection, illumination gradient, and silhouette smoothness. In such a case we recursively check its children. The three possible cases are shown in Figure 9.

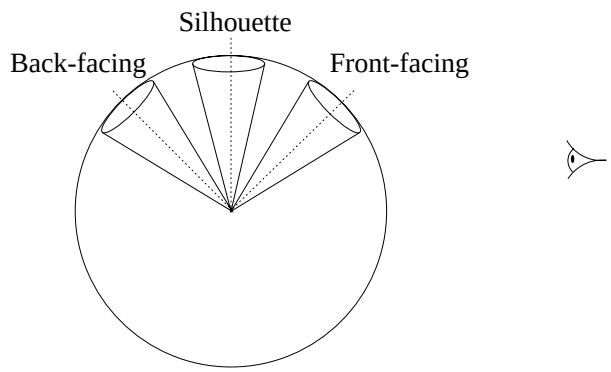


Figure 9: Selective refinement and simplification using normal cones

Similarly, for normal-based local illumination, such as Phong illumination, we use the range of the reflection vectors and determine whether they contain the view direction or not to determine whether to simplify or refine a given node of the merge tree.

We follow the procedures outlined above to select all those vertices for display that either (a) are leaf nodes and none of their parents have been marked as inactive, or (b) have their immediate child marked as inactive. This determines the list of primary display vertices.

5.1.2 Secondary Display Vertices

We follow the merge dependencies from the list of primary display vertices to select the final set of display vertices in the following manner. If a vertex v is in the initial list of display vertices and for it to be created (via a vertex split), the vertices $v_{d_0}, v_{d_1}, \ldots, v_{d_k}$ had to be present, we add the vertices $v_{d_0}, v_{d_1}, \ldots, v_{d_k}$ to the list of display vertices and recursively consider their dependencies. We continue this process until no new vertices are added.

When determining the vertices for display in frame $i+1$ we start from the vertex list for display used in frame i. We have found a substantial frame-to-frame coherence and the vertex display list does not change substantially from one frame to the next. There are minor local changes in the display list on account of vertices either refining or merging with other vertices. These are easily captured by either traversing the merge tree up or down from the current vertex position. The

scalar and vector attribute ranges stored in merge tree nodes can be used to guide refinements if the difference in the display vertex lists from one frame to the next becomes non-local for any reason. We compute the list of display vertices for first frame by initializing the list of display vertices for frame 0 to be all the vertices in the model and then proceeding as above.

5.2 Determination of display triangles

If the display triangles for frame i are known, determination of the display triangles for frame $i+1$ proceeds in an interleaved fashion with the determination of display vertices for frame $i+1$ from frame i. Every time a display vertex of frame i merges in frame $i+1$ we simply delete and add appropriate triangles to the list of display triangles as shown in Figure 10. The case where a display vertex in frame i splits for frame $i+1$ is handled analogously. Incremental determination of display triangles in this manner is possible because of the dependency conditions mentioned in Section 4.4. The list of display triangles for the first frame is obtained by initializing the list for frame 0 to be all the triangles in the model and then following the above procedure.

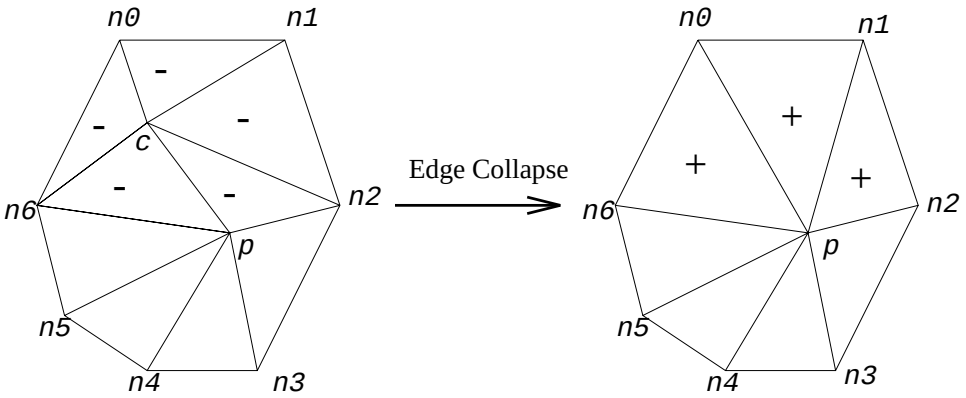
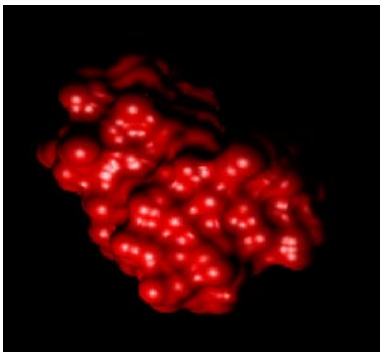


Figure 10: Display triangle determination

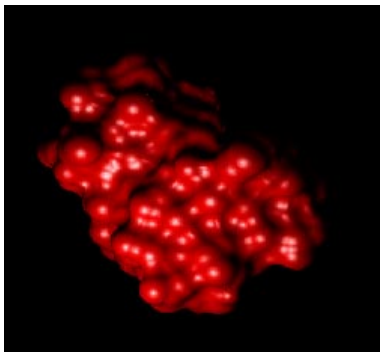
6 Results and Discussion

We have tried our implementation on several large triangulated models and have achieved encouraging results. These are summarized in Table 1 The images of teapot, bunny, crambin, phone, sphere, buddha, and dragon models that were produced for the times in Table 1 are shown in Figures 1, 11, 13, and 14 respectively. All of these timings are in milliseconds on a Silicon Graphics Onyx with RE2 graphics, a 194MHz R10000 processor, and 640MB RAM. It is easy to see that the time to traverse the merge tree and construct the list of triangles to be displayed from frame to frame is relatively small. This is because of our incremental computations that exploit image-space, object-space, and frame-to-frame coherences. The above times hold as the user moves through the model or moves the lights around. The triangulation of the model changes dynamically to track the highlights as well as the screen-space projections of the faces.

As can be seen from the merge tree depths, the trees are not perfectly balanced. However, they are still within a small factor of the optimal depths. This factor is the price that has to be paid

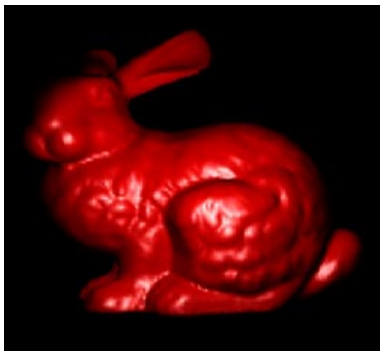


Highest detail Crambin surface



Simplified Crambin surface

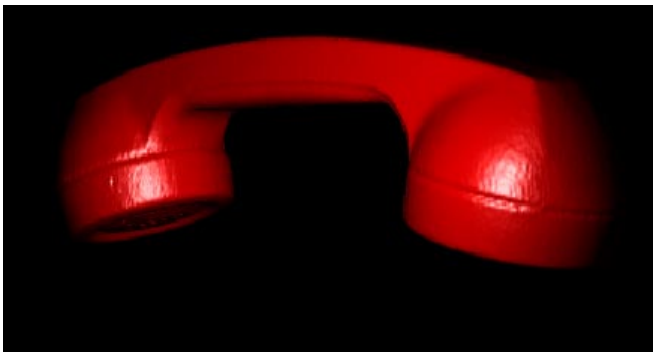
Highest detail Bunny



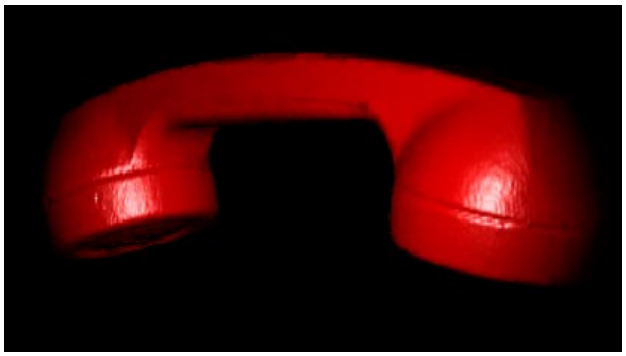
Simplified Bunny

Highest detail Teapot

Simplified Teapot

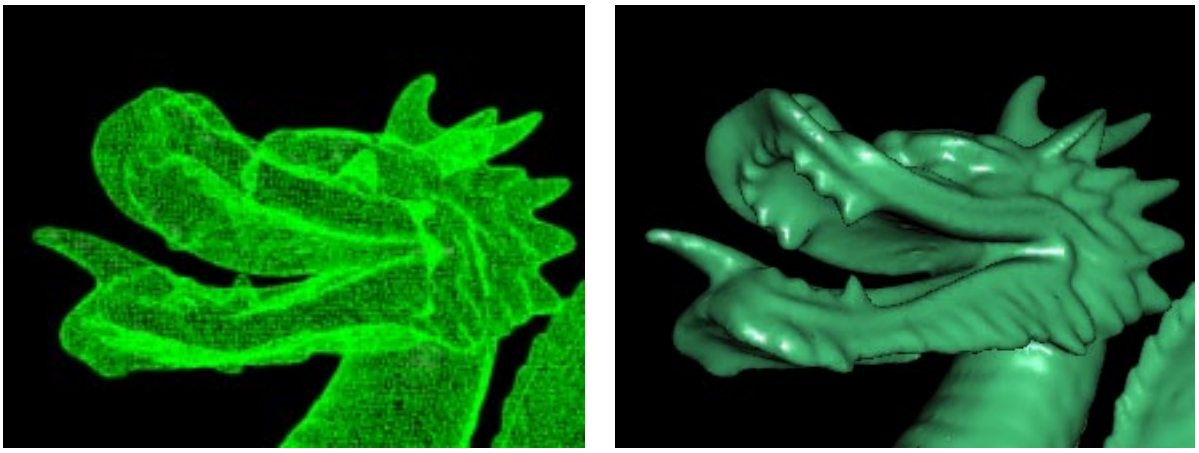


Highest detail Phone

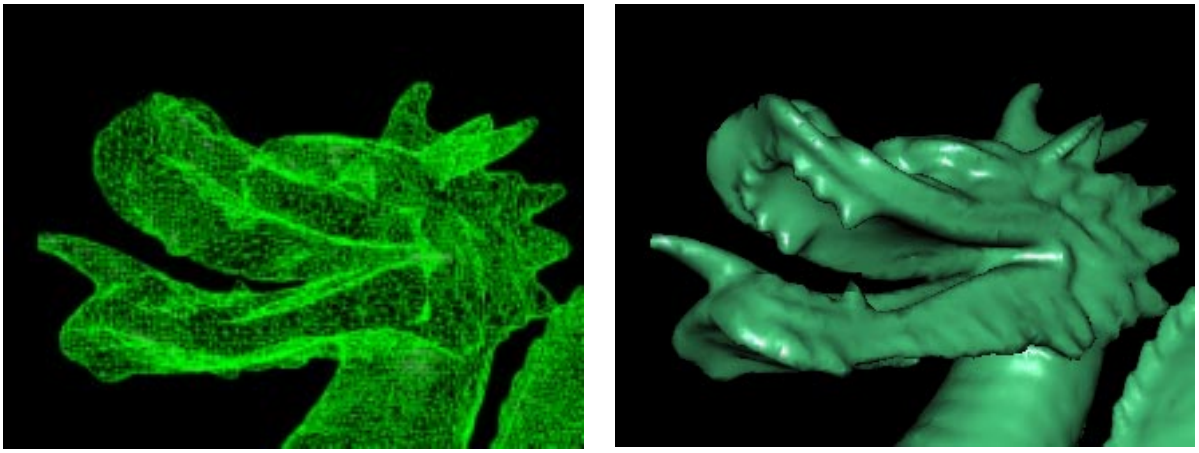


Simplified Phone

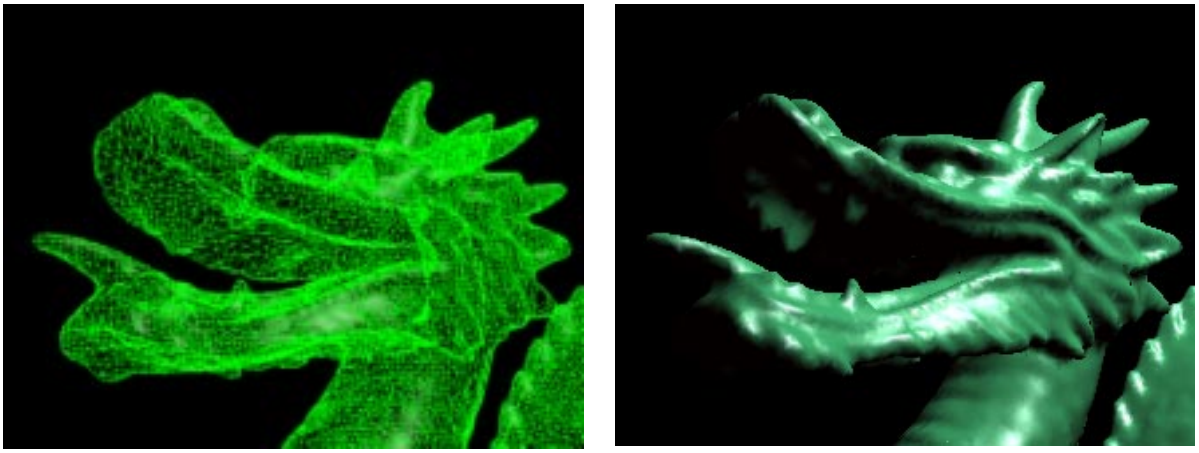
Figure 11: Dynamic adaptive simplification



Highest detail model – bottom light source



Dynamic adaptive simplification – top light source

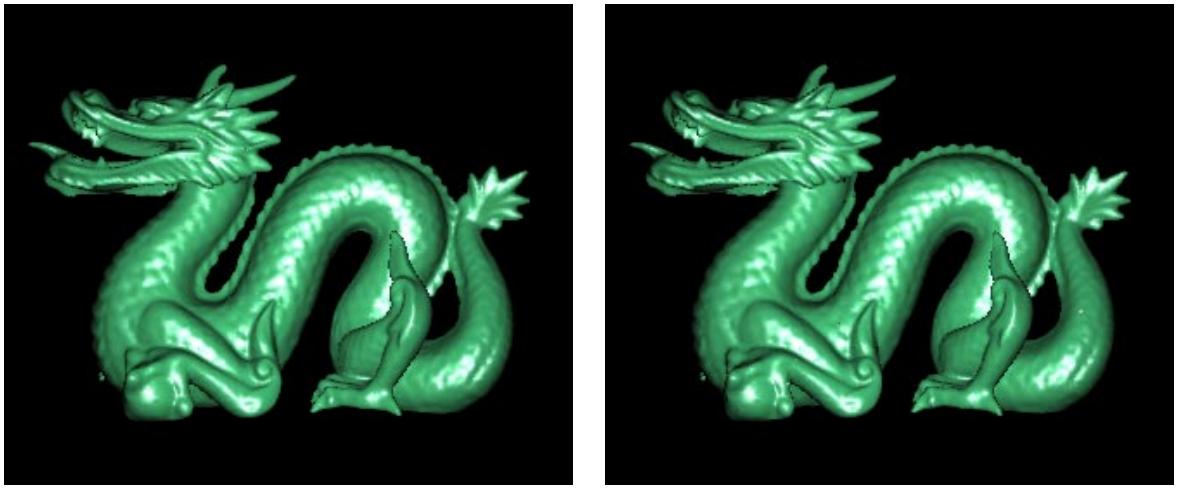


Dynamic adaptive simplification – top light source

Figure 12: Dynamic adaptive simplification for the head of the Dragon

Highest detail Simplified

Figure 13: Dynamic adaptive simplification for the Buddha



Highest detail Simplified

Figure 14: Dynamic adaptive simplification for the Dragon

Dataset	Highest Detail		Adaptive Detail					Reduction Ratio	
	Display Tris	Display Time	Display Tris	Tree Levels	Traverse Tree	Display Time	Total Time	Display Tris	Display Time
Teapot	3751	57	1203	36	10	17	27	32.0%	47.3 %
Sphere	8192	115	994	42	8	16	24	12.1%	20.8 %
Bunny	69451	1189	13696	65	157	128	285	19.7%	23.9 %
Crambin	109884	1832	19360	61	160	194	354	17.6%	19.3 %
Phone	165963	2629	14914	63	112	144	256	8.9 %	9.7 %
Dragon	202520	3248	49771	66	447	394	842	24.5%	25.9 %
Buddha	293232	4618	68173	69	681	546	1227	23.2%	26.5 %

Table 1: Adaptive level of detail generation times

to incorporate dependencies and avoid the expensive run-time floating-point checks for ensuring good triangulations. For each dataset, we continued the merge tree construction till 8 or fewer vertices were left. As expected, the factor by which the number of vertices decreases from one level to the next tapers off as we reach lower-detail levels since there are now fewer alternatives left to counter the remaining dependency constraints. As an example, for sphere, only 64 vertices were present at level 30 and it took another 12 levels to bring down the number to 8. If the tree depth becomes a concern one can stop sooner, trading-off the tree traversal time for the display time.

An interesting aspect of allowing dependencies in the merge tree is that one can now influence the characteristics of the run-time triangulation based upon static edge-collapse decisions during pre-processing. As an example, we have implemented avoidance of slivery (long and thin) triangles in the run-time triangulation. As Guéziec [20], we quantify the quality of a triangle with area a and lengths of the three sides $l_0, l_1,$ and l_2 based on the following formula:

$$\text{Quality} = \frac{4\sqrt{3}a}{l_0^2 + l_1^2 + l_2^2} \tag{3}$$

Using Equation 3 the quality of a degenerate triangle evaluates to 0 and that of an equilateral triangle to 1. We classify all edge collapses that result in slivery triangles to be invalid, trading-off quantity (amount of simplification) for quality.

One of the advantages of using normal cones for back-face simplification and silhouette definition is that it allows the graphics to focus more on the regions of the object that are perceptually more important. Thus, for instance, for generating the given image of the molecule crambin, 8729 front-facing vertices were traversed as compared to 3361 backfacing vertices; 1372 were classified as silhouette vertices. Similarly, for the model of phone, our approach traversed 6552 front-facing vertices compared to only 1300 backfacing vertices; 900 were classified as silhouette vertices.

Clearly, there is a tradeoff here between image-quality and amount of simplification achieved. The results for our simplifications given in this section correspond to the images that we thought were comparable to the images from the original highest detail models. Higher levels of simplifications (that are faster to incrementally compute and display) with correspondingly lower quality

images are obviously possible, allowing easy implementations of progressive refinement for display.

7 Conclusions and Future Work

We have outlined a simple approach to maintain dynamically adaptive level of detail triangulations. Crucial to this approach is the notion of merge trees that are computed statically and are used during run-time to take advantage of the incremental changes in the triangulation. In our current implementation we are using the method of edge collapses. However the idea behind merge trees is pretty general and can be used in conjunction with other local heuristics for simplification such as vertex deletion and vertex collapsing. We plan to study some of these other heuristics in the future and compare them with our current implementation that uses edge collapses.

At present we do not store color ranges at the nodes of the merge tree. Storing and using these should improve the quality of the visualizations produced using merge trees even further. Also of some interest will be techniques that create better balanced merge trees while still incorporating dependencies. We plan to investigate these issues further.

Of course, our approach also makes dynamically-specified manual simplifications possible, where the user can interactively specify the amounts of approximation desired at various regions of the object. Using this, certain parts of the object can be rendered at lower or higher details than otherwise. However, in this paper we have only considered automatic object simplifications during interactive display.

Acknowledgements

We would like to acknowledge several useful discussions with Arie Kaufman and Greg Turk. We should like to thank Greg Turk, Marc Levoy, and the Stanford University Computer Graphics laboratory for generously sharing models of the bunny, the phone, the dragon, and the happy Buddha. We should also like to acknowledge the several useful suggestions made by the anonymous reviewers that have helped improve the presentation of this paper. This work has been supported in part by the National Science Foundation CAREER award CCR-9502239 and a fellowship from the Fulbright/Israeli Arab Scholarship Program.

References

[1] J. M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, NC 27599-3175, 1990.

[2] J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. In Rich Riesenfeld and Carlo Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, No. 2, pages 41–50, March 1990.

[3] D. R. Baum, Mann S., Smith K. P., and Winget J. M. Making radiosity usable: Automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions. *Computer Graphics: Proceedings of SIGGRAPH'91*, 25, No. 4:51–60, 1991.

[4] L. Bergman, H. Fuchs, E. Grant, and S. Spach. Image rendering by adaptive refinement. In *Computer Graphics: Proceedings of SIGGRAPH'86*, volume 20, No. 4, pages 29–37. ACM SIGGRAPH, 1986.

[5] Jim F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78*, pages 286–292. ACM, 1978.

[6] Jim F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *CACM*, 19(10):542–547, October 1976.

[7] S. Chen. Quicktime VR – an image-based approach to virtual environment navigation. In *Computer Graphics Annual Conference Series (SIGGRAPH '95)*, pages 29–38. ACM, 1995.

[8] S. Chen and L. Williams. View interpolation for image synthesis. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 279–288, August 1993.

[9] J. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[10] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks, Jr., and W. V. Wright. Simplification envelopes. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 119 – 128. ACM SIGGRAPH, ACM Press, August 1996.

[11] M. Cosman and R. Schumacker. System strategies to optimize CIG image content. In *Proceedings of the Image II Conference*, Scottsdale, Arizona, June 10–12 1981.

[12] F. C. Crow. A more flexible image generation environment. In *Computer Graphics: Proceedings of SIGGRAPH'82*, volume 16, No. 3, pages 9–18. ACM SIGGRAPH, 1982.

[13] L. Darsa, B. Costa, and A. Varshney. Navigating static environments using image-space simplification and morphing. In *Proceedings, 1997 Symposium on Interactive 3D Graphics*, 1997.

[14] T. D. DeRose, M. Lounsbery, and J. Warren. Multiresolution analysis for surface of arbitrary topological type. Report 93-10-05, Department of Computer Science, University of Washington, Seattle, WA, 1993.

[15] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of SIGGRAPH 95 (Los Angeles, California, August 6–11, 1995)*, Computer Graphics Proceedings, Annual Conference Series, pages 173–182. ACM SIGGRAPH, August 1995.

[16] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH 93 (Anaheim, California, August 1–6, 1993)*, Computer Graphics Proceedings, Annual Conference Series, pages 247–254. ACM SIGGRAPH, August 1993.

[17] N. Greene. Hierarchical polygon tiling with coverage masks. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 65 – 74. ACM Siggraph, ACM Press, August 1996.

[18] Ned Greene and M. Kass. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240, 1993.

[19] M. H. Gross, R. Gatti, and O. Staadt. Fast multiresolution surface meshing. In G. M. Nielson and D. Silver, editors, *IEEE Visualization '95 Proceedings*, pages 135–142, 1995.

[20] A. Guéziec. Surface simplification with variable tolerance. In *Proceedings of the Second International Symposium on Medical Robotics and Computer Assisted Surgery, MRCAS '95*, 1995.

[21] B. Hamann. A data reduction scheme for triangulated surfaces. *Computer Aided Geometric Design*, 11:197–214, 1994.

[22] T. He, L. Hong, A. Varshney, and S. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–184, June 1996.

[23] James Helman. Graphics techniques for walkthrough applications. In *Interactive Walkthrough of Large Geometric Databases, Course Notes 32, SIGGRAPH '95*, pages B1–B25, 1995.

[24] P. Hinker and C. Hansen. Geometric optimization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings Visualization '93*, pages 189–195, October 1993.

[25] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 99 – 108. ACM SIGGRAPH, ACM Press, August 1996.

[26] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proceedings of SIGGRAPH 93 (Anaheim, California, August 1–6, 1993)*, Computer Graphics Proceedings, Annual Conference Series, pages 19–26. ACM SIGGRAPH, August 1993.

[27] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings, 1995 Symposium on Interactive 3D Graphics*, pages 105 – 106, 1995.

[28] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 Symposium on Interactive 3D Computer Graphics*, pages 95–102, 1995.

[29] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *Computer Graphics Annual Conference Series (SIGGRAPH '95)*, pages 39–46. ACM, 1995.

[30] J. Rohlf and J. Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, July 1994.

[31] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993.

[32] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *Computer Graphics: Proceedings SIGGRAPH '92*, volume 26, No. 2, pages 65–70. ACM SIGGRAPH, 1992.

[33] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 75–82. ACM SIGGRAPH, ACM Press, August 1996.

[34] S. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics: Proceedings of SIGGRAPH'91*, 25, No. 4:61–69, 1991.

[35] G. Turk. Re-tiling polygonal surfaces. In *Computer Graphics: Proceedings SIGGRAPH '92*, volume 26, No. 2, pages 55–64. ACM SIGGRAPH, 1992.

[36] A. Varshney. Hierarchical geometric approximations. Ph.D. Thesis TR-050-1994, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1994.

Variable-Precision Rendering

Xuejun Hao
hao@cs.umd.edu

Amitabh Varshney
varshney@cs.umd.edu

Department of Computer Science and UMIACS
University of Maryland at College Park
College Park, MD 20742
http://www.cs.umd.edu/~{hao, varshney}

Abstract

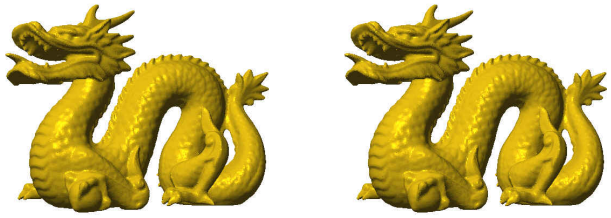
We propose the idea of using variable-precision geometry transformations and lighting to accelerate 3D graphics rendering. Multiresolution approaches reduce the *number* of primitives to be rendered; our approach complements the multiresolution techniques as it reduces the *precision* of each graphics primitive. Our method relates the minimum number of bits of accuracy required in the input data to achieve a desired accuracy in the display output. We achieve speedup by taking advantage of (a) SIMD parallelism for arithmetic operations, now increasingly common on modern processors, and (b) spatial-temporal coherence in frame-to-frame transformations and lighting. We show the results of our method on datasets from several application domains including laser-scanned, procedural, and mechanical CAD datasets.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Hierarchy and geometric transformations; I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types.

Additional Keywords: hierarchical rendering, levels of detail, variable-precision rendering, view-dependent rendering.

1 Introduction

As the complexity of visualization datasets has increased beyond the interactive rendering capabilities of the graphics hardware, research in graphics acceleration has engendered several novel techniques that reconcile the conflicting goals of scene realism and interactivity. These techniques can be broadly classified into two lines of research. The first line of research includes techniques such as multiresolution rendering and visibility-based culling. Such techniques operate by reducing the number of graphics primitives to be rendered based on viewing and illumination parameters, such that there are minimal visually discernible differences between viewing higher complexity and lower complexity scenes. Orthogonal to these advances, we have been witnessing another line of research whose goal is to reduce the precision of each graphics primitive being rendered. Recently, reduction in precision of the object properties such as colors [11, 33], normals [7, 34], and vertex coordinates [16, 17] has been successfully attempted. The contribution of this paper lies in merging these two lines of research for variable-precision, view-dependent rendering.



(a) Floating Point (32 bits/vertex coordinate)

(b) Variable Precision (Average 7.6 bits/vertex coordinate)

Figure 1: Variable Precision Rendering

Most transformations and lighting for graphics primitives are currently carried out at full floating-point precision only to be later converted to fixed-point representation during the rasterization phase. An argument can be made that such high accuracy during geometry transformation and lighting stage sometimes exceeds even the display accuracy and thus causes several bits worth of unnecessary precision computation. We are currently witnessing four important trends in 3D graphics that have increased the need for variable-precision rendering:

1. View-dependent Rendering: View-dependent rendering has already introduced the concept of rendering different regions of a scene at varying geometric, illumination, and texture detail [14, 18, 32] based on their perceptual significance. A natural extension of this approach is to render each object at the precision appropriate for it. Under a perspective projection, objects that are close to the observer need more bits of precision than objects that are far.

2. Bounded Dynamic Range: Most graphics datasets have limited dynamic range. For instance, biomolecular datasets such as protein structures are determined using X-ray crystallography, NMR experiments, and gel electrophoresis. All of these methods have their accuracy limitations. Bounded input data accuracy with limited dynamic range also occurs in volumetric, range, and image-based datasets. Even in CAD datasets, the accuracy of the model is often limited by numeric round-off errors during intersection computations and precision limitations of acquisition methods which are greater than one in a million [21].

3. Processor-Level Support: With rapid growth in the size of the 3D datasets, geometry processing (transformation and lighting) has become a significant computational component of the 3D

graphics pipeline. To partly alleviate such computations in graphics and image processing, a variety of matrix math extensions to the CPU instruction sets have emerged: Intel's Pentium II with MMX and Pentium III with SSE, AMD's K6/Athlon with 3DNow!, and the Motorola PowerPC G4 with AltiVec. All of these instruction sets take advantage of SIMD (single-instruction multiple-data) parallel execution of instructions [12]. For instance, the Intel MMX [22] allows variable precision integer arithmetic to be implemented in SIMD parallelism where either two 32-bit, four 16-bit or eight 8-bit integer values are operated on in parallel. Such processor-level support for variable-precision arithmetic has enabled efficient implementation for variable-precision rendering.

4. Geometry Bandwidth Bottleneck: Increase in the geometric complexity of the graphics datasets has far outpaced the increase in the display complexity. This has resulted in a bottleneck in transferring 3D vertex data from the geometry processor to the graphics processor. More recently, graphics architectures such as the NVIDIA's GeForce 2^{TM} and S3's Savage 2000^{TM} , have emerged that perform transformation, lighting, and rasterization on the same chip. This has further shifted the bottleneck in the graphics pipeline from processing to bandwidth to and from the graphics chip. If the variable-precision rendering techniques discussed in this paper are adopted in a graphics API and/or implemented on the chip itself (in a manner similar to the MMX technology), this could significantly reduce the bus traffic to the graphics chip and accelerate the trasformation and lighting stages on the graphics chip beyond the results reported in this paper.

In this paper we lay down the mathematical groundwork for performing variable-precision geometry transformations and lighting for 3D graphics. In particular, we explore the relationship between the distance of a given sample from the viewpoint, its location in the view-frustum, to the required accuracy with which it needs to be transformed and lighted to yield a given screen-space error bound. The main contributions of this paper are:

1. We show how variable-precision transformations and lighting (at arbitrary precisions, not just 32 and 16 bits) can speed up general 3D transformations, parallel and perspective, and result in more efficient lighting.
2. We present a careful error analysis to relate the number of bits of input precision required for a given display accuracy.
3. We study how variable-precision operations can be used with spatial and temporal coherences.

2 Previous and Related Work

In computational geometry and solid modeling, research has been done on performing robust geometric operations. Exact rational arithmetic (i.e. in homogeneous coordinates) has been found to address several shortcomings of the conventional floating-point arithmetic [13]. However, successive geometric operations can result in an unbounded growth in the precision required to accurately compute the result. One way to limit the growth of the required precision is to intersperse rounding between arithmetic operations. Rounding-off vertex coordinates (or even line and plane coefficients [13, 28]) is reasonably well-understood now. However, such rounding is much more difficult if it must preserve some combinatorial or topological structure amongst the primitives (in/out, above/below, clockwise/counterclockwise orientation etc.). Several sophisticated approaches have been proposed that perform rounding and preserve some of these relationships by adding some extra points [8] or re-adjusting the rounded-off numbers to approximately maintain the relationships [19]. In a number of cases, such results are used only to establish topological relationships amongst primitives. This can be efficiently done by using sufficiently accurate (as opposed to exact) arithmetic [3, 9, 15].

Most of the research in graphics dealing with limiting the precision of vertex coordinates has focused on rounding-off the vertex coordinates (perhaps with attributes) independently of the topological structure defined by the vertices. Thus, with such approaches it is possible that the lower-precision models suffer from artifacts such as self-intersection and false incidences, even if the original higher-precision models did not. In practice, such artifacts have not been observed frequently enough yet, to convince most graphics practitioners to adopt the more time-consuming algorithms to preserve the topological structures. In this paper, we continue this line of thinking and quantize the vertex coordinates independently of the underlying topological constraints. Deering [7] has demonstrated that quantizing the normals down to 12 bits (i.e. only $4K$ unique normals) and vertex coordinates to 24 bits results in only minimal degradation in the rendered image quality. Reducing the precision of the vertex coordinates is implicit in the work of Rossignac and Borrel [27] and more recently, Luebke and Erikson [18]. The focus there is on reducing the geometric complexity of the high detail models. Consequently, even though the resulting vertex coordinates are effectively quantized on a grid and octree respectively, the reduced precision has not been taken advantage of during transformation and rendering.

Within the area of compression of 3D models, a lot of attention has been given to reducing the number of bits to represent vertex coordinates. Most approaches have used multi-stage quantization with Huffman encoding of delta-differences between successive vertices [2, 4, 7, 17, 29, 30]. Recently, progressive compression and transmission has been actively exploited [1, 6, 20, 21, 29]. Using the techniques of geometry prediction and progressive mesh encoding [1], combined with batch processing [6, 20] and entropy encoding [21], compression ratios for progressive compression have started approaching those for single resolution compression. King and Rossignac [16] have further balanced the reduction of the number of vertices and the reduction of bits per vertex coordinate using a shape complexity measure. For a nice survey of 3D geometry compression the interested reader may refer [2, 26]. Thus far, the reduced number of bits for representing vertex coordinates have not been used for speeding-up the rendering.

Researchers at Intel [22] have shown how to use the MMX^{TM} instruction-set to perform vector-matrix multiplies in short (16-bit) precision to speed-up the parallel projection. Specifically, they show how to perform four 16-bit operations in parallel.

3 Precision and Complexity

Let us first note the difference between multi-resolution and variable-precision rendering for 3D graphics models. Multi-resolution hierarchies have traditionally involved modeling each object at multiple levels of detail, where the detail is usually measured in the number of geometric primitives required for representation. Thus, a high-detail triangle-mesh object will require a higher number of vertices, edges, and triangles for representation. This complexity is largely independent of the precision at which each vertex is being represented. As can be seen in Figure 2(b), a multiresolution technique can be used to identify how *many* primitives are necessary for a faithful representation of a given object with a given set of viewing and lighting parameters. A variable-precision technique provides bounds on the bits of *accuracy* per primitive that are required for high-fidelity rendering. This can be seen in Figure 2(c) where the points selected to represent the circle all fall on the quantization grid. Thus, the two techniques are orthogonal to each other and depending on the application requirements for accuracy and speed can be used in a complementary manner.

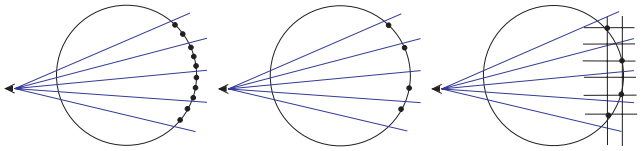


Figure 2: Varying Complexity versus Varying Precision

In the following analysis of different kinds of errors in geometric transformations, we shall assume that a minimum-sized cube has been constructed to cover the whole object using an algorithm similar to [10] and each axis has been normalized to the range of $[-1.0, 1.0]$. Thus the operands a and b are n-bit fixed-point representations of floating-point quantities within $[-1.0, 1.0]$. Additionally, we assume that we computed the n-bit fixed-point representation from such normalized floating-point representation by multiplying by 2^{n-1} and rounding to the nearest integer. Also, we would like to point out that this paper performs a worst-case analysis to guide the selection of appropriate precision. A good reference for sources and propagation of numerical errors is [24].

3.1 Representation Error

Often input data has uncertainty. A recent standards report from NIST outlines several types of uncertainty [31]. These include statistical (e.g., confidence intervals with mean and variance) and error (differences among estimates of the data from multiple sources and/or multiple time instants) uncertainties. Such uncertainties often limit the data acquisition precision. Other sources of error in the input data include approximations in the abstractions from which data is derived, numerical errors in computing the data using limited precision arithmetic, as well as instabilities in the mathematical models (as in ill-conditioned systems). Such errors often limit the number of bits of precision in the input dataset. For a n-bit fixed-point representation derived by rounding from a normalized floating-point representation, the representation error is at most half bit: $\varepsilon^{rep} \leq \frac{1}{2}$

3.2 Addition Error

For adding two n-bit integers, the error arises from the propagated error from the representation. $\varepsilon = \varepsilon^{gen} + \varepsilon^{prop} = \varepsilon^{prop} \leq \frac{1}{2} + \frac{1}{2} = 1$. So we will lose at most one bit of accuracy due to each addition.

3.3 Multiplication Error

We shall use $2n$ bits to store the intermediate result of multiplication of two n-bit integers. Since each normalized floating-point operand was magnified by a factor of 2^{n-1} during conversion to fixed-point before multiplication, we need to take out that extra 2^{n-1} factor by right shifting the intermediate result $n-1$ bits. The n-bit final result thus obtained has the largest error when both multiplier and multiplicand are close to 2^{n-1} and the absolute representation error is $\frac{1}{2}$: $\varepsilon \leq \frac{\frac{1}{2}\times 2^{n-1} + \frac{1}{2}\times 2^{n-1}}{2^{n-1}} = 1$. Thus, we lose one bit of accuracy due to each multiplication.

3.4 Division Error

Per-vertex division happens during the transformation from homogeneous coordinate to 3D image-space coordinates. The propagated error due to the division is:

$$\varepsilon^{prop} = \varepsilon_{\frac{a}{b}}{}^{prop} = \left|\frac{\partial(\frac{a}{b})}{\partial a}\right| \varepsilon_a + \left|\frac{\partial(\frac{a}{b})}{\partial b}\right| \varepsilon_b = \frac{\varepsilon_a}{b} + \frac{a}{b^2}\varepsilon_b$$

Here, ε_a and ε_b are the representation errors in a and b and are each at most $\frac{1}{2}$. For vertex within the view volume, we have $a \leq b$. Also, the generated error due to truncation is 1. Thus:

$$\varepsilon = \varepsilon^{gen} + \varepsilon^{prop} = 1 + \frac{\frac{1}{2}}{b} + \frac{a \times \frac{1}{2}}{(b)^2} \leq 1 + \frac{1}{b}$$

Since in viewing transformations, the divisor b is the distance of a scene vertex to eye in normalized view-volume representation (where the distance of the farthest point is 1.0),

$$\varepsilon \leq 1 + \frac{\text{distance of far plane in view-volume from eye}}{\text{distance of scene vertex from eye}}$$

So the loss of number of bits accuracy is

$$\left\lceil \log_2(1 + \frac{\text{distance of far plane from eye}}{\text{distance of scene vertex to eye}}) \right\rceil$$

3.5 Putting it all together

For a 1024×1024 window, with pixel level accuracy, we need 10 bits in each x and y to represent the position of a vertex on the screen. Transformation of a vertex in homogeneous coordinates with a 4×4 matrix requires four multiplications and three additions for each coordinate. The height of this operation tree is three (leaves at level 3 have four multiplies, level 2 has two additions, and the root at level 1 has the final addition). Thus, we will lose 3 bits of accuracy in this matrix-vector multiplication. To get n bits of accuracy after transformation and homogeneous division, we need m bits to represent the input data:

$$m = n + 3 + \left\lceil \log_2(1 + \frac{\text{distance of far plane from eye}}{\text{distance of scene vertex to eye}}) \right\rceil$$

Thus, if the display window is 1024×1024, $n = 10$ for pixel level accuracy; and if the distance of the point being rendered is half way across the view-volume, we shall need 15 bits to represent the vertex data: $m = 10 + 3 + \lceil \log_2(1+2) \rceil = 15$. This can be used to compute the requisite number of bits of precision required for each vertex based on its distance from the eye and forms the basis of view-dependent precision-based rendering.

For applications which require sub-pixel accuracy, we can increase the window resolution in the above formula. For example, if the application needs four bits of sub-pixel accuracy along each dimension, then we add four more bits to the requirements, which in the above example will result in a requirement of 19 bits of accuracy per input vertex coordinate for a 1024×1024 window.

4 View-dependent Transformation

The formula from the last section gives the upper bound on the number of bits needed to transform the vertices in order to get n bits of accuracy. In reality, if the object projects to the screen in an area that is small compared to the screen size, we may need less than n bits to get window-resolution-level accuracy. For view-dependent transformation, we have to find out the number of bits needed for vertices at different locations.

4.1 Octree-based Bounding Volume Hierarchy

To take advantage of the view-dependent information, we need an efficient way to estimate the projected size of different parts of an object. An octree bounding volume hierarchy is easy to build and very efficient to get the bounding volume of the projected vertices.

The idea is to find the minimum and maximum number of bits required for each bounding box using equations in the following subsections. If the two numbers are equal, then all vertices within this box will need the same number of bits during the transformation. Otherwise, vertices in this bounding box need different number of bits, and we should recurse to the lower levels of the octree hierarchy. In our implementation, we have used the normalized object coordinates, i.e., all x, y, and z coordinates lie within $[-1.0, +1.0]$.

4.2 Projected Size of the Dataset

For each view point, we calculate the projection of the eight corner points of the root level bounding box. From these projected points, we can find out the size of the object on the screen. The corner points are transformed into parallel-projection canonical viewing volume. The whole viewport will map into $[-1.0, +1.0]$ in both x and y direction of this viewing volume, so the relative size of the projected object to screen is just half the range of these projected corner points. During the calculation, we store the transformed minimum and maximum W value (i.e., minimum and maximum depth, W_{min} and W_{max}) of these eight points for later usage. The distance from the nearest visible scene vertex to view point is just the bigger one of W_{min} and near clipping plane.

4.3 Nearest Visible Vertex Accuracy

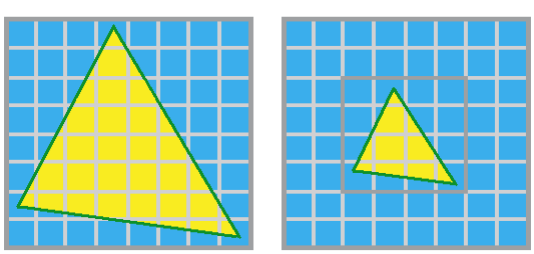
Given the width and height of the screen in number of pixels, the number of bits for pixel-level accuracy is (if sub-pixel accuracy is desired, just add those in):

$$n = screen_bits = \max(\lceil \log_2 width \rceil, \lceil \log_2 height \rceil)$$

From last subsection, we know the ratio of the projected object size to the screen size is the projected range divided by 2, so the bits needed to represent the object will be $\left\lfloor \left| \log_2 (\frac{\text{projected range}}{2}) \right| \right\rfloor$ bits less than $screen_bits$.

Taking into account the computation error due to the multiplication, addition, and division as mentioned in the last section, the number of bits needed for the nearest visible scene vertex is:

$$near_bits = n + 3 - \left\lfloor \left| \log_2 (\frac{\text{projected range}}{2}) \right| \right\rfloor + \left\lceil \log_2 (1 + \frac{\text{distance of far plane from eye}}{\text{distance of nearest vertex to eye}}) \right\rceil$$



(a) Need 3 bits in x and y (b) Need 2 bits (and 1 bit of offset)

Figure 3: Object of Smaller Projected Size needs Less Precision

As shown in Figure 3, the smaller object in (b) only occupies less than half of the screen in each dimension, so it will need one bit less than the bigger object in (a). The extra screen offset will be added in the final viewport transformation step.

4.4 Accuracy to Represent Each Vertex

Due to the perspective foreshortening, an object appears smaller as its distance to the viewpoint increases. As an example, an object at twice the distance will have half the size on the screen, and thus needs one less bit to represent.

Generally, given the $near_bits$ as defined before, we try to find the number of bits for any other vertex. After the transformed W value (i.e., depth) is known, we calculate the vertex_bits as:

$$vertex_bits = near_bits - \left\lfloor \log_2 (\frac{\text{transformed } W \text{ of this vertex}}{\text{distance of nearest vertex to eye}}) \right\rfloor$$

It will be expensive if we need to do this calculation for each vertex. Fortunately, with the bounding box hierarchy, very few calculations need to be done.

Starting from the top of the hierarchy, we calculate the minimum and maximum transformed W value for that node. First we calculate the transformed W value of the center of the node (denoted as W'_{center}), then we check the eight corner points of the node to figure out the minimum and the maximum. As we already have the W_{min} and W_{max} of the corner points at the root level, for the subtree at level k:

$$W'_{\min} = W'_{center} + \frac{W_{\min}}{2^k} \quad \text{and} \quad W'_{\max} = W'_{center} + \frac{W_{\max}}{2^k}$$

where the denominator is due to the fact that the node size is reduced by a factor of two when we go down one level in the octree.

Using the above two equations, we can find out the minimum and the maximum number of bits needed for vertices within the box:

$$V_{min} = near_bits - \left\lfloor \log_2 (\frac{W'_{\max}}{\text{distance of nearest vertex to eye}}) \right\rfloor$$

$$V_{max} = near_bits - \left\lfloor \log_2 (\frac{W'_{\min}}{\text{distance of nearest vertex to eye}}) \right\rfloor$$

If these two numbers are equal, then we know that all the vertices within the box will need these number of bits to represent. Otherwise, vertices in the box require different numbers of bits and we need to recurse down one more level of the octree.

5 Spatio-Temporal Coherence

In the last two sections we have seen the relationship between the input bits of accuracy and the bits of accuracy required for the output. For the same number of bits of accuracy for the output, we can further reduce the bits of accuracy required in the input by taking advantage of spatial and temporal coherence. This can result in further savings in processing time as well as in the bandwidth to the graphics processor.

5.1 Spatial Coherence

The basic idea that we use to take advantage of the spatial coherence is that the difference in spatially close vertices can usually be represented in far fewer bits than those required to represent each vertex coordinate in its entirety. This idea has been used with great success in the research on 3D compression of geometry as discussed in

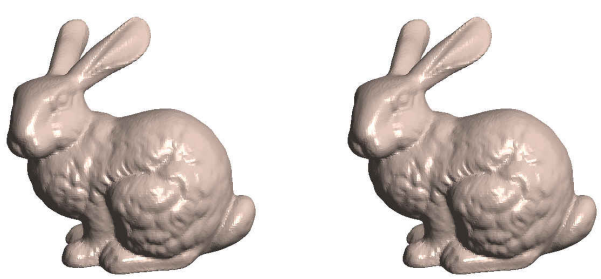
Section 2. If a vertex coordinate x' can be represented by a delta-difference with respect to another coordinate x as $x' = x + \Delta x$ then one can decompose the transformation for coordinate x' as: $Mx' = M(x + \Delta x) = Mx + M\Delta x$

Since the number of bits of accuracy required to transform Δx is much smaller, one can perform several of them in parallel. To exploit this idea, we can partition the dataset by any spatial subdivision scheme such as an octree over the vertices of the model. In our implementation we have used an octree that subdivides by the volume centroid at each level. In this scheme, since each level reduces the range by half, the vertices in each lower level require one bit less than their parents. The accuracy of the transformation of a vertex coordinate with a matrix is represented by the lower of the two accuracies. Thus if the vertex coordinates can be quantized in less bits, the transformation matrix values can also be quantized with less number of bits.

In this approach we independently transform the delta difference in the vertex coordinate position between the current level of the octree and its parent. Then we can get the final transformed results for each vertex by a top-down tree traversal as the following: ($LIMIT$ is the lowest level of tree below which the difference between the transformed parent and children is negligible)

Top-Down-Tree-Traversal(x)
if $x \neq NULL$
  if $x.level \leq LIMIT$
    $x.value = x.parent.value + x.transform$
    for i from 1 to 8
      Top-Down-Tree-Traversal($x.child(i)$)
  else
    $x.value = x.parent.value$
    for i from 1 to 8
      Top-Down-Tree-Traversal($x.child(i)$)

As an example, if we could operate on byte and short precision operands and we required 16 bits of accuracy, then we could transform the top eight levels of the octree in short-precision and the lower levels could be transformed in byte-precision (or even lesser, if available). By using such hierarchical schemes, one can get a better precision efficiency without losing accuracy. Figure 4 compares the results of bunny model using floating point and variable-precision transformation.



(a) Floating Point Transform (32 bits/vertex coordinate)

(b) Variable Precision Transform (Average 7.9 bits/vertex coordinate)

Figure 4: Variable-Precision Transformation of Bunny Model (Stanford model, 69K triangles; lighting for both images has been calculated in floating point)

5.2 Temporal Coherence

Similar to the idea of spatial coherence, we can take advantage of temporal coherence by noting that the difference in the transformed vertex positions does not differ significantly from one frame to the next. Thus if we calculate the difference in the transformation matrix from one frame to the next and use the difference matrix ΔM to transform a vertex, we can then add it to the previously transformed vertex position in less number of bits: $M'x = (M + \Delta M)x = Mx + \Delta Mx$. Extending this idea further, we note that one can combine the spatial and temporal coherences: $M'x' = (M + \Delta M)(x + \Delta x) = Mx + \Delta Mx + M\Delta x + \Delta M\Delta x$ As we show in Tables 2 and 3 for the Auxiliary Machine Room dataset, the average number of bits that are operated upon for each vertex as well as the equivalent number of operations can both be greatly reduced by taking advantage of both spatial and temporal coherences.

6 Variable-Precision Lighting

Color is usually represented by 8-bits of precision in red, green, and blue components and sometimes even less (for instance in lower-range graphics cards and Personal Digital Assistants). Also, if depth cueing is turned on and the far objects are displayed at lower intensities, their color can be represented using fewer bits.

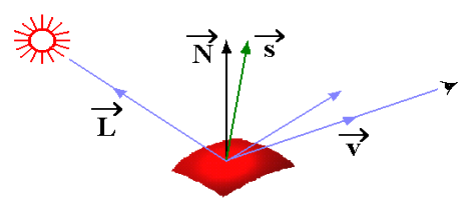


Figure 5: Lighting Calculation

Before we go to the detailed treatment of the variable-precision lighting, let us review the formula for the lighting calculation we have used. Although there are good psychophysically-based light reflection models [23], we decided to implement the OpenGL illumination model. As in OpenGL, we assumed diffuse and Phong illumination with Gouraud shading without per-pixel normal evaluation:

$$\begin{aligned} Color \;=\; & \text{emission}_{\text{material}} + \text{ambient}_{\text{light_model}} * \text{ambient}_{\text{material}} + \\ & + \sum_{i=0}^{m-1} \left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i * (\text{spotlight effect})_i * \\ & (C_{\text{ambient}} + C_{\text{diffuse}} + C_{\text{specular}})_i \end{aligned}$$

where m is the number of light sources, $(\frac{1}{k_c + k_l d + k_q d^2})$ gives the attenuation factor in which d is the distance between the vertex and the local light source. $C_{\text{ambient}} = \text{ambient}_{\text{light}} * \text{ambient}_{\text{material}}$, $C_{\text{diffuse}} = (\max\{\vec{L} \cdot \vec{N}, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}}$ and $C_{\text{specular}} = (\max\{\vec{s} \cdot \vec{N}, 0\})^{\text{shin}} * \text{specular}_{\text{light}} * \text{specular}_{\text{material}}$. $\vec{L}$ is the unit vector that points from the vertex to the light position, $\vec{N}$ is the unit normal vector at the vertex, $\vec{s}$ is the normalized half way vector between the directions of the light source and the viewer, and *shin* is the shininess, i.e., the specular exponent. Our goal here is to find the necessary number of bits to represent the input illumination data in order to get the required accuracy in output color.

6.1 Sources of Error in Local Illumination

There are several additional sources of error in local illumination computation beyond the sources of error we have already discussed in the transformation stage (representation error, addition error, multiplication error, and division error). In lighting computations we have to deal with addition and multiplication errors for operands with different bits of accuracy, the square root operation error which results from vector normalization, and the error induced by exponentiation in specular illumination. Also, the special case of dot product of two unit vectors is worthy of separate analysis.

To reduce the error propagation, we can multiply the light coefficient with the object material property coefficient in floating-point form before converting to the n bits fixed-point representation. For example, instead of converting $ambient_{\text{light_model}}$ and $ambient_{\text{material}}$ to n bits of integer, we multiply them in floating-point representation and then convert the result to n bits of integer. This way, we can save one bit of accuracy which will be lost due to the multiplication of two n-bit integers. We next consider the other sources of error in the following subsections.

6.1.1 Error for Operands with Different Accuracy

Let us consider two operands with different bits of accuracy, say n and n' where $n' < n$. This means that if the maximum possible value is 1, then the representation errors are $2^{-(n+1)}$ and $2^{-(n'+1)}$, respectively. For addition, the error ε can be computed as:

$$\varepsilon \leq 2^{-(n+1)} + 2^{-(n'+1)} = 2^{-(n'+1)}(1 + 2^{-(n-n')})$$

As an example, if $n - n' = 2$, then: $\varepsilon \leq 2^{-(n'+1)}(1 + \frac{1}{4})$. The error will stay at $(n'+1)^{th}$ bit, and the result will get n' bits of accuracy, i.e., the same accuracy as the less accurate operand.

Similarly, for multiplication of operands with n and n' $(n' < n)$ accuracy, the maximum possible error happens when the operands are close to the maximum possible value which we treat as 1, as we discussed in last section:

$$\varepsilon \leq 2^{-(n+1)} * 1 + 2^{-(n'+1)} * 1 = 2^{-(n'+1)}(1 + 2^{-(n-n')})$$

Again, the result has the same accuracy as the less accurate operand.

6.1.2 Error in the Dot Product of Unit Vectors

Let us consider two unit vectors, say $\overrightarrow{\alpha}$ and $\overrightarrow{\beta}$, with n bits of accuracy in each of their three components:

$$\overrightarrow{\alpha} = (\alpha_1, \alpha_2, \alpha_3) \text{ and } \overrightarrow{\beta} = (\beta_1, \beta_2, \beta_3)$$

Since the error in the three components ε_{α_i} and ε_{β_i} $(i = 1, 2, 3)$ is in the $(n+1)^{th}$ bit, i.e., $2^{-(n+1)}$, their dot product error is:

$$\varepsilon_{(\overrightarrow{\alpha} \cdot \overrightarrow{\beta})} = \sum_{i=1}^{3}(\beta_i \varepsilon_{\alpha_i} + \alpha_i \varepsilon_{\beta_i}) \leq 2^{-(n+1)}(\sum_{i=1}^{3} \alpha_i + \sum_{i=1}^{3} \beta_i)$$

For unit vector $\overrightarrow{\alpha}$, we have: $\alpha_1^2 + \alpha_2^2 + \alpha_3^2 = 1$.
From the inequality: $a^2 + b^2 \geq 2ab$, we get:

$$(\alpha_1 + \alpha_2 + \alpha_3)^2 \leq 3(\alpha_1^2 + \alpha_2^2 + \alpha_3^2) = 3$$

So we have $(\alpha_1 + \alpha_2 + \alpha_3) \leq \sqrt{3}$. Similarly, we have $(\beta_1 + \beta_2 + \beta_3) \leq \sqrt{3}$. Then:

$$\varepsilon_{(\overrightarrow{\alpha} \cdot \overrightarrow{\beta})} \leq 2^{-(n+1)}(\sum_{i=1}^{3} \alpha_i + \sum_{i=1}^{3} \beta_i) \leq \sqrt{3} * 2^{-n}$$

That means, we will lose one to two bits of accuracy for dot product of two unit vectors.

6.1.3 Error in the Square Root Operation

For lighting calculations we need to normalize the vectors to unit length before we compute the dot product. Normalization involves division by the magnitude of the vector which requires a square root operation. In order to perform all the operations in the fixed-point arithmetic, we use a a table lookup to get the square root of an unsigned integer.

For an unsigned integer X with $2n$ bits of accuracy we take the most significant n bits (say X') as the lookup index into the square-root table to find the square root.

$$X' = (X >> n) << n$$

The maximum possible error of X' relative to X is 2^n (because the information in the lower n bits is lost). We can reduce this error by half though. If the value of the nth bit of X is one, we can add one to $X >> n$, so that it becomes a kind of rounding error instead of trunction error.

Next, we use the square-root table to find the square-root of X'. Let this be a' in integer representation: $X' = a'^2$. Suppose the square root of X in integer representation is $a : X = a^2$. Let $a' = a + \varepsilon_a$ (ε_a is the error), then:

$$a'^2 = (a + \varepsilon_a)^2 = a^2 + 2a\varepsilon_a + (\varepsilon_a)^2 = X'$$

That is, $2a\varepsilon_a + (\varepsilon_a)^2 = X' - X \leq 2^{n-1}$.

If $X > 2^{2n-2}$, then $a > 2^{n-1}$, thus:

$$2a\varepsilon_a < 2a\varepsilon_a + (\varepsilon_a)^2 \leq 2^{n-1} \text{ and } \varepsilon_a < \frac{2^{n-1}}{2a} < \frac{2^{n-1}}{2*2^{n-1}} < \frac{1}{2}$$

Which means that if we use the most significant n bits of the unsigned integer as index into the square root table then as long as the integer is bigger than 2^{2n-2}, the result has n bits of accuracy.

6.1.4 Error in the Evaluation of Specular Exponentiation

To calculate the specular component of illumination, we have to compute the exponent of the dot product of half-way vector with the normal vector. Due to the fact that the dot product a of two unit vectors is always smaller or equal to 1 and that we are only dealing with positive values of the dot product, we use 2^m to represent the largest value 1. The maximum possible representation error will be $\frac{1}{2}$, i.e., $2^{-(m+1)}$ relative to 1.

If ε_a is the error in the value a of the dot product, then:

$$(a + \varepsilon_a)^n \cong a^n + na\varepsilon_a \text{ (if } \varepsilon_a << a)$$

The maximum absolute error happens when $a = 1$, $\varepsilon_a = 2^{-(m+1)}$, and n is the maximum value of 128: (as implemented by OpenGL)

$$na\varepsilon_a < 128 * 2^{-(m+1)} < 2^{-(m-6)}$$

So we will have $m - 6$ bits accuracy in the result, i.e., we will lose 6 bits accuracy due to this exponentiation.

6.1.5 Putting it all together

From the above analysis, we can get an equation which relates the input data accuracy with the output color accuracy. Assume the output color needs n bits accuracy per R, G, and B, which requires m bits of accuracy in the input data. We next relate n and m.

First, the normalization of each vector will lose 1 bit. As shown before, the square root will have nearly the same accuracy as the input data. To avoid the loss of accuracy due to division, instead of storing the square root, we store the reciprocal of the square root in the lookup table. This reciprocal is calculated in the floating-point

representation before converting it to the n bits fixed-point representation. Thus the only error induced in the normalization will be in the final multiplication which is a loss of one bit of accuracy.

The dot product of two unit vectors will lose one to two bits of accuracy. Since the exponentiation will lose six bits, the term $(\max\{\vec{s}\cdot\vec{N},0\})^{\text{shin}}$ will lose $1+(1\text{ to }2)+6=8$ to 9 bits of accuracy. So the above term have between $m-8$ and $m-9$ bits of accuracy. Further, the term C_{specular} will have the same accuracy because $\text{specular}_{\text{light}}*\text{specular}_{\text{material}}$ will have m bits of accuracy, which is much higher than the accuracy of $(\max\{\vec{s}\cdot\vec{N},0\})^{\text{shin}}$.

Similarly, the term C_{diffuse} will get between $m-2$ and $m-3$ bits of accuracy. And we know the term C_{ambient} will have m bits of accuracy as explained in the overview.

Overall, $(C_{\text{ambient}}+C_{\text{diffuse}}+C_{\text{specular}})$ will have the accuracy as the lowest accurate term C_{specular}, i.e., $m-8$ or $m-9$ bits of accuracy.

Since the attenuation and the spotlight terms can all be evaluated with more than $m-8$ bits of accuracy, the required color accuracy bits of the entire illumination equation can be expressed as:

$$n = m-8 \text{ or } m-9$$

For example, if $n=8$, i.e., eight bits per R, G, and B, then the required accuracy for the input data will be $n+8$ or $n+9$, i.e., we will need 16 or 17 bits to represent the input data to get the desired accuracy of 8 bits per color component.

6.2 View-dependent Variable-Precision Lighting

Similar to the case of transformation in Section 5, we can take advantage of the spatial coherence of the adjacent vertices in lighting calculations.The basic idea is that the viewing and lighting directions do not vary much for the spatially close vertices. Once we find those directions for one vertex, we can calculate the directions for the nearby vertices incrementally, i.e., calculate the difference in far fewer number of bits. The direction difference between the nearby vertices depend not only on the absolute spatial difference of the vertices, but also on their distances from the viewer and light source to the vertices. Once the viewer moves closer to the vertex and below a threshold (which we will describe below) we will switch back to the original case, i.e., treat that particular vertex independently of its adjacent vertices.

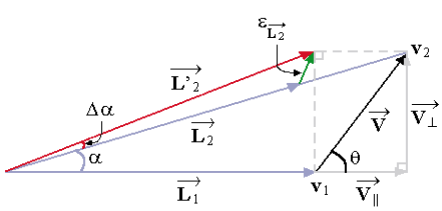


Figure 6: Incremental Lighting Calculation

In Figure 6 we show how to compute the lighting incrementally. Let $\vec{L_1}$ be the light vector for vertex v_1 for which we have already calculated the illumination. Now we would like to find out the light vector $\vec{L_2}$ for its adjacent vertex v_2. The displacement vector $\vec{V}$ between v_1 and v_2 is normalized by the distance between the v_1 and the light source, i.e., its length is equal to the real distance between v_1 and v_2 divided by the distance between the v_1 and the light source.[1] Both $\vec{L_1}$ and $\vec{L_2}$ are unit vectors.

One way to accurately compute $\vec{L_2}$ is to normalize the sum of $\vec{L_1}$ and the vector between v_1 and v_2. This approach requires roughly the same amount of computation as to compute the $\vec{L_2}$ directly from the vector between v_2 and the light source. To reduce the computation, we instead use $\vec{L'_2}$ (equal to $(\vec{L_1}+\vec{V_\perp})$) as the approximation of $\vec{L_2}$ if it satisfies our accuracy requirements. $\vec{V_\perp}$ is the component vector of $\vec{V}$ on the perpendicular direction of $\vec{L_1}$, which can be easily computed: $\vec{V_\perp}=\vec{V}-\vec{L_1}(\vec{V}\cdot\vec{L_1})$. Using this approach, the induced error $\varepsilon_{\vec{L_2}}$ is equal to $\vec{L'_2}-\vec{L_2}$.

If the length of $\vec{V}$, $\left\|\vec{V}\right\|$, is much smaller than 1 (the length of $\vec{L_1}$ and $\vec{L_2}$), then we have:

$$\begin{aligned}\left\|\vec{L'_2}\right\| &= \sqrt{\left\|\vec{L_1}\right\|^2+\left\|\vec{V_\perp}\right\|^2}=\sqrt{1+\left\|\vec{V_\perp}\right\|^2}\\ &\approx 1+\frac{\left\|\vec{V_\perp}\right\|^2}{2}\le 1+\frac{\left\|\vec{V}\right\|^2}{2}\end{aligned}$$

Let the angle between $\vec{L_1}$ and $\vec{L_2}$ be α, between $\vec{L_2}$ and $\vec{L'_2}$ be $\Delta\alpha$, and $\vec{V_\parallel}$ be the component vector of $\vec{V}$ along the direction of $\vec{L_1}$:

$$\alpha=\arctan\left(\frac{\left\|\vec{V_\perp}\right\|}{\left\|\vec{L_1}\right\|+\left\|\vec{V_\parallel}\right\|}\right)=\arctan\left(\frac{\left\|\vec{V_\perp}\right\|}{1+\left\|\vec{V_\parallel}\right\|}\right)$$

$$(\alpha+\Delta\alpha)=\arctan\left(\frac{\left\|\vec{V_\perp}\right\|}{\left\|\vec{L_1}\right\|}\right)=\arctan\left(\left\|\vec{V_\perp}\right\|\right)$$

$$\begin{aligned}\text{So}\qquad \Delta\alpha &= \arctan\left(\left\|\vec{V_\perp}\right\|\right)-\arctan\left(\frac{\left\|\vec{V_\perp}\right\|}{1+\left\|\vec{V_\parallel}\right\|}\right)\\ &\approx \left\|\vec{V_\perp}\right\|\left(1-\frac{1}{1+\left\|\vec{V_\parallel}\right\|}\right)=\frac{\left\|\vec{V_\perp}\right\|\left\|\vec{V_\parallel}\right\|}{1+\left\|\vec{V_\parallel}\right\|}\\ &< \left\|\vec{V_\perp}\right\|\left\|\vec{V_\parallel}\right\|\le\frac{1}{2}\left\|\vec{V}\right\|^2\end{aligned}$$

The last inequality is because $\left\|\vec{V_\perp}\right\|=\left\|\vec{V}\right\|\sin\theta$ and $\left\|\vec{V_\parallel}\right\|=\left\|\vec{V}\right\|\cos\theta$, and $\sin\theta\cos\theta=\frac{1}{2}\sin(2\theta)\le\frac{1}{2}$. Thus if the distance between v_1 and v_2 is much less than the distance between v_1 and the light source, then $\left\|\vec{L'_2}\right\|\approx 1=\left\|\vec{L_2}\right\|$ and $\Delta\alpha<<1$, therefore

$$\left\|\varepsilon_{\vec{L_2}}\right\|\approx 2\left\|\vec{L_2}\right\|\tan\left(\frac{\Delta\alpha}{2}\right)\approx\Delta\alpha\le\frac{1}{2}\left\|\vec{V}\right\|^2$$

This means, the error of using $\vec{L'_2}$ as an approximation of $\vec{L_2}$ is less than $\frac{1}{2}\left\|\vec{V}\right\|^2$. If we want 15 bits of accuracy in $\vec{L_2}$, we only need $\left\|\vec{V}\right\|\le 2^{-7}$, i.e., the distance between v_1 to the light source should be $2^7=128$ times larger than the distance between v_1 and its addjacent vertex v_2. This way we only use the local spatial differences in calculating the new direction and avoid an expensive vector normalization operation.

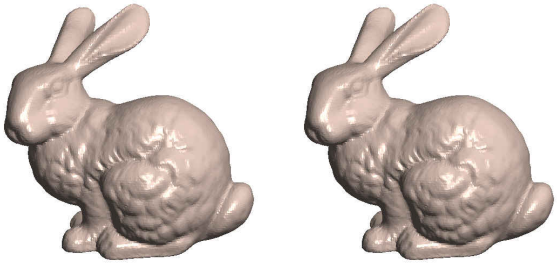
7 Some Implementation Details

In addition to what we have already described in the previous sections, there are some other implementation details which are worth mentioning.

[1] Note that this assumption is not shown in Figure 6, where $\vec{V}$ is shown to have its length as the distance between v_1 and v_2.

Model		Bunny	DHFR	Dragon	Venus	AMR	Buddha
Size (triangles)		69K	145K	202K	268K	376K	1087K
Floating Point	Transform	61	130	185	230	330	968
	Lighting	469	1042	1374	1830	2503	7481
	Other	56	108	167	218	298	902
	Total	586	1280	1726	2278	3131	9351
Variable Precision	Transform	17	33	46	59	83	235
	Lighting	79	155	212	280	337	882
	Other	42	87	127	160	212	616
	Total	138	275	385	499	632	1733
Speedup		4.25	4.65	4.48	4.57	4.96	5.40
e_{rms} (object space)		1.3e-4	1.3e-4	1.2e-4	1.2e-4	1.1e-4	1.2e-4
Max error (obj. space)		3.0e-4	3.1e-4	3.0e-4	2.9e-4	2.6e-4	3.1e-4
e_{rms} (image space)		8.5e-3	8.8e-3	8.7e-3	6.0e-3	8.4e-3	7.0e-3

Table 1: Results from Rendering at Varying Precisions



(a) Floating Point Lighting (b) Variable-Precision Lighting (Speedup: 2.99)

Figure 7: Variable-Precision Lighting of Bunny Model
(Transformations have been calculated in floating-point)

7.1 Batched Transformation and Lighting

Most graphics APIs (OpenGL, Direct3D, Glide) allow the user to transform and light the triangles one at a time and send the transformed and lighted triangles in floating-point screen coordinates to the rasterizer. Since these APIs do not accept screen-space triangles in the fixed-point representation, we had to convert our fixed-point results to floating-point representation before asking the graphics API to rasterize the triangles. In MMX technology, this means that we need to reset the register flag back and forth when we switch from the integer operation to floating point because these two share the same registers. The frequent resetting costs time, so the intuitive solution is to minimize the number of resets, e.g., transform and light the whole dataset first in object space, then do the viewport transform and then send to the rasterizer. There are two problems with this approach. First we lose some opportunities of pipelining which the hardware is very smart at. Second, there are lots of extra memory accesses due to the write-back, so this does not work well.

To solve this problem, we make a tradeoff. Instead of transforming and lighting the triangle one by one or all at the same time, we do them batch by batch. The resetting of the flag only happens between batches and we avoid the extra memory accesses. In practice, we find batch size of several hundred triangles works gracefully. If the graphics APIs accepted screen-space fixed-point representation triangles, we would not have to deal with this and our results would have been better than reported here since switching from fixed-point to floating-point is expensive even when we do them in batches.

7.2 Full-precision Matrix Calculation

At each view point we first calculate the transformation matrix and then apply it to all the vertices in the dataset. The initial matrix calculation is a negligible fraction of the overall computation which includes transformation of hundreds of thousands of vertices. So we compute it in full precision floating point before converting it into the fixed-point representation. This way, we save the precision of the matrix elements, and avoid the possibility of error build up when we take advantage of the temporal coherence of the frames in transformation because the matrix is computed in full precision separately for each frame.

8 Results

We have tested our approach on polygonal datasets from several application domains including laser-scanned, mechanical CAD, and procedurally generated datasets. The results of our approach are summarized in Tables 1, 2, 3 and appear in Figures 1, 4, 7–14.

We obtained the results shown in the paper and the video on a Pentium II 400MHz PC with 128MB RAM and a Voodoo3 3500 graphics card. Table 1 compares the results using variable precision with the one using traditional single-precision floating point and times are reported in milliseconds. The variable precision rendering showing here is under the requirements of guaranteed pixel-level position accuracy and eight bits per R, G, and B color. The object space root-mean-square error and maximum error are measured in transformed object space as the distance between the single precision floating point transformed vertices and variable precision transformed ones, while the image space root-mean-square error is measured in the final image space as the difference between the R, G, B color components. The formula for image space root-mean-square error is the following:

$$e_{rms} = \left[\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left[\widehat{f}(x,y) - f(x,y) \right]^2 \right]^{1/2}$$

Where $f(x,y)$ represents the original image, $\widehat{f}(x,y)$ denotes an estimate of the image, and $M \times N$ is the image size.

From Table 1, we can see that under the pixel-level accuracy, the maximum transformed distance between the two methods is less than 0.00033 for all the six datasets tested . We know the normalized transformed object space is in the range [-1.0, +1.0], so the difference is less than six-thousandth of the total range. This shows robustness of our method. Further, instead of getting pixel-level accuracy, our method actually gave us 2 to 3 sub-pixel bits of accuracy. This is because our error analysis gives the upper bounds of the error; the real error is usually much less. To roughly compare how variable precision rendering stacks up against multiresolution rendering, we compared the object space Hausdorff error in a 16K triangle model of the Bunny using Metro [5] against a 69K triangle model of the Bunny using 7.9 bits/vertex coordinate. Although both give a factor of 4 speedup, the variable precision method has an order of magnitude smaller object space Hausdorff error (0.012% of the bounding box diagonal) compared with 16K triangle full precision model (0.12% of the bounding box diagonal).

We can see more than a factor of four speedup in all the datasets tested. One aspect of our algorithm is that it scales well. The speedup factor goes up with the increase of scene complexity (which means more data will be rendered in less precision) and the

Output bits	Conventional		Spatial		Spatio-Temporal	
	Add	Mult	Add	Mult	Add	Mult
32 bits	32	32	42.21	31.55	54.08	29.23
16 bits	16	16	12.95	7.77	13.77	4.02
8 bits	8	8	1.35	0.66	0.77	0.08
4 bits	4	4	0.03	0.02	0.01	0.001

Table 2: Average Number of Bits per Vertex Coordinate Operated upon for Appropriate Output Precision

Output bits	Conventional		Spatial		Spatio-Temporal	
	Add	Mult	Add	Mult	Add	Mult
32 bits	6	8	7.92	7.89	10.14	7.31
16 bits	3	4	2.43	1.94	2.58	1.01
8 bits	1.5	2	0.25	0.17	0.14	0.02
4 bits	0.75	1	0.01	0.004	0.002	0.0002

Table 3: Average number of equivalent 32-bit operations per vertex coordinate for appropriate output precision

number of light sources. See Figure 8 and Table 1.

Figure 9 shows the histogram of percentage of vertices transformed in different number of bits for AMR dataset, which has a very low average 4.18 bits/vertex coordinate for variable-precision transformation, instead of 32 bits/vertex coordinate as in the single-precision floating point case. Figures 1, 10–14 show the images rendered by variable precision rendering and compare them with the single-precision floating point rendering. Even from the closeup there is hardly any visual distinguishable difference.

Table 2 shows the average number of bits that have to be manipulated per vertex during the transformations while exploiting spatial and temporal coherences. Since the vertices that are at the lower levels of the octree require less number of bits for transformation, the overall average number of bits turns out to be much less. The leftmost column indicates the number of bits that are required in the output display.

Table 3 shows the average number of equivalent 32-bit operations per vertex during the transformations while exploiting spatial and temporal coherences. Central to this idea is that one 32-bit operation is equivalent to two 16-bit, four 8-bit, and eight 4-bit operations. Even though SIMD parallelism at the level of 4-bits is not yet available in the current generation of processors, the table shows the effectiveness of our scheme if such parallelisms were to become available in future. As in table 2, the leftmost column indicates the number of bits that are required in the output display.

9 Conclusions

We have presented a novel approach to take advantage of SIMD parallelism in modern processors to speedup the transformation and lighting stages of the graphics pipeline. Our approach can successfully trade-off the precision for speed without significantly affecting the visual quality of the rendered images. In addition, Our approach is complementary to the conventional multiresolution approaches that rely on speeding up the rendering by reducing the number of graphics primitives for display.

In this paper we have mostly dealt with reducing the precision of all the three coordinates x, y, and z of the displayed primitives. However, a case can be made for not reducing the precision for the eye-space z in applications that have a high depth complexity since it can interfere with the proper execution of the visible-surface determination stage of the graphics pipeline (e.g., see [25]). This suggests potential for further work in relating precision in x, y, z with accurate scan-conversion, extending the precision work in [25] and in this paper.

Here we have focused on using variable precision for transformations and lighting. In future, one can carry out a similar analysis to extend it to texture coordinate computation as well. Also interesting will be applications of this method to compute the delta-differences on-the-fly for hardware-supported graphics primitives such as triangle strips that currently only exploit connectivity coherence, but using ideas in this paper could also benefit from spatial coherence. We plan to explore these ideas in the future.

10 Acknowledgements

We would like to acknowledge the reviewers for their very detailed and constructive comments which have led to a much better presentation of our results. This work has been supported in part by the NSF grants: IIS-00-81847, ACR-98-12572, and DMI-98-00690.

References

[1] C. Bajaj, V. Pascucci, and G. Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In *Proceedings Visualization 99*, pages 307 – 316, Los Alamitos, California, 1999. IEEE, Computer Society Press.

[2] C. Bajaj, V. Pascucci, and G. Zhuang. Single resolution compression of arbitrary triangular meshes with properties. *Computational Geometry: Theory and Applications*, vol 14:167–186, 2000.

[3] R. Banerjee and J. Rossignac. Topologically exact evaluation of polyhedra defined in CSG with loose primitives. *Computer Graphics Forum*, 15, No. 4:205–217, 1996.

[4] M. Chow. Optimized geomerty compression for real-time rendering. In *IEEE Visualization '97 Proceedings*, pages 403 – 410. ACM Press, October 1997.

[5] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17, No. 2:167–174, June 1998.

[6] D. Cohen-Or, D. Levin, and O. Remez. Progressive compression of arbitrary triangular meshes. In *Proceedings Visualization 99*, pages 67–72, Los Alamitos, California, 1999. IEEE, Computer Society Press.

[7] M. F. Deering. Geometry compression. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 13–20. ACM SIGGRAPH, Addison Wesley. Los Angeles, California, August 1995.

[8] S. Fortune. Vertex rounding a three-dimensional polyhedral subdivision. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 116–125. ACM Press, June 1998.

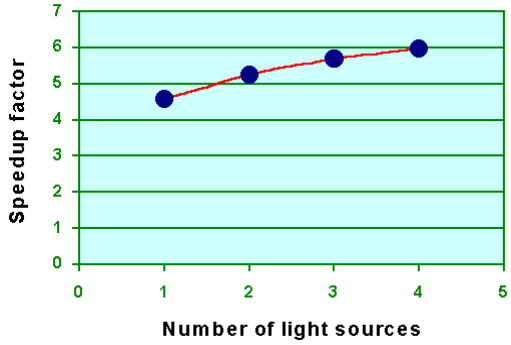


Figure 8: Speedup Factor as Function of Number of Light Sources (Venus Model)

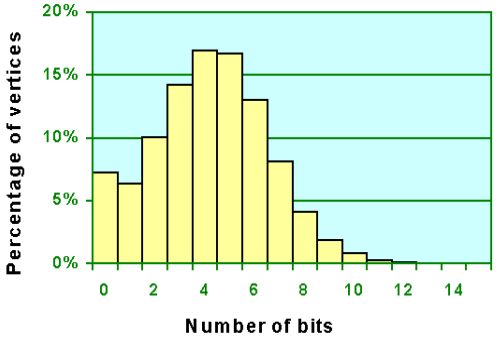
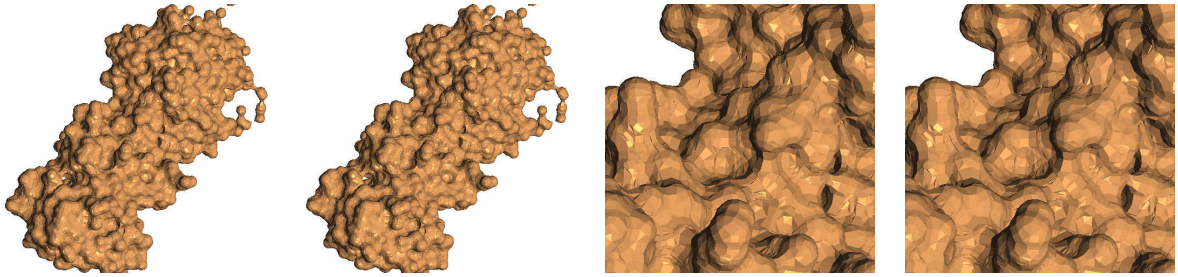
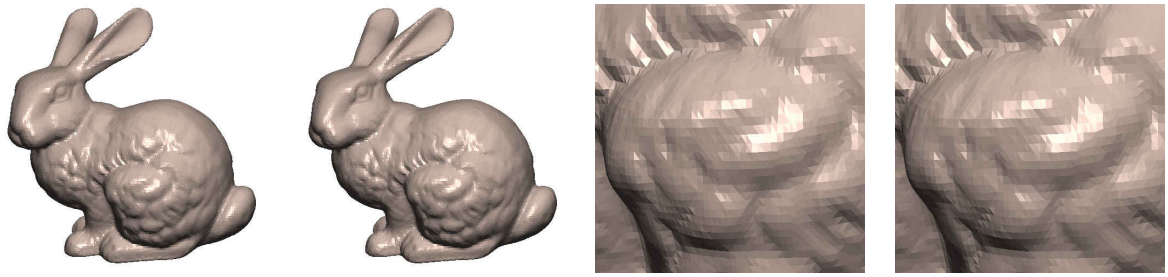


Figure 9: Histogram of Vertices Transformed in Different Number of Bits using Variable Precision (AMR Model)



(a) Floating Point (32 bits/Vertex Coordinate) (b) Variable Precision (7.9 bits/Vertex Coordinate) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 10: Dihydrofolate Reductase Molecular Surface (145K triangles) Rendered in Variable Precision



(a) Floating Point (32 bits/vertex coordinate) (b) Variable Precision (7.9 bits/vertex coordinate) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 11: Stanford Bunny (69K triangles) Rendered in Variable Precision

[9] S. Fortune and C. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.

[10] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 171–180. Addison Wesley, August 1996.

[11] P. Heckbert. Color image quantization for frame buffer display. In *Computer Graphics (SIGGRAPH 82 Procs)*, volume 16(3), pages 297–307, July 1982.

[12] J. Hirshon. A guide to CPU 3D instruction sets, August 1999. http://www.3d-design.com /newsletter /1999 /0899/horizon0899.html.

[13] C. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, San Mateo, California, 1989.

[14] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH 97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 189 – 197. ACM Press, August 1997.

[15] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *15th ACM Symp. on Computational Geometry*, 1999.

[16] D. King and J. Rossignac. Optimal bit allocation in compressed 3d models. *Journal of Computational Geometry, Theory and Applications*, 14:91–118, November 1999.

[17] J. Li and C. C. Kuo. Progressive coding of 3D graphics models. *Proceedings of the IEEE*, 86(6):1052–1063, June 1998.

[18] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97 Conference Proceedings (Los Angeles, CA)*, Annual Conference Series, pages 198 – 208. ACM Press, August 1997.

[19] V. Milenkovic and L. Nackman. Finding compact coordinate representations for polygons and polyhedra. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, pages 244–252. ACM Press, June 1990.

[20] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, January 2000.

[21] R. Pajarola and J. Rossignac. Squeeze: Fast and progressive decompression of triangle meshes. In *Proceedings Computer Graphics International CGI 2000*, pages 173–182, Los Alamitos, California, 2000. IEEE, Computer Society Press.

[22] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[23] F. Pellacini, J. A. Ferwerda, and D. P. Greenberg. Toward a psychophysically-based light reflection model for image synthesis. In *Siggraph 2000 Conference Proceedings*, Annual Conference Series, pages 55–64. ACM Press, 2000.

[24] S. M. Pizer and V. L. Wallace. *To Compute Numerically: Concepts and Strategies*. Little, Brown Computer. Systems Series, 1983.

[25] J. Rossignac. Accurate scan-conversion of triangulated surfaces. In *Advances in Computer Graphics Hardware VI*, Proc. of the 6th Eurographcis Workshop on Computer Graphics Hardware, pages 13–20. Springer Verlag. September 1991.

[26] J. Rossignac. Edgebreaker: Connnectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1), January 1999.

[27] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June 1993.

[28] K. Sugihara. On finite-precision representations of geometric objects. *Journal of Computer and System Sciences*, 39:236–247, 1989.

[29] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive forest split compression. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 123–132. ACM SIGGRAPH, 1998.

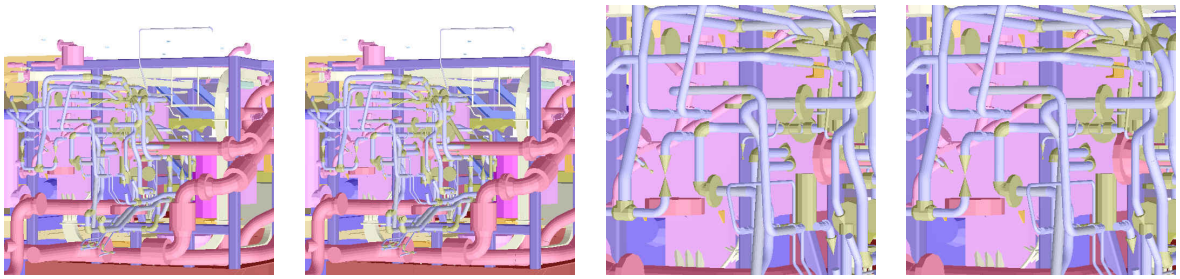
[30] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, April 1998.

[31] B. Taylor and C. E. Kuyatt. Guidelines for evaluating and expressing the uncertainity of NIST measurement results. Technical Report Technical Note 1297, National Institute of Standards and Technology, Gaithersburg, MD, January 1993.

[32] J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3, No. 2:171 – 183, June 1997.

[33] Z. Xiang. Color image quantization by minimizing the maximum intercluster distance. *ACM Transactions on Graphics*, 16(3):260–276, July 1997.

[34] H. Zhang and K. E. Hoff. Fast backface culling using normal masks (color plate S. 189). In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 103–106, New York, April27–30 1997. ACM Press.

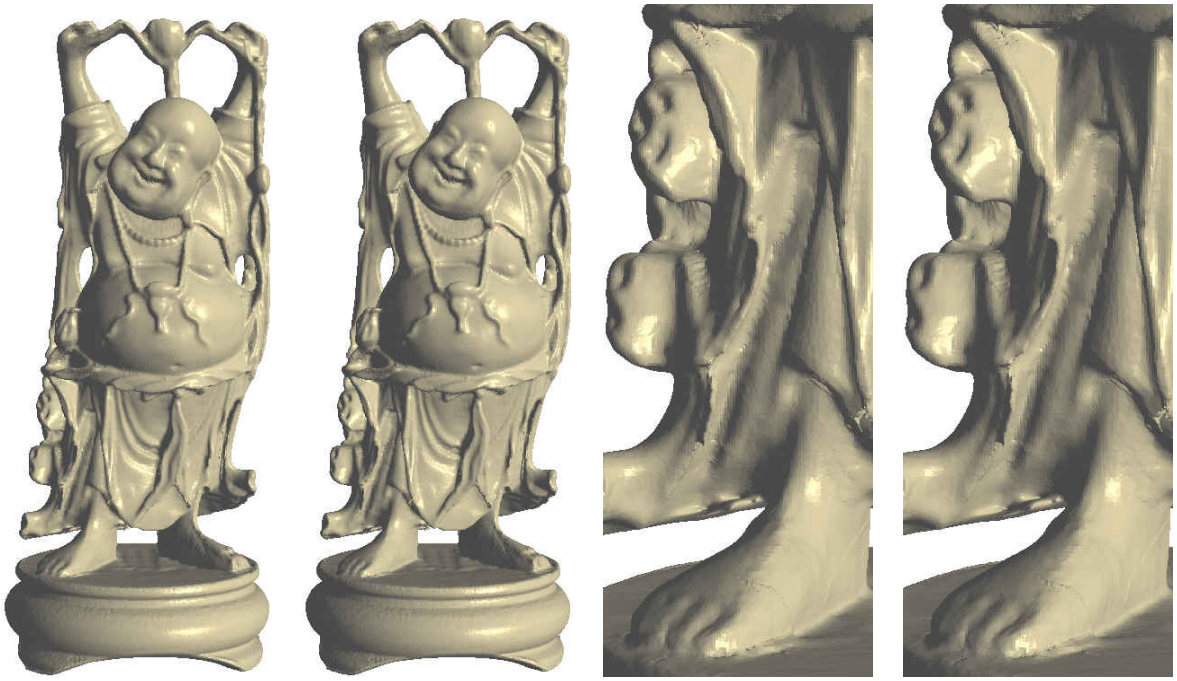


(a) Floating Point (32 bits/vertex coordinate) (b) Variable Precision (4.2 bits/vertex coordinate) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 12: Auxiliary Machine Room (376K triangles) Rendered in Variable Precision

(a) Floating Point (32 bits/vertex coordinate) (b) Variable Precision (7.1 bits/vertex coordinate) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 13: Cyberware Venus (268K triangles) Rendered in Variable Precision



(a) Floating Point (32 bits/vertex coordinate) (b) Variable Precision (5.6 bits/vertex coordinate) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 14: Buddha Model(1087K triangles) Rendered in Variable Precision

Concepts and Algorithms for Polygonal Simplification

Jonathan D. Cohen

Department of Computer Science, The Johns Hopkins University

1. INTRODUCTION

1.1 Motivation

In 3D computer graphics, polygonal models are often used to represent individual objects and entire environments. Planar polygons, especially triangles, are used primarily because they are easy and efficient to render. Their simple geometry has enabled the development of custom graphics hardware, currently capable of rendering millions or even tens of millions of triangles per second. In recent years, such hardware has become available even for personal computers. Due to the availability of such rendering hardware and of software to generate polygonal models, polygons will continue to play an important role in 3D computer graphics for many years to come.

However, the simplicity of the triangle is not only its main advantage, but its main disadvantage as well. It takes many triangles to represent a smooth surface, and environments of tens or hundreds of millions of triangles or more are becoming quite common in the fields of industrial design and scientific visualization. For instance, in 1994, the UNC Department of Computer Science received a model of a notional submarine from the Electric Boat division

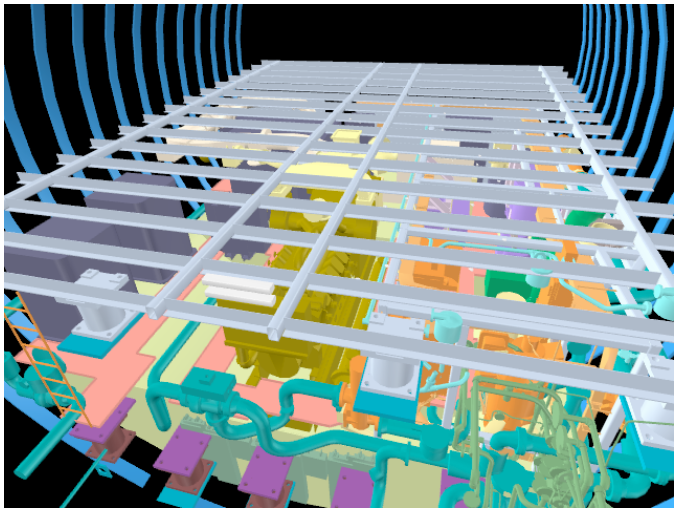


Figure 1: The auxiliary machine room of a notional submarine model: 250,000 triangles

of General Dynamics, including an auxiliary machine room composed of 250,000 triangles (see Figure 1) and a torpedo room composed of 800,000 triangles. In 1997, we received from ABB Engineering a coarsely-tessellated model of an entire coal-fired power plant, composed of over 13,000,000 triangles. It seems that the remarkable performance increases of 3D graphics hardware systems cannot yet match the desire and ability to generate detailed and realistic 3D polygonal models.

1.2 Polygonal Simplification

This imbalance of 3D rendering performance to 3D model size makes it difficult for graphics applications to achieve *interactive* frame rates (10-20 frames per second or more). Interactivity is an important property for applications such as architectural walkthrough, industrial design, scientific visualization, and virtual reality. To achieve this interactivity in spite of the enormity of data, it is often necessary to trade fidelity for speed.

We can enable this speed/fidelity tradeoff by creating a *multi-resolution* representation of our models. Given such a representation, we can render smaller or less important objects in the scene at a lower resolution (i.e. using fewer triangles) than the larger or more important objects, and thus we render fewer triangles overall. Figure 2 shows a widely-used test model: the Stanford bunny. This model was acquired using a laser range-scanning device; it contains over 69,000 triangles. When the 2D image of this model has a fairly large area, this may be a reasonable number of triangles to use for rendering the image. However, if the image is smaller, like Figure 3 or Figure 4, this number of triangles is probably too large. The right-most image in each of these figures shows a bunny with fewer triangles. These complexities are often more appropriate for image of these sizes. Each of these images is typically some small piece of a much larger image of a complex scene.

For CAD models, such representations could be created as part of the process of building the original model. Unfortunately, the robust modeling of 3D objects and environments is already a difficult task, so we would like to explore solutions that do not add extra burdens to the original modeling process. Also, we would like to create such representations for models acquired by other means (e.g. laser scanning), models that already exist, and models in the process of being built.

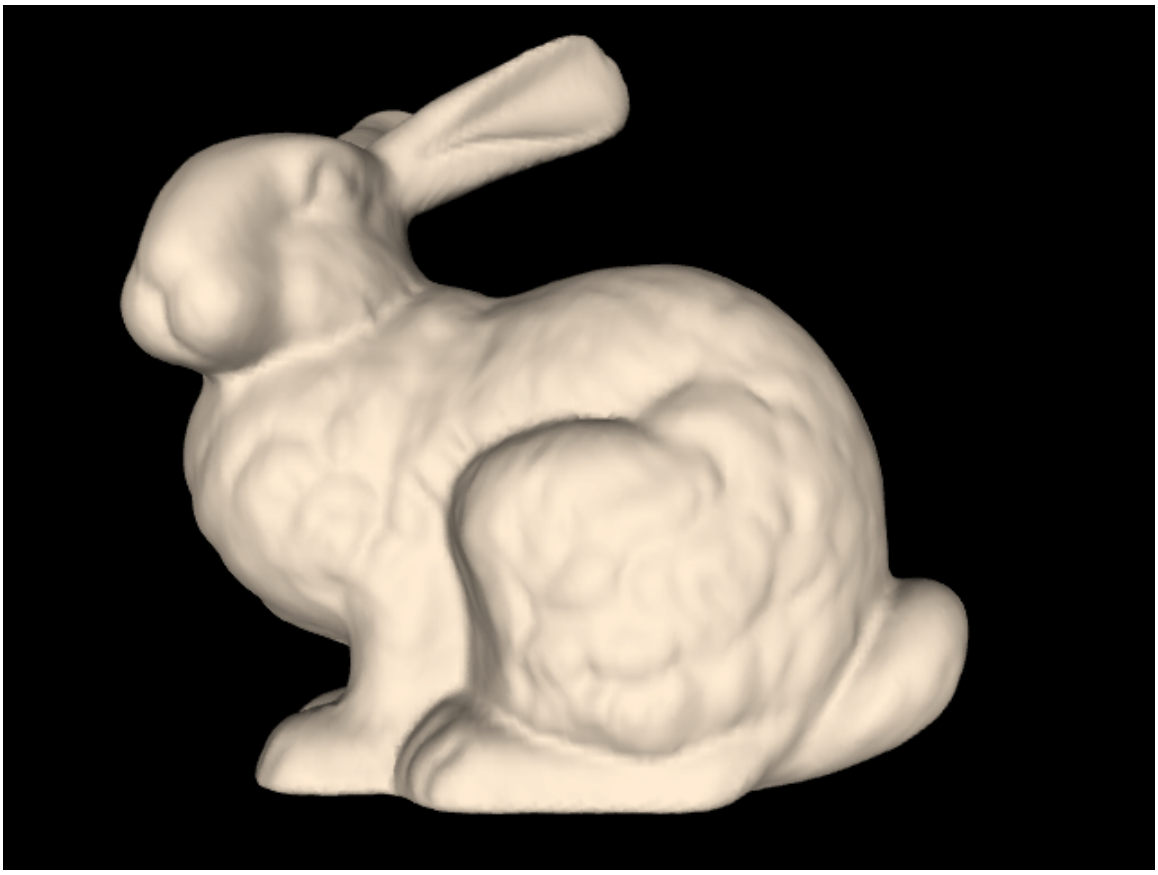


Figure 2: The Stanford bunny model: 69,451 triangles

69,451 triangles

2,204 triangles

Figure 3: Medium-sized bunnies.

69,451 triangles

575 triangles

Figure 4: Small-sized bunnies.

Simplification is the process of automatically reducing the complexity of a given model. By creating one or more simpler representations of the input model (generally called *levels of detail*), we convert it to a multi-resolution form. This problem of automatic simplification is rich enough to provide many interesting and useful avenues of research. There are many issues related to how we represent these multi-resolution models, how we create them, and how we manage them within an interactive graphics application. This dissertation is concerned primarily with the issues of level-of-detail quality and rendering performance. In particular, we explore the question of how to preserve the appearance of the input models to within an intuitive, user-specified tolerance and still achieve a significant increase in rendering performance.

1.3 Topics Covered

This paper reviews some fundamental concepts necessary to understand algorithms for simplification of polygonal models at a high level. These concepts include optimal/near-optimal solutions for the simplification problem, the use of local simplification operations, topology preservation, level-of-detail representations for polygonal models, error measures for surface deviation, and the preservation of appearance attributes. This is not a complete survey of the field of polygonal model simplification, which has grown to be quite large (for more information, several survey papers are available [Erikson 1996, Heckbert and Garland 1997]). In particular, this paper does *not* provide much coverage of algorithms specialized for simplifying polygonal terrains, nor does it cover simplification and compression algorithms geared towards progressive transmission applications.

2. OPTIMALITY

There are two common formulations of the simplification problem, described in [Varshney 1994], to which we may seek optimal solutions:

- **Min-# Problem**: Given some error bound, ε, and an input model, I, compute the minimum complexity approximation, A, such that no point of A is farther than ε distance away from I and vice versa (the complexity of A is measured in terms of number of vertices or faces).

- **Min-ε Problem**: Given some target complexity, n, and an input model, I, compute the approximation, A, with the minimum error, ε, described above.

In computational geometry, it has been shown that computing the min-# problem is NP-hard for both convex polytopes [Das and Joseph 1990] and polyhedral terrains [Agarwal and Suri 1994]. Thus, algorithms to solve these problems have evolved around finding polynomial-time approximations that are *close* to the optimal.

Let k_0 be the size of a min-# approximation. An algorithm has been given in [Mitchell and Suri 1992] for computing an ε-approximation of size $O(k_0 \log n)$ for convex polytopes of initial complexity n. This has been improved by Clarkson in [Clarkson 1993]; he proposes a randomized algorithm for computing an approximation of size $O(k_0 \log k_0)$ in expected time $O(k_0 n^{1+\delta})$ for any $\delta > 0$ (the constant of proportionality depends on δ, and tends to $+\infty$ as δ tends to 0). In [Brönnimann and Goodrich 1994] Brönnimann and Goodrich observed that a variant of Clarkson's algorithm yields a polynomial-time deterministic algorithm that computes an approximation of size $O(k_0)$. Working with polyhedral terrains, [Agarwal and Suri 1994] present a polynomial-time algorithm that computes an ε-approximation of size $O(k_0 \log k_0)$ to a polyhedral terrain.

Because the surfaces requiring simplification may be quite complex (tens of thousands to millions of triangles), the simplification algorithms used in practice must be $o(n^2)$ (typically $O(n \log n)$) for the running time to be reasonable. Due to the difficulty of computing near-optimal solutions for general polygonal meshes and the required efficiency, most of the algorithms described in the computer graphics literature employ local, greedy heuristics to achieve what appear to be reasonably good simplifications with no guarantees with respect to the optimal solution.

3. LOCAL SIMPLIFICATION OPERATIONS

Simplification is often achieved by performing a series of local operations. Each such operation serves to coarsen the polygonal model by some small amount. A simplification algorithm generally chooses one of these operation types and applies it repeatedly to its input surface until the desired complexity is achieved for the output surface.

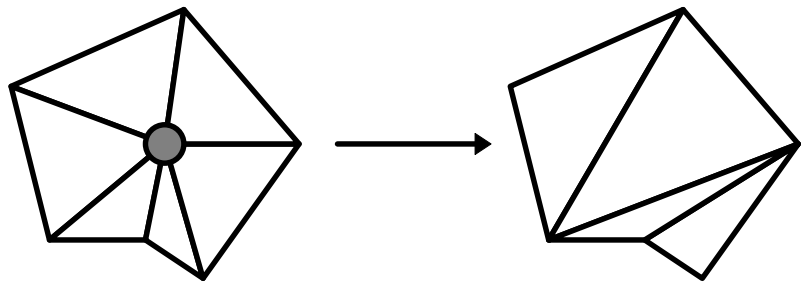


Figure 5: Vertex remove operation

3.1 Vertex Remove

The vertex remove operation involves removing from the surface mesh a single vertex and all the triangles touching it. This removal process creates a hole that we then fill with a new set of triangles. Given a vertex with *n* adjacent triangles, the removal process creates a hole with *n* sides. The hole filling problem involves a discrete choice from among a finite number of possible retriangulations for the hole. The *n* triangles around the vertex are replaced by this new triangulation with *n*-2 triangles. The Catalan sequence,

$$\mathrm{C}(i) = \frac{1}{i+1} * \binom{2i}{i} = \frac{1}{i+1} * \frac{(2i)!}{i!(2i-i)!} = \frac{1}{i+1} * \frac{(2i)!}{i!i!} = \frac{(2i)!}{(i+1)!i!} \ , \qquad (1)$$

describes the number of unique ways to triangulate a convex, planar polygon with *i*+2 sides [Dörrie 1965, Plouffe and Sloan 1995]. This provides an upper bound on the number of non-self-intersecting triangulations of a hole in 3D. For example, holes with 3 sides have only 1 triangulation, and holes with 4, 5, 6, 7, 8, and 9 sides have up to 2, 5, 14, 42, 132, and 429 triangulations, respectively.

Both [Turk 1992] and [Schroeder et al. 1992] apply the vertex remove approach as part of their simplification algorithms. Turk uses point repulsion (weighted according to curvature) to distribute some number of new vertices across the original surface, then applies vertex remove operations to remove most of the original vertices. Holes are retriangulated using a planar projection approach. Schroeder also uses vertex remove operations to reduce mesh complexity, employing a recursive loop splitting algorithm to fill the necessary holes.

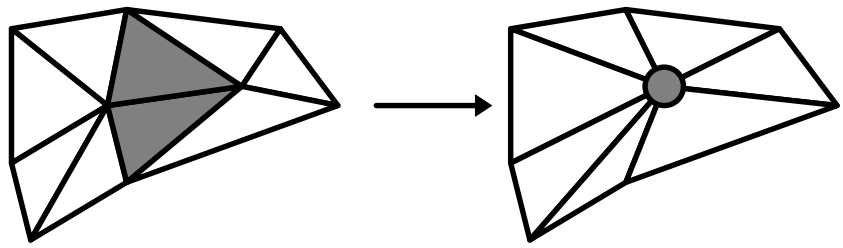


Figure 6: Edge collapse operation

3.2 Edge Collapse

The edge collapse operation has become popular in the graphics community in the last several years. The two vertices of an edge are merged into a single vertex. This process distorts all the neighboring triangles. The triangles that contain both of the vertices (i.e. those that touch the entire edge) degenerate into 1-dimensional edges and are removed from the mesh. This typically reduces the mesh complexity by 2 triangles.

Whereas the vertex remove operation amounts to making a discrete choice of triangulations, the edge collapse operation requires us to choose the coordinates of the new vertex from a continuous domain. Common choices for these new coordinates include the coordinates of one of the two original vertices, the midpoint of the collapsed edge, arbitrary points along the collapsed edge, or arbitrary points in the neighborhood of the collapsed edge.

Not only is the choice of new vertex coordinates for the edge collapse a continuous problem, but the actual edge collapse operation may be performed continuously in time. We can linearly interpolate the two vertices from their original positions to the final position of the new vertex. This allows us to create smooth transitions as we change the mesh complexity. As described in [Hoppe 1996], we can even perform *geomorphs*, which smoothly transition between versions of the model with widely varying complexity by performing many of these interpolations simultaneously.

In terms of the ability to create identical simplifications, the vertex removal and edge collapse operations are not equivalent. If we collapse an edge to one of its original vertices, we can create n of the triangulations possible with the vertex remove, but there are still $C(n+2)-n$ triangulations that the edge collapse cannot create. Of course, if we allow the edge collapse to choose arbitrary coordinates for its new vertex, it can create infinitely many simplifications that the vertex remove operation cannot create. For a given input model and

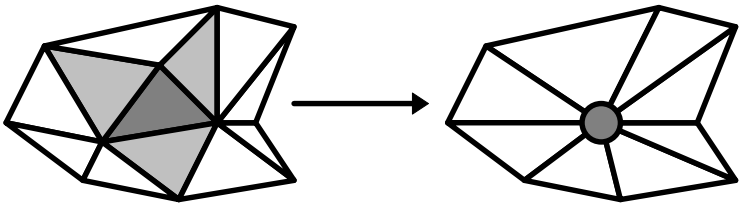


Figure 7: Face collapse operation

desired output complexity, it is not clear which type of operation can achieve a closer approximation to the input model.

The edge collapse was used by [Hoppe et al. 1993] as part of a mesh optimization process that employed the vertex remove and edge swap operations as well (the edge swap is a discrete operation that takes two triangles sharing an edge and swaps which pair of opposite vertices are connected by the edge). In [Hoppe 1996], the vertex remove and edge swaps are discarded, and the edge collapse alone is chosen as the simplification operation, allowing a simpler system that can take advantage of the features of the edge collapse. Although systems employing multiple simplification operations might possibly result in better simplifications, they are generally more complex and cannot typically take advantage of the inherent features of any one operation.

3.3 Face Collapse

The face collapse operation is similar to the edge collapse operation, except that it is more coarse-grained. All three vertices of a triangular face are merged into a single vertex. This causes the original face to degenerate into a point and three adjacent faces to degenerate into line segments, removing a total of four triangles from the model. The coarser granularity of this operation may allow the simplification process to proceed more quickly, at the expense of the fine-grained local control of the edge collapse operation. Thus, the error is likely to accumulate more quickly for a comparable reduction in complexity. [Hamann 1994, Gieng et al. 1997] use the face collapse operation in their simplification systems. The new vertex coordinates are chosen to lie on a local quadratic approximation to the mesh. Naturally, it is possibly to further generalize these collapse operations to collapse even larger connected portions of the input model. It may even be possible to reduce storage requirements by grouping nearby collapse operations with similar error bounds into larger collapse operations.

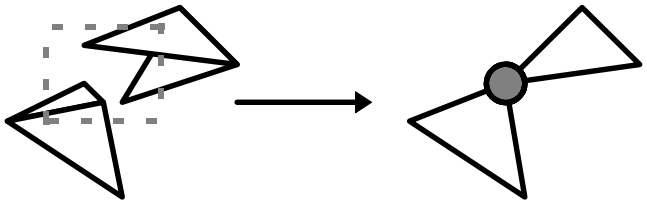


Figure 8: Vertex Cluster operation

Thus, the fine-grained control may be traded for reduced storage and other overhead requirements in certain regions of the model.

3.4 Vertex Cluster

Unlike the preceding simplification operations, the vertex cluster operation relies solely on the geometry of the input (i.e. the vertex coordinates) rather than the topology (i.e. the adjacency information) to reduce the complexity. Like the edge and face collapses, several vertices are merged into a single vertex. However, rather than merging a set of topologically adjacent vertices, a set of "nearby" vertices are merged [Rossignac and Borrel 1992]. For instance, one possibility is to merge all vertices that lie within a particular 3D axis-aligned box. The new, merged vertex may be one of the original vertices that "best represents" the entire set, or it may be placed arbitrarily to minimize some error bound. An important property of this operation is that it can be robustly applied to arbitrary sets of triangles, whereas all the preceding operations assume that the triangles form a connected, manifold mesh.

The effects of this vertex cluster are similar to those of the collapse operations. Some triangles are distorted, whereas others degenerate to a line segment or a point. In addition, there may be coincident triangles, line segments, and points originating from non-coincident geometry. One may choose to render the degenerate triangles as line segments and points, or one may simply not render them at all. Depending on the particular graphics engine, rendering a line or a point may not be much faster than rendering a triangle. This is an important consideration, because achieving a speed-up is one of the primary motivations for simplification.

There is no point in rendering several coincident primitives, so multiple copies are filtered down to a single copy. However, the question of how to render coincident geometry is complicated by the existence of other surface attributes, such as normals and colors. For

instance, suppose two triangles of wildly different colors become coincident. No matter what color we render the triangle, it may be noticeably incorrect.

[Rossignac and Borrel 1992] use the vertex clustering operation in their simplification system to perform very fast simplification on arbitrary polygonal models. They partition the model space with a uniform grid, and vertices are collapsed within each grid cell. [Luebke and Erikson 1997] build an octree hierarchy rather than a grid at a single resolution. They dynamically collapse and split the vertices within an octree cell depending on the current size of the cell in screen space as well as silhouette criteria.

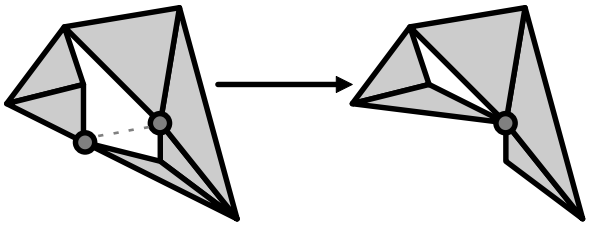


Figure 9: Generalized edge collapse operation

3.5 Generalized Edge Collapse

The generalized edge collapse (or vertex pair) operation combines the fine-grained control of the edge collapse operation with the generality of the vertex cluster operation. Like the edge collapse operation, it involves the merging of two vertices and the removal of degenerate triangles. However, like the vertex cluster operation, it does not require that the merged vertices be topologically connected (by a topological edge), nor does it require that topological edges be manifold.

[Garland and Heckbert 1997] apply the generalized edge collapse in conjunction with error quadrics to achieve simplification that gives preference to the collapse of topological edges, but also allows the collapse of virtual edges (arbitrary pairs of vertices). These virtual edges are chosen somewhat heuristically, based on proximity relationships in the original mesh.

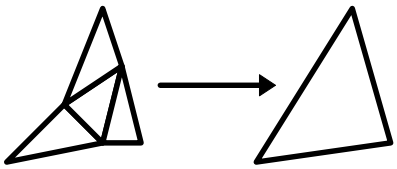


Figure 10: Unsubdivide operation

3.6 Unsubdivide

Subdivision surface representations have also been proposed as a solution to the multi-resolution problem. In the context of simplification operations, we can think of the “unsubdivide” operation (the inverse of a subdivision refinement) as our simplification operation. A common form of subdivision refinement is to split one triangle into four triangles. Thus the unsubdivide operation merges four triangles of a particular configuration into a single triangle, reducing the triangle count by three triangles.

[DeRose et al. 1993] shows how to represent a subdivision surface at some finite resolution as a sequence of wavelet coefficients. The sequence of coefficients is ordered from lower to higher frequency content, so truncating the sequence at a particular point determines a particular mesh resolution. [Eck et al. 1995] presents an algorithm to turn an arbitrary topology mesh into one with the necessary subdivision connectivity. They construct a base mesh of minimal resolution and guide its refinement to come within some tolerance of the original mesh. This new refined subdivision mesh is used in place of the original mesh, and its resolution is controlled according to the wavelet formulation.

4. TOPOLOGICAL CONSIDERATIONS

4.1 Manifold vs. Non-manifold Meshes

Polygonal simplification algorithms may be distinguished according to the type of input they accept. Some algorithms require the input to be a *manifold* triangle mesh, while others accept more general triangle sets. In the continuous domain, a manifold surface is one that is everywhere homeomorphic to an open disc. In the discrete domain of triangle meshes, such a surface has two topological properties. First, every vertex is adjacent to a set of triangles that form a single, complete cycle around the vertex. Second, each edge is adjacent to exactly two triangles. For a manifold mesh with *borders*, these restrictions are slightly relaxed. A border is simply a chain of edges with adjacent triangles only to one side. In a manifold mesh with borders, a vertex may be surrounded by a single, incomplete cycle (i.e. the beginning need not meet the end). Also, an edge may be adjacent to either one or two triangles.

A mesh that does not have the above properties is said to be *non-manifold*. Such meshes may occur in practice by accident or by design. Accidents are possible, for example, during either the creation of the mesh or during conversions between representation, such as the conversion from a solid to a boundary representation. The correction of such accidents is a subject of much interest [Barequet and Kumar 1997, Murali and Funkhouser 1997]. They may occur by design because such a mesh may require fewer triangles to render than a visually-comparable manifold mesh or because such a mesh may be easier to create in some situations. If the non-manifold portions of a mesh are few and far between, we may refer to the mesh as *mostly manifold*.

At the extreme, some data sets take the form of a set of triangles, with no connectivity information whatsoever (sometimes referred to as a "triangle soup"). Such data might turn out to be manifold or non-manifold if we were to attempt to reconstruct the connectivity information. In general, if any conversion has been performed on the original data, it's safe to assume that a naïve reconstruction will result in at least some non-manifold regions.

The most robust algorithms, based on vertex clusters, operate as easily on a triangle soup as on a perfectly manifold mesh [Rossignac and Borrel 1992], [Luebke and Erikson 1997]. This advantage cannot be stressed enough and is extremely important in the case where the simplification user has no control over the data. The ability to view an large, unfamiliar data set interactively is invaluable in the process of learning its ins and outs, and these algorithms allow one to get up and running quickly.

However, these very general algorithms do not typically create simplifications that look as attractive as those produced by algorithms that operate on manifold meshes. These algorithms, which rely on operations such as the vertex remove or edge collapse, respect the topology of the original mesh and avoid catastrophic changes to the surface and its appearance. The manifold input criterion does limit the applicability of these algorithms to some real-world models, but many of these algorithms may be modified to handle mostly manifold meshes by avoiding simplification of the non-manifold regions. This can be an effective strategy until the non-manifold regions begin to dominate the surface complexity.

The vertex pair and edge collapse operations can both operate on non-manifold meshes as well as manifold ones. Vertex-pair algorithms must deal with the non-manifold meshes they are bound to create by merging non-adjacent vertices. Edge collapse algorithms can operate on non-manifold meshes, but it may be difficult to adapt the most rigorous error metrics for manifold meshes to use on non-manifold meshes.

4.2 Topology Preservation

The topological structure of a polygonal surface typically refers to features such as its *genus* (number of topological holes, e.g. 0 for a sphere, 1 for a torus or coffee mug) and the number and arrangement of its borders. These features are fully determined by the adjacency graph of the vertices, edges, and faces of a polygonal mesh. For manifold meshes with no borders (i.e. closed surfaces), the Euler equation holds:

$$F - E + V = 2 - G, \tag{2}$$

where F is the number of faces, E is the number of edges, V is the number of vertices, and G is the genus.

In addition to this combinatorial description of the topological structure, the embedding of the surface in 3-space impacts its perceived topology in 3D renderings. Generally, we expect the faces of a surface to intersect only at their shared edges and vertices.

Most of the simplification operations described in section 3 (all except the vertex cluster and the generalized edge collapse) preserve the connectivity structure of the mesh. If a simplification algorithm uses such an operation and also prevents local self-intersections (intersections within the adjacent neighborhood of the operation), we say the algorithm *preserves local topology*. If the algorithm prevents any self-intersections in the entire mesh, we say it *preserves global topology*.

If the simplified surface is to be used for purposes other than rendering (e.g. finite element computations), topology preservation may be essential. For rendering applications, however, it is not always necessary. In fact, it is often possible to construct simplifications with fewer polygons for a given error bound if topological modifications are allowed.

However, some types of topological modifications may have a dramatic impact on the appearance of the surface. For instance, many meshes are the surfaces of solid objects. For example, consider the surface of a thin, hollow cylinder. When the surface is modified by more than the thickness of the cylinder wall, the interior surface will intersect the outer surface. This can cause artifacts that cover a large area on the screen. Problems also occur when polygons with different color attributes become coincident.

Certain types of topological changes are clearly beneficial in reducing complexity, and have a smaller impact on the rendered image. These include the removal of topological holes and thin features (such as the antenna of a car). Topological modifications are encouraged in [Rossignac and Borrel 1992], [Luebke and Erikson 1997], [Garland and Heckbert 1997] and [Erikson and Manocha 1998] and controlled modifications are performed in [He et al. 1996] and [El-Sana and Varshney 1997].

5. LEVEL-OF-DETAIL REPRESENTATIONS

We can classify the possible representations for level-of-detail models into two broad categories: *static* and *dynamic*. Static levels of details are computed totally off-line. They are fully determined as a pre-process to the visualization program. Dynamic levels of detail are typically computed partially off-line and partially on-line within the visualization program. We now discuss these representations in more detail.

5.1 Static Levels of Detail

The most straightforward level-of-detail representation for an object is a set of independent meshes, where each mesh has a different number of triangles. A common heuristic for the generation of these meshes is that the complexity of each mesh should be reduced by a factor of two from the previous mesh. Such a heuristic generates a reasonable range of complexities, and requires only twice as much total memory as the original representation.

It is common to organize the objects in a virtual environment into a hierarchical *scene graph* [van Dam 1988, Rohlf and Helman 1994]. Such a scene graph may have a special type of node for representing an object with levels of detail. When the graph is traversed, this

level-of-detail node is evaluated to determine which child branch to traverse (each branch represents one of the levels of detail). In most static level-of-detail schemes, the children of the level-of-detail nodes are the leaves of the graph. [Erikson and Manocha 1998] presents a scheme for generating *hierarchical levels of detail*. This scheme generates level-of-detail nodes throughout the hierarchy rather than just at the leaves. Each such interior level-of-detail node involves the merging of objects to generate even simpler geometric representations. This overcomes one of the previous limitations of static levels of detail — the necessity for choosing a single scale at which objects are identified and simplified.

The transitions between these levels of detail are typically handled in one of three ways: discrete, blended, or morphed. The discrete transitions are instantaneous switches; one level of detail is rendered during one frame, and a different level of detail is rendered during the following frame. The frame at which this transition occurs is typically determined based on the distance from the object to the viewpoint. This technique is the most efficient of the three transition types, but also results in the most noticeable artifacts.

Blended transitions employ alpha-blending to fade between the two levels of detail in question. For several frames, both levels of detail are rendered (increasing the rendering cost during these frames), and their colors are blended. The blending coefficients change gradually to fade from one level of detail to the other. It is possible to blend over a fixed number of frames when the object reaches a particular distance from the viewpoint, or to fade over a fixed range of distances [Rohlf and Helman 1994]. If the footprints of the objects on the screen are not identical, blending artifacts may still occur at the silhouettes.

Morphed transitions involve gradually changing the shape of the surface as the transition occurs. This requires the use of some correspondence between the two levels of detail. Only one representation must be rendered for each frame of the transition, but the vertices require some interpolation each frame. For instance, [Hoppe 1996] describes the *geomorph* transition for levels of detail created by a sequence of edge collapses. The simpler level of detail was originally generated by collapsing some number of vertices, and we can create a transition by simultaneously interpolating these vertices from their positions on one level of detail to their positions on the other level of detail. Thus the number of triangles we render during the

transition is equal to the maximum of the numbers of triangles in the two levels of detail. It is also possible to morph using a mutual tessellation of the two levels of detail, as in [Turk 1992], but this requires the rendering of more triangles during the transition frames.

5.2 Dynamic Levels of Detail

Dynamic levels of detail provide representations that are more carefully tuned to the viewing parameters of each particular rendered frame. Due to the sheer number of distinct representations this requires, each representation cannot simply created and stored independently. The common information among these representations is used to create a single representation for each simplified object. From this unified representation, a geometric representation that is tuned to the current viewing parameters is extracted. The coherence of the viewing parameters enables incremental modifications to the geometry rendered in the previous frame; this makes the extraction process feasible at interactive frame rates.

[Hoppe 1996] presents a representation called the *progressive mesh*. This representation is simply the original object plus an ordered list of the simplification operations performed on the object. It is generally more convenient to reverse the order of this intuitive representation, representing the simplest *base mesh* plus the inverse of each of the simplification operations. Applying all of these inverse operations to the base mesh will result in the original object representation. A particular level of detail of this progressive mesh is generated by performing some number of these operations.

In [Hoppe 1997], the progressive mesh is reorganized into a vertex hierarchy. This hierarchy is a tree that captures the dependency of each simplification operation on certain previous operations. Similar representations include the *merge tree* of [Xia et al. 1997], the *multiresolution model* of [Klein and Krämer 1997], the *vertex tree* of [Luebke and Erikson 1997], and the *multi-triangulation* of [DeFloriani et al. 1997]. Such hierarchies allow selective refinement of the geometry based on various metrics for screen-space deviation, normal deviation, color deviation, and other important features such as silhouettes and specular highlights. A particular level of detail may be expressed as a *cut* through these graphs, or a *front* of vertex nodes. Each frame, the nodes on the current front are examined, and may cause the graph to be refined at some of these nodes.

[DeFloriani et al. 1997] discuss the properties of such hierarchies in terms of graph characteristics. Examples of these properties include compression ratio, linear growth, logarithmic height, and bounded width. They discuss several different methods of constructing such hierarchies and test these methods on several benchmarks. For example, one common heuristic for building these hierarchies is to choose simplification operations in a greedy fashion according to an error metric. Another method is to choose a set of operations with disjoint areas of influence on the surface and apply this entire set before choosing the next set. The former method does not guarantee logarithmic height, whereas the latter does. Such height guarantees can have practical implications in terms of the length of the chain of dependent operations that must be performed in order to achieve some particular desired refinement.

[DeRose et al. 1993] present a wavelet-based representation for surfaces constructed with subdivision connectivity. [Eck et al. 1995] make this formulation applicable to arbitrary triangular meshes by providing a remeshing algorithm to approximate an arbitrary mesh by one with the necessary subdivision connectivity. Both the remeshing and the filtering/reconstruction of the wavelet representation provide bounded error on the surfaces generated. [Lee et al. 1998] provide an alternate remeshing algorithm based on a smooth, global parameterization of the input mesh. Their approach also allows the user to constrain the parameterization at vertices or along edges of the original mesh to better preserve important features of the input.

5.3 Comparison

Static levels of detail allow us to perform simplification entirely as a pre-process. The real-time visualization system performs only minimal work to select which level of detail to render at any given time. Because the geometry does not change, it may be rendered in retained mode (i.e. from cached, optimized *display lists*). Retained-mode rendering should always be at least as fast as immediate mode rendering, and is much faster on most current high-end hardware. Perhaps the biggest shortcoming of using static levels of detail is that they require that we partition the model into independent “objects” for the purpose of simplification. If an object is large with respect to the user or the environment, especially if the viewpoint is often contained inside the object, little or no simplification may be possible.

This may require that such objects be subdivided into smaller objects, but switching the levels of detail of these objects independently causes visible cracks, which are non-trivial to deal with.

Dynamic levels of detail perform some of simplification as a pre-process, but defer some of the work to be computed by the real-time visualization system at run time. This allows us to provide more fine-tuning of the exact tessellation to be used, and allows us to incorporate more view-dependent criteria into the determination of this tessellation. The shortcoming of such dynamic representations is that they require more computation in the visualization system as well as the use of immediate mode rendering. Also, the memory requirements for such representations are often somewhat larger than for the static levels of detail.

6. SURFACE DEVIATION ERROR BOUNDS

Measuring the deviation of a polygonal surface as a result of simplification is an important component of the simplification process. This surface deviation error gives us an idea of the quality of a particular simplification. It helps guide the simplification process to produce levels of detail with low error, determine when it is appropriate to show a particular level of detail of a given surface, and optimize the levels of detail for an entire scene to achieve a high overall image quality for the complexity of the models actually rendered.

6.1 Distance Metrics

Before discussing the precise metrics and methods used by several researchers for measuring surface deviation, we consider two formulations of the distance between two surfaces. These are the Hausdorff distance and the mapping distance. The Hausdorff distance is a well-known concept from topology, used in image processing as well as surface modeling, and the mapping distance is a commonly used metric for parametric surfaces.

6.1.1 Hausdorff Distance

The Hausdorff distance is a distance metric between point sets. Given two sets of points, A and B, the Hausdorff distance is defined as

$$\mathrm{H}(A,B) = \max(\mathrm{h}(A,B), \mathrm{h}(B,A)) , \qquad (3)$$

where

$$\mathrm{h}(A,B) = \max_{a \in A} \min_{b \in B} \left\| a - b \right\|. \tag{4}$$

Thus the Hausdorff distance measures the farthest distance from a point in one point set to its closest point in the other point set (notice that h(A,B) ≠ h(B,A)). Because a surface is a particular type of continuous point set, the Hausdorff distance provides a useful measure of the distance between two surfaces.

6.1.2 Mapping Distance

The biggest shortcoming of the Hausdorff distance metric for measuring the distance between surfaces is that it makes no use of the point neighborhood information inherent in the surfaces. The function h(A,B) implicitly assigns to each point of surface A the closest point of surface B. However, this mapping may have discontinuities. If points i and j are "neighboring" points on surface A (i.e. there is a path on the surface of length no greater than ε that connects them), their corresponding points, i' and j', on surface B may not be neighboring points. In addition, the mapping implied by h(A,B) is not identical to the mapping implied by h(B,A).

For the purpose of simplification, we would like to establish a continuous mapping between the surface's levels of detail. Ideally, the correspondences described by this mapping should coincide with a viewer's perception of which points are "the same" on the surfaces. Given such a continuous mapping

$$\mathrm{F}: A \rightarrow B$$

the mapping distance is defined as

$$\mathrm{D}(\mathrm{F}) = \max_{a \in A} \left\| a - \mathrm{F}(a) \right\|. \tag{5}$$

Because there are many such mappings, there are many possible mapping distances. The minimum mapping distance is simply

$$\mathrm{D}_{\min} = \min_{\mathrm{F} \in M} \mathrm{D}(\mathrm{F}), \tag{6}$$

where M is the set of all such continuous mapping functions. Note that although D_{min} and its associated mapping function may be difficult to compute, all continuous mapping functions provide an upper bound on D_{min}.

6.2 Surface Deviation Algorithms

We now classify several simplification algorithms according to how they measure the surface deviation error of their levels of detail.

6.2.1 Mesh Optimization

[Hoppe et al. 1993] pose the simplification problem in terms of optimizing an energy function. This function has terms corresponding to number of triangles, surface deviation error, and a heuristic spring energy. To quantify surface deviation error, they maintain a set of point samples from the original surface and their closest distance to the simplified surface. The sum of squares of these distances is used as the surface deviation component of the energy function. The spring energy term is required because the surface deviation error is only measured in one direction: it approximates the closest distance from the original surface to the simplified surface, but not vice versa. Without this term, small portions of the simplified surface can deviate quite far from the original surface, as long as all the point samples are near to some portion of the simplified surface.

6.2.2 Vertex Clustering

[Rossignac and Borrel 1993] present a simple and general algorithm for simplification using vertex clustering. The vertices of each object are clustered using several different sizes of uniform grid. The surface deviation in this case is a Hausdorff distance and must be less than or equal to the size of grid cell used in determining the vertex clusters. This is a very conservative bound, however. A slightly less conservative bound is the maximum distance from a vertex in the original cluster to the single representative vertex after the cluster is collapsed. Even this bound is quite conservative in many cases; the actual maximum deviation from the original surface to the simplified surface may be considerably smaller than the distance the original vertices travel during the cluster operation.

[Luebke and Erikson 1997] take a similar approach, but their system uses an octree instead of a single-resolution uniform grid. This allows them to take a more dynamic approach, folding and unfolding octree cells at run-time and freely merging nearby objects. The measure of surface deviation remains the same, but they allow a more flexible choice of error tolerances in their run-time system. In particular, they use different tolerances for silhouette and non-silhouette clusters.

6.2.3 Superfaces

[Kalvin and Taylor 1996] present an efficient simplification algorithm based on merging adjacent triangles to form polygonal patches, simplifying the boundaries of these patches, and finally retriangulating the patches themselves. This algorithm guarantees a maximum deviation from vertices of the original surface to the simplified surface and from vertices of the simplified surface to the original surface. Unfortunately, even this bidirectional bound does not guarantee a maximum deviation between points on the simplified surface and points on the original surface. For instance, suppose we have two adjacent triangles that share an edge, forming a non-planar quadrilateral. If we retriangulate this quadrilateral by performing an edge swap operation, the maximum deviation between these two surfaces is non-zero, even though their four vertices are unchanged (thus the distance measured from vertex to surface is zero).

6.2.4 Error Tolerance Volumes

[Guéziec 1995] presents a simplification system that measures surface deviation using error volumes built around the simplified surface. These volumes are defined by spheres, specified by their radii, centered at each of the simplified surface's vertices. We can associate with any point in a triangle a sphere whose radius is a weighted average of the spheres of the triangle's vertices. The error volume of an entire triangle is the union of the spheres of all the points on the triangle, and the error volume of a simplified surface is the union of the error volumes of its triangles. As edge collapses are performed, not only are the coordinates of the new vertex computed, but new sphere radii are computed such that the new error volume contains the previous error volume. The maximum sphere radius is a bound on the Hausdorff

distance of the simplified surface from the original, and thus provides a bound for surface deviation in both 3D and 2D (after perspective projection).

6.2.5 Simplification Envelopes

The simplification envelopes technique of [Cohen and Varshney et al. 1996] bounds the Hausdorff distance between the original and simplified surfaces without actually making measurements during the simplification process. For a particular simplification, the input surface is surrounded by two envelope surfaces, which are constructed to deviate by no more than a specified tolerance, ε, from the input surface. As the simplification progresses, the modified triangles are tested for intersection with these envelopes. If no intersections occur, the simplified surface is within distance ε from the input surface. Similar constructions are built to constrain error around the borders of bordered surfaces. By including extensive self-intersection testing as well, the algorithm provides complete global topology preservation. This algorithm does an excellent job at generating small-triangle-count surface approximations for a given error bound. The biggest limitations are the up-front processing costs required for envelope construction (for each level of detail to be generated) and the conservative nature of the envelopes themselves, which do not expand beyond the point of self-intersection.

6.2.6 Error Quadrics

[Ronfard and Rossignac 1996] describe a fast method for approximating surface deviation. They represent surface deviation error for each vertex as a sum of squared distances to a set of planes. The initial set of planes for each vertex are the planes of its adjacent faces. As vertices are merged, the sets of planes are unioned. This metric provides a useful and efficient heuristic for choosing an ordering of edge collapse operations, but it does not provide any guarantees about the maximum or average deviation of the simplified surface from the original.

[Garland and Heckbert 1997] present some improvements over [Ronfard and Rossignac 1996]. The error metric is essentially the same, but they show how to approximate a vertex's set of planes by a quadric form (represented by a single 4x4 matrix). These matrices are simply added to propagate the error as vertices are merged. Using this metric, it is possible to

choose an optimal vertex placement that minimizes the error. In addition, they allow the merging of vertices that are not joined by an edge, allowing increased topological modification. [Erikson and Manocha 1998] further improve this technique by automating the process of choosing which non-edge vertices to collapse and by encouraging such merging to preserve the local surface area.

6.2.7 Mapping Error

[Bajaj and Schikore 1996] perform simplification using the vertex remove operation, and measure surface deviation using local, bijective (one-to-one and onto) mappings in the plane between points on the surface just before and just after the simplification operation. This approach provides a fairly tight bound on the maximum deviation over all points on the surface, not just the vertices (as does [Guéziec 1995]) and provides pointwise mappings between the original and simplified surfaces.

A similar technique is employed by [Cohen et al. 1997], who perform mappings in the plane for the edge collapse operation. They present rigorous and efficient techniques for finding a plane in which to perform the mapping, as well as applying the mapping and propagating error from operation to operation. The computed mappings are used not only to guide the simplification process in its choice of operations, but also to assign texture coordinates to the post-collapse vertices and to control the switching of levels of detail in interactive graphics applications.

6.2.8 Hausdorff Error

[Klein et al. 1996] measure a one-sided Hausdorff distance (with appropriate locality restrictions) between the original surface and the simplified surface. By definition, this approach produces the smallest possible bound on maximum one-sided surface deviation, but the one-sided formulation does not guarantee a true bound on overall maximum deviation. At each step of the simplification process, the Hausdorff distance must be measured for each of the original triangles mapping to the modified portion of the surface. The computation time for each simplification operation grows as the simplified triangles cover more and more of the mesh, but of course, there are also fewer and fewer triangles to simplify. [Klein and Krämer 1997] present an efficient implementation of this algorithm.

6.2.9 Memory-efficient Simplification

[Lindstrom and Turk 1998] demonstrate the surprising result that good simplifications are possible without measuring anything with respect to the original model. All errors in this method are measured purely as incremental changes in the local surface. The error metric used preserves the total volume while minimizing volume changes of each triangle. Another interesting aspect of this work is that they perform after-the-fact measurements to compare the "actual" mean and maximum simplification errors of several algorithm implementations. These measurement use the *Metro* geometric comparison tool [Cignoni et al. 1996], which uniformly samples the simplified surface, computes correspondences with the original surface, and measures the error of the samples.

7. APPEARANCE ATTRIBUTE PRESERVATION

We now classify several algorithms according to how they preserve the appearance attributes of their input models.

7.1 Scalar Field Deviation

The mapping algorithm presented in [Bajaj and Schikore 1996] allows the preservation of arbitrary scalar fields across a surface. Such scalar fields are specified at the mesh vertices and linearly interpolated across the triangles. Their approach computes a bound on the maximum deviation of the scalar field values between corresponding points on the original surface and the simplified surface.

7.2 Color Preservation

[Hughes et al. 1996] describes a technique for simplifying colored meshes resulting from global illumination algorithms. They use a logarithmic function to transform the vertex colors into a more perceptually linear space before applying simplification. They also experiment with producing mesh elements that are quadratically- or cubically-shaded in addition to the usual linearly-shaded elements.

[Hoppe 1996] extends the error metric of [Hoppe et al. 1993] to include error terms for scalar attributes and discontinuities as well as surface deviation. Like the surface deviation,

the scalar attribute deviation is measured as a sum of squared Euclidean distances in the attribute space (e.g. the RGB color cube). The distances are again measured between sampled points on the original surface and their closest points on the simplified surface. This metric is useful for prioritizing simplification operations in order of increasing error. However, it does not provide much information about the true impact of attribute error on the final appearance of the simplified object on the screen. A better metric should incorporate some degree of area weighting to indicate how the overall illuminance of the final pixels may be affected.

[Erikson and Manocha 1998] present a method for measuring the maximum attribute deviation in Euclidean attribute spaces. Associated with each vertex is an attribute volume for each attribute being measured. The volume is a disc of the appropriate dimension (i.e. an interval in 1D, a circle in 2D, a sphere in 3D, etc.). Each attribute volumes is initially a point in the attribute space (an *n*-disk with radius zero). As vertex pairs are merged, the volumes grow to contain the volumes of both vertices.

[Garland and Heckbert 1998] extend the algorithm of [Garland and Heckbert 1997] to consider color and texture coordinate error as well as geometry. The error quadrics are lifted to higher dimensions to accommodate the combined attribute spaces (e.g. 3 dimensions for RGB color and 2 dimensions for texture coordinates). The associated form matrices grow quadratically with the dimension, but standard hardware-accelerated rendering models typically require a dimension of 9 or less. The error is thus measured and optimized for all attributes simultaneously. The method makes the simplifying assumption that the errors in all these attribute values may be measured as in a Euclidean space.

[Certain et al. 1996] present a method for preserving vertex colors in conjunction with the wavelet representation for subdivision surfaces [DeRose et al. 1993]. The geometry and color information are stored as two separate lists of wavelet coefficients. Coefficients may be added or deleted from either of these lists to adjust the complexity of the surface and its geometric and color errors. They also use the surface parameterization induced by the subdivision to store colors in texture maps to render as textured triangles for machines that support texture mapping in hardware.

[Bastos et al. 1997] use texture maps with bicubic filtering to render the complex solutions to radiosity illumination computations. The radiosity computation often dramatically increases the number of polygons in the input mesh in order to create enough vertices to store the resulting colors. Storing the colors instead in texture maps removes unnecessary geometry, reducing storing requirements and rasterization overhead.

The appearance-preserving simplification technique of [Cohen et al. 1998] is in some sense a generalization of this “radiosity as textures” work. Colors are stored as texture maps before the simplification is applied. Mappings are computed as in [Cohen et al. 1997], but this time in the 2D texture domain, effectively measuring the 3D displacements of a texture map as a surface is simplified. Whereas [Bastos et al. 1997] reduces geometry complexity to that of the pre-radiositized mesh, [Cohen et al. 1998] simplify complex geometry much farther, quantifying the distortions caused by the simplification of non-planar, textured surfaces. [Cignoni et al. 98] describe a method for compactly storing attribute values into map structures that are customized to a particular simplified mesh.

7.3 Normal Vector Preservation

[Xia et al. 1997] associate a cone of normal vectors with each vertex during their simplification preprocess. These cones initially have an angle of zero, and grow to contain the cones of the two vertices merged in an edge collapse. Their run-time, dynamic simplification scheme uses this range of normals and the light direction to compute a range of reflectance vectors. When this range includes the viewing direction, the mesh is refined, adapting the simplification to the specular highlights. The results of this approach are visually quite compelling, though they do not allow increased simplification of the highlight area as it gets smaller on the screen (i.e. as the object gets farther from the viewpoint).

[Klein 1998] maintains similar information about the cone of normal deviation associated with each vertex. The refinement criterion takes into account the spread of reflected normals (i.e. the specular exponent, or shininess) in addition to the reflectance vectors themselves. Also, refinement is performed in the neighborhood of silhouettes with respect to the light sources as well as specular highlights. Again, this normal deviation metric does not allow

increased simplification in the neighborhood of the highlights and light silhouettes as the object gets smaller on the screen.

[Cohen et al. 1998] apply their appearance-preserving technique to normals as well as colors by storing normal vectors in normal maps. Figure 11 shows a view of a complex "armadillo" model. Applying the appearance-preserving algorithm to this model generates the simplified versions of Figure 12 and Figure 13, in which it is nearly impossible to distinguish the simplifications from the original. Compared this to the bunnies in Figure 3 and Figure 4. Although the positions of the surfaces are preserved quite well, as evidenced by the similarity of the silhouettes of the bunnies, the shading makes it quite easy to tell which bunnies have been simplified and which have not (i.e. the appearance has not been totally preserved).

The appearance-preserving approach to normal preservation has the advantage that the normal values need not be considered in the simplification process – only texture distortion error constrains the simplification process. In fact, the error in the resulting images can be characterized entirely by the number of pixels of deviation of the textured surface on the screen. The major disadvantage to this approach is that it assumes a per-pixel lighting model is applied to shade the normal-mapped triangles. Per-pixel lighting is still too computationally expensive for most graphics hardware, though support for such lighting is making its way into standard graphics APIs such as OpenGL.

Figure 11: “Armadillo” model: 249,924 triangles

249,924 triangles

7,809 triangles

Figure 12: Medium-sized “armadillos”

249,924 triangles

975 triangles

Figure 13: Small-sized “armadillos”

8. CONCLUSIONS

As is the case for many classes of geometric algorithms, there does not seem to be any single best simplification algorithm or scheme. An appropriate scheme depends not only on the characteristics of the input models, but also the final application to which the multi-resolution output will be applied.

For poorly-behaved input data (mostly non-manifold or triangle soups), the vertex clustering algorithms [Rossignac and Borrel 1992], [Luebke and Erikson 1997] should yield the fastest and most painless success. For cleaner input data, one of the many methods which respect topology will likely produce more appealing results.

When even pre-computation time is of the essence, a fast algorithm such as [Garland and Heckbert 1997] may be appropriate, while applications required better-controlled visual fidelity should invest some extra pre-computation time in an algorithm such as [Cohen et al. 1998], [Guéziec 1995], or [Hoppe 1996], to achieve guaranteed or at least higher quality.

For applications and machines with extra processing power to spare, dynamic level of detail techniques such as [Hoppe 1997] and [Luebke and Erikson 1997] can provide smooth level-of-detail transitions with minimal triangle counts. However, for applications requiring maximal triangle throughput (including display lists) or need to actually employ their CPU(s) for application-related processing, static levels of detail (possibly with geomorphs between levels of detail) are often preferable (they also add less complexity to application code).

The construction and use of levels of detail have become essential tools for accelerating the rendering process. The field has now reached a level of maturity at which there is a rich “bag of tricks” from which to choose when considering the use of levels of detail for a particular application. Making sense of the available techniques as well as when and how well they work is perhaps the next step towards answering the question, “What is a good simplification?”, both statically, and over the course of an interactive application.

9. ACKNOWLEDGMENTS

We gratefully acknowledge Greg Angelini, Jim Boudreaux, and Ken Fast at the Electric Boat division of General Dynamics for the submarine model; the Stanford Computer Graphics Laboratory for the bunny and model; and Venkat Krishnamurthy and Marc Levoy at the Stanford Computer Graphics Laboratory and Peter Schröder for the "armadillo" model. Thanks to Amitabh Varshney of the State University of New York at Stonybrook for the original material for the section on optimality. Finally, thanks to Dinesh Manocha, my advisor, for the continued guidance that led to the completion of my dissertation.

10. REFERENCES

Agarwal, Pankaj K. and Subhash Suri. Surface Approximation and Geometric Partitions. *Proceedings of 5th ACM-SIAM Symposium on Discrete Algorithms.* 1994. pp. 24-33.

Bajaj, Chandrajit and Daniel Schikore. Error-bounded Reduction of Triangle Meshes with Multivariate Data. *SPIE.* vol. 2656. 1996. pp. 34-45.

Barequet, Gill and Subodh Kumar. Repairing CAD Models. *Proceedings of IEEE Visualization '97.* October 19-24. pp. 363-370, 561.

Bastos, Rui, Mike Goslin, and Hansong Zhang. Efficient Rendering of Radiosity using Texture and Bicubic Interpolation. *Proceedings of 1997 ACM Symposium on Interactive 3D Graphics.*

Brönnimann, H. and Michael T. Goodrich. Almost Optimal Set Covers in Finite VC-Dimension. *Proceedings of 10th Annual ACM Symposium on Computational Geometry.* 1994. pp. 293-302.

Certain, Andrew, Jovan Popovic, Tony DeRose, Tom Duchamp, David Salesin, and Werner Stuetzle. Interactive Multiresolution Surface Viewing. *Proceedings of SIGGRAPH 96.* pp. 91-98.

Cignoni, Paolo, Claudio Montani, Claudio Rocchini, and Roberto Scopigno. A General Method for Recovering Attribute Values on Simplified Meshes. *Proceedings of IEEE Visualization '98.* pp. 59-66, 518.

Cignoni, Paolo, Claudio Rocchini, and Roberto Scopigno. Metro: Measuring Error on Simplified Surfaces. Technical Report B4-01-01-96, Instituto I. E. I.- C.N.R., Pisa, Italy, January 1996.

Clarkson, Kenneth L. Algorithms for Polytope Covering and Approximation. *Proceedings of 3rd Workshop on Algorithms and Data Structures.* 1993. pp. 246-252.

Cohen, Jonathan, Dinesh Manocha, and Marc Olano. Simplifying Polygonal Models using Successive Mappings. *Proceedings of IEEE Visualization '97.* pp. 395-402.

Cohen, Jonathan, Marc Olano, and Dinesh Manocha. Appearance-Preserving Simplification. *Proceedings of ACM SIGGRAPH 98.* pp. 115-122.

Cohen, Jonathan, Amitabh Varshney, Dinesh Manocha, Gregory Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification Envelopes. *Proceedings of SIGGRAPH 96.* pp. 119-128.

Das, G. and D. Joseph. The Complexity of Minimum Convex Nested Polyhedra. *Proceedings of 2nd Canadian Conference on Computational Geometry.* 1990. pp. 296-301.

DeFloriani, Leila, Paola Magillo, and Enrico Puppo. Building and Traversing a Surface at Variable Resolution. *Proceedings of IEEE Visualization '97.* pp. 103-110.

DeRose, Tony, Michael Lounsbery, and J. Warren. Multiresolution Analysis for Surfaces of Arbitrary Topology Type. Technical Report TR 93-10-05. Department of Computer Science, University of Washington. 1993.

Dörrie, H. Euler's Problem of Polygon Division. *100 Great Problems of Elementary Mathematics: Their History and Solutions.* Dover, New York.1965. pp. 21-27.

Eck, Matthias, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution Analysis of Arbitrary Meshes. *Proceedings of SIGGRAPH 95.* pp. 173-182.

El-Sana, Jihad and Amitabh Varshney. Controlled Simplification of Genus for Polygonal Models. *Proceedings of IEEE Visualization'97.* pp. 403-410.

Erikson, Carl. Polygonal Simplification: An Overview. Technical Report TR96-016. Department of Computer Science, University of North Carolina at Chapel Hill. 1996.

Erikson, Carl and Dinesh Manocha. Simplification Culling of Static and Dynamic Scene Graphs. Technical Report TR98-009. Department of Computer Science, University of North Carolina at Chapel Hill. 1998.

Garland, Michael and Paul Heckbert. Simplifying Surfaces with Color and Texture using Quadric Error Metrics. *Proceedings of IEEE Visualization '98.* pp. 263-269, 542.

Garland, Michael and Paul Heckbert. Surface Simplification using Quadric Error Bounds. *Proceedings of SIGGRAPH 97.* pp. 209-216.

Gieng, Tran S., Bernd Hamann, Kenneth I. Joy, Gregory L. Schlussmann, and Isaac J. Trotts. Smooth Hierarchical Surface Triangulations. *Proceedings of IEEE Visualization '97.* pp. 379-386.

Guéziec, André. Surface Simplification with Variable Tolerance. *Proceedings of Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery (MRCAS '95).* pp. 132-139.

Hamann, Bernd. A Data Reduction Scheme for Triangulated Surfaces. *Computer Aided Geometric Design.* vol. 11. 1994. pp. 197-214.

He, Taosong, Lichan Hong, Amitabh Varshney, and Sidney Wang. Controlled Topology Simplification. *IEEE Transactions on Visualization and Computer Graphics.* vol. 2(2). 1996. pp. 171-814.

Heckbert, Paul and Michael Garland. Survey of Polygonal Simplification Algorithms. *SIGGRAPH 97 Course Notes.*1997.

Hoppe, Hugues. Progressive Meshes. *Proceedings of SIGGRAPH 96.* pp. 99-108.

Hoppe, Hugues. View-Dependent Refinement of Progressive Meshes. *Proceedings of SIGGRAPH 97.* pp. 189-198.

Hoppe, Hugues, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. *Proceedings of SIGGRAPH 93.* pp. 19-26.

Hughes, Merlin., Anselmo Lastra, and Eddie Saxe. Simplification of Global-Illumination Meshes. *Proceedings of Eurographics '96, Computer Graphics Forum.* pp. 339-345.

Kalvin, Alan D. and Russell H. Taylor. Superfaces: Polygonal Mesh Simplification with Bounded Error. *IEEE Computer Graphics and Applications.* vol. 16(3). 1996. pp. 64-77.

Klein, Reinhard. Multiresolution Representations for Surface Meshes Based on the Vertex Decimation Method. *Computers and Graphics.* vol. 22(1). 1998. pp. 13-26.

Klein, Reinhard and J. Krämer. Multiresolution Representations for Surface Meshes. *Proceedings of Spring Conference on Computer Graphics 1997.* June 5-8. pp. 57-66.

Klein, Reinhard, Gunther Liebich, and Wolfgang Straßer. Mesh Reduction with Error Control. *Proceedings of IEEE Visualization '96.*

Krishnamurthy, Venkat and Marc Levoy. Fitting Smooth Surfaces to Dense Polygon Meshes. *Proceedings of SIGGRAPH 96.* pp. 313-324.

Lindstrom, Peter and Greg Turk. Fast and Memory Efficient Polygonal Simplification. *Proceedings of IEEE Visualization '98*. pp. 279-286, 544.

Luebke, David and Carl Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. *Proceedings of SIGGRAPH 97.* pp. 199-208.

Mitchell, Joseph S. B. and Subhash Suri. Separation and Approximation of Polyhedral Surfaces. *Proceedings of 3rd ACM-SIAM Symposium on Discrete Algorithms.* 1992. pp. 296-306.

Murali, T. M. and Thomas A. Funkhouser. Consistent Solid and Boundary Representations from Arbitrary Polygonal Data. *Proceedings of 1997 Symposium on Interactive 3D Graphics.* April 27-30. pp. 155-162, 196.

O'Rourke, Joseph. *Computational Geometry in C.* Cambridge University Press 1994. 357 pages.

Plouffe, Simon and Neil James Alexander Sloan. The Encyclopedia of Integer Sequences. Academic Press 1995. pp. 587.

Rohlf, John and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real--Time 3D Graphics. *Proceedings of SIGGRAPH 94.* July 24-29. pp. 381-395.

Ronfard, Remi and Jarek Rossignac. Full-range Approximation of Triangulated Polyhedra. *Computer Graphics Forum.* vol. 15(3). 1996. pp. 67-76 and 462.

Rossignac, Jarek and Paul Borrel. Multi-Resolution 3D Approximations for Rendering. *Modeling in Computer Graphics.* Springer-Verlag1993. pp. 455-465.

Rossignac, Jarek and Paul Borrel. Multi-Resolution 3D Approximations for Rendering Complex Scenes. Technical Report RC 17687-77951. IBM Research Division, T. J. Watson Research Center. Yorktown Heights, NY 10958. 1992.

Schikore, Daniel and Chandrajit Bajaj. Decimation of 2D Scalar Data with Error Control. Technical Report CSD-TR-95-004. Department of Computer Science, Purdue University. 1995.

Schroeder, William J., Jonathan A. Zarge, and William E. Lorensen. Decimation of Triangle Meshes. *Proceedings of SIGGRAPH 92.* pp. 65-70.

Turk, Greg. Re-tiling Polygonal Surfaces. *Proceedings of SIGGRAPH 92.* pp. 55-64.

van Dam, Andries. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics.* vol. 22(3). 1988. pp. 125-218.

Varshney, Amitabh. Hierarchical Geometric Approximations. Ph.D. Thesis. Department of Computer Science. University of North Carolina at Chapel Hill. 1994.

Xia, Julie C., Jihad El-Sana, and Amitabh Varshney. Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models. *IEEE Transactions on Visualization and Computer Graphics.* vol. 3(2). 1997. pp. 171-183.a

Appearance-Preserving Simplification

Jonathan Cohen Marc Olano Dinesh Manocha

University of North Carolina at Chapel Hill

Abstract

We present a new algorithm for appearance-preserving simplification. Not only does it generate a low-polygon-count approximation of a model, but it also preserves the appearance. This is accomplished for a particular display resolution in the sense that we properly sample the surface position, curvature, and color attributes of the input surface. We convert the input surface to a representation that decouples the sampling of these three attributes, storing the colors and normals in texture and normal maps, respectively. Our simplification algorithm employs a new *texture deviation metric*, which guarantees that these maps shift by no more than a user-specified number of pixels on the screen. The simplification process filters the surface position, while the run-time system filters the colors and normals on a per-pixel basis. We have applied our simplification technique to several large models achieving significant amounts of simplification with little or no loss in rendering quality.

CR Categories: I.3.5: Object hierarchies, I.3.7: Color, shading, shadowing, and texture

Additional Keywords: simplification, attributes, parameterization, color, normal, texture, maps

1 INTRODUCTION

Simplification of polygonal surfaces has been an active area of research in computer graphics. The main goal of simplification is to generate a low-polygon-count approximation that maintains the high fidelity of the original model. This involves preserving the model's main features and overall appearance. Typically, there are three *appearance attributes* that contribute to the overall appearance of a polygonal surface:

1. **Surface position**, represented by the coordinates of the polygon vertices.
2. **Surface curvature**, represented by a field of normal vectors across the polygons.
3. **Surface color**, also represented as a field across the polygons.

The number of samples necessary to represent a surface accurately depends on the nature of the model and its area in screen pixels (which is related to its distance from the viewpoint). For a simplification algorithm to preserve the appearance of the input surface, it must guarantee adequate sampling of these three attributes. If it does, we say that it has preserved the appearance with respect to the display resolution.

e-mail: {cohenj,dm}@cs.unc.edu, olano@engr.sgi.com
WWW: http://www.cs.unc.edu/~geom/APS

The majority of work in the field of simplification has focused on *surface approximation* algorithms. These algorithms bound the error in surface position only. Such bounds can be used to guarantee a maximum deviation of the object's silhouette in units of pixels on the screen. While this guarantees that the object will cover the correct pixels on the screen, it says nothing about the final colors of these pixels.

Of the few simplification algorithms that deal with the remaining two attributes, most provide some threshold on a maximum or average deviation of these attribute values across the model. While such measures do guarantee adequate sampling of all three attributes, they do *not* generally allow increased simplification as the object becomes smaller on the screen. These threshold metrics do not incorporate information about the object's distance from the viewpoint or its area on the screen. As a result of these metrics and of the way we typically represent these appearance attributes, simplification algorithms have been quite restricted in their ability to simplify a surface while preserving its appearance.

1.1 Main Contribution

We present a new algorithm for appearance-preserving simplification. We convert our input surface to a *decoupled representation*. Surface position is represented in the typical way, by a set of triangles with 3D coordinates stored at the vertices. Surface colors and normals are stored in texture and normal maps, respectively. These colors and normals are mapped to the surface with the aid of a surface parameterization, represented as 2D texture coordinates at the triangle vertices.

The surface position is filtered using a standard surface approximation algorithm that makes local, complexity-reducing simplification operations (e.g. edge collapse, vertex removal, etc.). The color and normal attributes are filtered by the run-time system at the pixel level, using standard mip-mapping techniques [1].

Because the colors and normals are now decoupled from the surface position, we employ a new *texture deviation metric,* which effectively bounds the deviation of a mapped attribute value's position from its correct position on the original surface. We thus guarantee that each attribute is appropriately sampled and mapped to screen-space. The deviation metric necessarily constrains the simplification algorithm somewhat, but it is much less restrictive than retaining sufficient tessellation to accurately represent colors and normals in a standard, per-vertex representation. The preservation of colors using texture maps is possible on all current graphics systems that supports real-time texture maps. The preservation of normals using normal maps is possible on prototype machines today, and there are indications that hardware

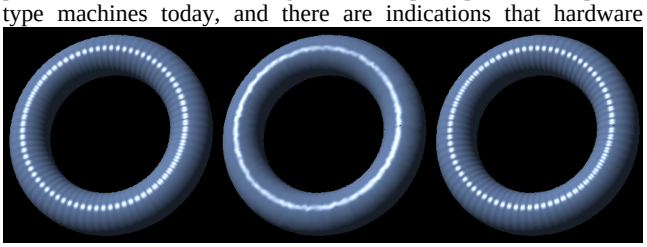


Figure 1: Bumpy Torus Model. *Left*: 44,252 triangles full resolution mesh. *Middle and Right*: 5,531 triangles, 0.25 mm maximum image deviation. *Middle*: per-vertex normals. *Right*: normal maps

support for real-time normal maps will become more widespread in the next several years.

One of the nice properties of this approach is that the user-specified error tolerance, ε, is both simple and intuitive; it is a screen-space deviation in pixel units. A particular point on the surface, with some color and some normal, may appear to shift by at most ε pixels on the screen.

We have applied our algorithm to several large models. Figure 1 clearly shows the improved quality of our appearance-preserving simplification technique over a standard surface approximation algorithm with per-vertex normals. By merely controlling the switching distances properly, we can discretely switch between a few statically-generated levels of detail (sampled from a progressive mesh representation) with no perceptible artifacts. Overall, we are able to achieve a significant speedup in rendering large models with little or no loss in rendering quality.

1.2 Paper Organization

In Section 2, we review the related work from several areas. Section 3 presents an overview of our appearance-preserving simplification algorithm. Sections 4 through 6 describe the components of this algorithm, followed by a discussion of our particular implementation and results in Section 7. Finally, we mention our ongoing work and conclude in Section 8.

2 RELATED WORK

Research areas related to this paper include geometric levels-of-detail, preservation of appearance attributes, and map-based representations. We now briefly survey these.

2.1 Geometric Levels-Of-Detail

Given a polygonal model, a number of algorithms have been proposed for generating levels-of-detail. These methods differ according to the local or global error metrics used for simplification and the underlying data structures or representations. Some approaches based on vertex clustering [2, 3] are applicable to all polygonal models and do not preserve the topology of the original models. Other algorithms assume that the input model is a valid mesh. Algorithms based on vertex removal [4, 5] and local error metrics have been proposed by [6-10]. Cohen et al. [11] and Eck et al. [12] have presented algorithms that preserve topology and use a global error bound. Our appearance-preserving simplification algorithm can be combined with many of these.

Other simplification algorithms include decimation techniques based on vertex removal [4], topology modification [13], and controlled simplification of genus [14]. All of these algorithms compute static levels-of-detail. Hoppe [15] has introduced an incremental representation, called the *progressive mesh*, and based on that representation view-dependent algorithms have been proposed by [16, 17]. These algorithms use different view-dependent criteria like local illumination, screen-space surface approximation error, and silhouette edges to adaptively refine the meshes. Our appearance preserving simplification algorithm generates a progressive mesh, which can be used by these view-dependent algorithms.

2.2 Preserving Appearance Attributes

Bajaj and Schikore [18] have presented an algorithm to simplify meshes with associated scalar fields to within a given tolerance. Hughes et al. [19] have presented an algorithm to simplify radiositized meshes. Erikson and Manocha[20] grow error volumes for appearance attributes as well as geometry. Many algorithms based on multi-resolution analysis have been proposed as well. Schroeder and Sweldens [21] have presented algorithms for simplifying functions defined over a sphere. Eck et al. [12] apply multi-resolution analysis to simplify arbitrary meshes, and Certain et al. [22] extend this to colored meshes by separately analyzing surface geometry and color. They make use of texture mapping hardware to render the color at full resolution. It may be possible to extend this approach to handle other functions on the mesh. However, algorithms based on vertex removal and edge collapses [11, 15] have been able to obtain more drastic simplification (in terms of reducing the polygon count) and produce better looking simplifications [15].

Hoppe [15] has used an optimization framework to preserve discrete and scalar surface appearance attributes. Currently, this algorithm measures a maximum or average deviation of the scalar attributes across the model. Our approach can be incorporated into this comprehensive optimization framework to preserve the appearance of colors and normals, while allowing continued simplification as an object's screen size is reduced.

2.3 Map-based Representations

Texture mapping is a common technique for defining color on a surface. It is just one instance of mapping, a general technique for defining attributes on a surface. Other forms of mapping use the same texture coordinate parameterization, but with maps that contain something other than surface color. *Displacement maps* [23] contain perturbations of the surface position. They are typically used to add surface detail to a simple model. *Bump maps* [24] are similar, but instead give perturbations of the surface normal. They can make a smooth surface appear bumpy, but will not change the surface's silhouette. *Normal maps* [25] can also make a smooth surface appear bumpy, but contain the actual normal instead of just a perturbation of the normal.

Texture mapping is available in most current graphics systems, including workstations and PCs. We expect to see bump mapping and similar surface shading techniques on graphics systems in the near future [26]. In fact, many of these mapping techniques are already possible using the procedural shading capabilities of PixelFlow[27].

Several researchers have explored the possibility of replacing geometric information with texture. Kajiya first introduced the "hierarchy of scale" of geometric models, mapping, and lighting[28]. Cabral et. al. [29] addressed the transition between bump mapping and lighting effects. Westin et. al. [30] generated BRDFs from a Monte-Carlo ray tracing of an idealized piece of surface. Becker and Max [31] handle transitions from geometric detail in the form of displacement maps to shading in the form of bump maps. Fournier [25] generates maps with normal and shading information directly from surface geometry. Krishnamurthy and Levoy [32] fit complex, scanned surfaces with a set of smooth B-spline patches, then store some of the lost geometric information in a displacement map or bump map. Many algorithms first capture the geometric complexity of a scene in an image-based representation by rendering several different views and then render the scene using texture maps [33-36].

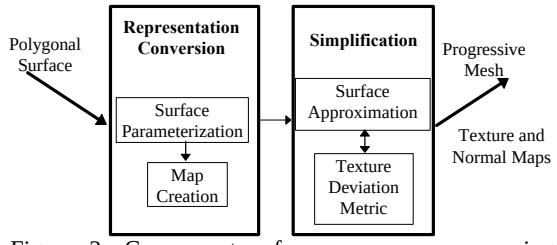


Figure 2: Components of an appearance-preserving simplification system.

3 OVERVIEW

We now present an overview of our appearance-preserving simplification algorithm. Figure 2 presents a breakdown of the algorithm into its components. The input to the algorithm is the polygonal surface, $\mathbb{M}_0$, to be simplified. The surface may come from one of a wide variety of sources, and thus may have a variety of characteristics. The types of possible input models include:

- **CAD models**, with per-vertex normals and a single color
- **Radiositized models**, with per-vertex colors and no normals
- **Scientific visualization models**, with per-vertex normals and per-vertex colors
- **Textured models**, with texture-mapped colors, with or without per-vertex normals

To store the colors and normals in maps, we need a parameterization of the surface, $\mathbf{F}_0(\mathbf{X})$: $\mathbb{M}_0 \rightarrow \mathbb{P}$, where $\mathbb{P}$ is a 2D texture domain (*texture plane*), as shown in Figure 3. If the input model is already textured, such a parameterization comes with the model. Otherwise, we create one and store it in the form of per-vertex texture coordinates. Using this parameterization, per-vertex colors and normals are then stored in texture and normal maps.

The original surface and its texture coordinates are then fed to the surface simplification algorithm. This algorithm is responsible for choosing which simplification operations to perform and in what order. It calls our texture deviation component to measure the deviation of the texture coordinates caused by each proposed operation. It uses the resulting error bound to help make its choices of operations, and stores the bound with each operation in its progressive mesh output.

We can use the resulting progressive mesh with error bounds to create a static set of levels of detail with error bounds, or we can use the progressive mesh directly with a view-dependent simplification system at run-time. Either way, the error bound allows the run-time system to choose or adjust the tessellation of the models to meet a user-specified tolerance. It is also possible for the user to choose a desired polygon count and have the run-time system increase or decrease the error bound to meet that target.

4 REPRESENTATION CONVERSION

Before we apply the actual simplification component of our algorithm, we perform a representation conversion (as shown in Figure 2). The representation we choose for our surface has a significant impact on the amount of simplification we can perform for a given level of visual fidelity. To convert to a form which decouples the sampling rates of the colors and normals from the sampling rate of the surface, we first parameterize the surface, then store the color and normal information in separate maps.

4.1 Surface Parameterization

To store a surface's color or normal attributes in a map, the surface must first have a 2D parameterization. This function, $\mathbf{F}_0(\mathbf{X})$: $\mathbb{M}_0 \rightarrow \mathbb{P}$, maps points, $\mathbf{X}$, on the input surface, $\mathbb{M}_0$, to points, $\mathbf{x}$,* on the texture plane, $\mathbb{P}$ (see Figure 3). The surface is typically decomposed into several *polygonal patches*, each with its own parameterization. The creation of such parameterizations has been an active area of research and is fundamental for shape transformation, multi-resolution analysis, approximation of meshes by NURBS, and texture mapping. Though we do not present a new algorithm for such parameterization here, it is useful to consider

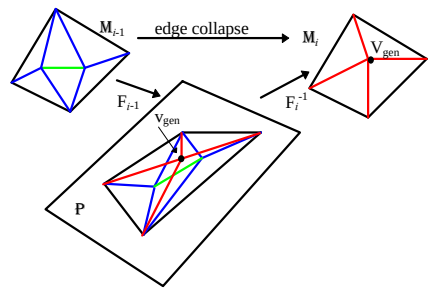


Figure 3: A look at the *i*th edge collapse. Computing $\mathbf{V}_{gen}$ determines the shape of the new mesh, $\mathbb{M}_i$. Computing $\mathbf{v}_{gen}$ determines the new mapping $\mathbf{F}_i$, to the texture plane, $\mathbb{P}$.

the desirable properties of such a parameterization for our algorithm. They are:

1. **Number of patches**: The parameterization should use as few patches as possible. The triangles of the simplified surface must each lie in a single patch, so the number of patches places a bound on the minimum mesh complexity.
2. **Vertex distribution**: The vertices should be as evenly distributed in the texture plane as possible. If the parameterization causes too much area compression, we will require a greater map resolution to capture all of our original per-vertex data.
3. **One-to-one mapping**: The mapping from the surface to the texture plane should be one-to-one. If the surface has folds in the texture plane, parts of the texture will be incorrectly stored and mapped back to the surface

Our particular application of the parameterization makes us somewhat less concerned with preserving aspect ratios than some other applications are. For instance, many applications apply $\mathbf{F}^{-1}(\mathbf{x})$ to map a pre-synthesized texture map to an arbitrary surface. In that case, distortions in the parameterization cause the texture to look distorted when applied to the surface. However, in our application, the color or normal data originates on the surface itself. Any distortion created by applying $\mathbf{F}(\mathbf{X})$ to map this data onto $\mathbb{P}$ is reversed when we apply $\mathbf{F}^{-1}(\mathbf{x})$ to map it back to $\mathbb{M}$.

Algorithms for computing such parameterizations have been studied in the computer graphics and graph drawing literature.

Computer Graphics: In the recent computer graphics literature, [12, 37, 38] use a spring system with various energy terms to distribute the vertices of a polygonal patch in the plane. [12, 32, 38, 39] provide methods for subdividing surfaces into separate patches based on automatic criteria or user-guidance. This body of research addresses the above properties one and two, but unfortunately, parameterizations based on spring-system algorithms do not generally guarantee a one-to-one mapping.

Graph Drawing: The field of graph drawing addresses the issue of one-to-one mappings more rigorously. Relevant topics include straight-line drawings on a grid [40] and convex straight-line drawings [41]. Battista et al. [42] present a survey of the field. These techniques produce guaranteed one-to-one mappings, but the necessary grids for a graph with V vertices are worst case (and typically) O(V) width and height, and the vertices are generally unevenly spaced.

To break a surface into polygonal patches, we currently apply an automatic subdivision algorithm like that presented in [12]. Their application requires a patch network with more constraints than ours. We can generally subdivide the surface into fewer patches. During this process, which grows Voronoi-like patches, we simply require that each patch not expand far enough to touch itself. To produce the parameterization for each patch, we employ

* Capital letters (e.g. **X**) refer to points in 3D, while lowercase letters (e.g. **x**) refer to points in 2D.

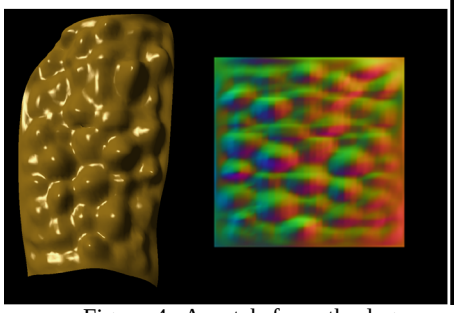


Figure 4: A patch from the leg of an armadillo model and its associated normal map.

Figure 5: Lion model.

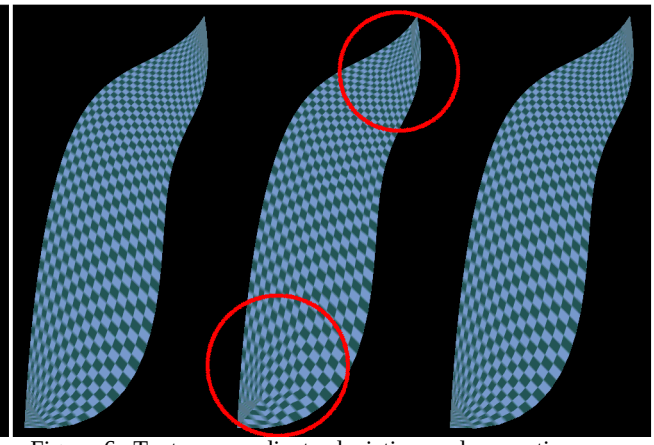


Figure 6: Texture coordinate deviation and correction on the lion's tail. *Left*: 1,740 triangles full resolution. *Middle and Right*: 0.25 mm maximum image deviation. *Middle*: 108 triangles, no texture deviation metric. *Right*: 434 triangles with texture metric.

a spring system with uniform weights. A side-by-side comparison of various choices of weights in [12] shows that uniform weights produce more evenly-distributed vertices than some other choices. For parameterizations used only with one particular map, it is also possible to allow more area compression where data values are similar. While this technique will generally create reasonable parameterizations, it would be better if there were a way to *also* guarantee that $\mathbf{F}(\mathbf{X})$ is one-to-one, as in the graph drawing literature.

4.2 Creating Texture and Normal Maps

Given a polygonal surface patch, $\mathbb{M}_0$, and its 2D parameterization, $\mathbf{F}$, it is straightforward to store per-vertex colors and normals into the appropriate maps using standard rendering software. To create a map, scan convert each triangle of $\mathbb{M}_0$, replacing each of its vertex coordinates, $\mathbf{V}_j$, with $\mathbf{F}(\mathbf{V}_j)$, the texture coordinates of the vertex. For a texture map, apply the Gouraud method for linearly interpolating the colors across the triangles. For a normal map, interpolate the per-vertex normals across the triangles instead (Figure 4).

The most important question in creating these maps is what the maximum resolution of the map images should be. To capture all the information from the original mesh, each vertex's data should be stored in a unique texel. We can guarantee this conservatively by choosing $1/d$ x $1/d$ for our map resolution, where d is the minimum distance between vertex texture coordinates:

$$d = \min_{\mathbf{V}_i, \mathbf{V}_j \in \mathbb{M}_0, i \neq j} \left\| \mathbf{F}(\mathbf{V}_i) - \mathbf{F}(\mathbf{V}_j) \right\| \qquad (1)$$

If the vertices of the polygonal surface patch happen to be a uniform sampling of the texture space (e.g. if the polygonal surface patch was generated from a parametric curved surface patch), then the issues of scan conversion and resolution are simplified considerably. Each vertex color (or normal) is simply stored in an element of a 2D array of the appropriate dimensions, and the array itself is the map image.

It is possible to trade off accuracy of the map data for run-time texturing resources by scaling down the initial maps to a lower resolution.

5 SIMPLIFICATION ALGORITHM

Once we have decomposed the surface into one or more parameterized polygonal patches with associated maps, we begin the actual simplification process. Many simplification algorithms perform a series of edge collapses or other local simplification operations to gradually reduce the complexity of the input surface. The order in which these operations are performed has a large impact on the quality of the resulting surface, so simplification algorithms typically choose the operations in order of increasing error according to some metric. This metric may be local or global in nature, and for surface approximation algorithms, it provides some bound or estimate on the error in surface position. The operations to be performed are typically maintained in a priority queue, which is continually updated as the simplification progresses. This basic design is applied by many of the current simplification algorithms, including [6-8, 15].

To incorporate our appearance-preservation approach into such an algorithm, the original algorithm is modified to use our texture deviation metric in addition to its usual error metric. When an edge is collapsed, the error metric of the particular surface approximation algorithm is used to compute a value for $\mathbf{V}_{gen}$, the surface position of the new vertex (see Figure 3). Our texture deviation metric is then applied to compute a value for $\mathbf{v}_{gen}$, the texture coordinates of the new vertex.

For the purpose of computing an edge's priority, there are several ways to combine the error metrics of surface approximation along with the texture deviation metric, and the appropriate choice depends on the algorithm in question. Several possibilities for such a *total error metric* include a weighted combination of the two error metrics, the maximum or minimum of the error metrics, or one of the two error metrics taken alone. For instance, when integrating with Garland and Heckbert's algorithm [6], it would be desirable to take a weighted combination in order to retain the precedence their system accords the topology-preserving collapses over the topology-modifying collapses. Similarly, a weighted combination may be desirable for an integration with Hoppe's system [15], which already optimizes error terms corresponding to various mesh attributes.

The interactive display system later uses the error metrics to determine appropriate distances from the viewpoint either for switching between static levels of detail or for collapsing/splitting the edges dynamically to produce adaptive, view-dependent tessellations. If the system intends to guarantee that certain tolerances are met, the maximum of the error metrics is often an appropriate choice.

6 TEXTURE DEVIATION METRIC

A key element of our approach to appearance-preservation is the measurement of the *texture coordinate deviation* caused by the simplification process. We provide a bound on this deviation, to

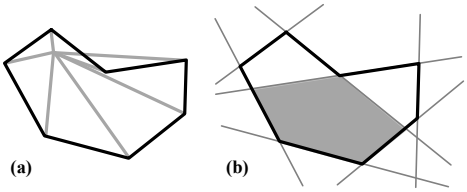


Figure 7: (a) An invalid choice for $\mathbf{v}_{gen}$ in $\mathbb{P}$, causing the new triangles extend outside the polygon. (b) Valid choices must lie in the shaded *kernel*.

be used by the simplification algorithm to prioritize the potential edge collapses and by the run-time visualization system to choose appropriate levels of detail based on the current viewpoint. The lion's tail in Figure 6 demonstrates the need to measure texture coordinate deviation. The center figure is simplified by a surface approximation algorithm without using a texture deviation metric. The distortions are visible in the areas marked by red circles. The right tail is simplified using our texture deviation metric and does not have visible distortions. The image-space deviation bound now applies to the texture as well as to the surface.

For a given point, $\mathbf{X}$, on simplified mesh $\mathbb{M}_i$, this deviation is the distance in 3D from $\mathbf{X}$ to the point on the input surface with the same texture coordinates:

$$\mathrm{T}_i(\mathbf{X}) = \left\| \mathbf{X} - \mathbf{F}_0^{-1}(\mathbf{F}_i(\mathbf{X})) \right\| \tag{2}$$

We define the texture coordinate deviation of a whole triangle to be the maximum deviation of all the points in the triangle, and similarly for the whole surface:

$$\mathrm{T}_i(\Delta) = \max_{\mathbf{X}\in\Delta} \mathrm{T}_i(\mathbf{X}); \quad \mathrm{T}_i(\mathbb{M}_i) = \max_{\mathbf{X}\in\mathbb{M}_i} \mathrm{T}_i(\mathbf{X}) \tag{3}$$

To compute the texture coordinate deviation incurred by an edge collapse operation, our algorithm takes as input the set of triangles before the edge collapse and $\mathbf{V}_{gen,}$ the 3D coordinates of the new vertex generated by the collapse operation. The algorithm outputs $\mathbf{v}_{gen}$, the 2D texture coordinates for this generated vertex, and a bound on $\mathrm{T}_i(\Delta)$ for each of the triangles after the collapse.

6.1 Computing New Texture Coordinates

We visualize the neighborhood of an edge to be collapsed in the texture plane, $\mathbb{P}$, as shown in Figure 3. The boundary of the edge neighborhood is a polygon in $\mathbb{P}$. The edge collapse causes us to replace the two vertices of the edge with a single vertex. The 3D coordinates, $\mathbf{V}_{gen}$ of this generated vertex are provided to us by the surface approximation algorithm. The first task of the texture deviation algorithm is to compute $\mathbf{v}_{gen}$, the 2D texture coordinates of this generated vertex.

For $\mathbf{v}_{gen}$ to be valid, it must lie in the convex *kernel* of our polygon in the texture plane [43] (see Figure 7). Meeting this criterion ensures that the set of triangles after the edge collapse covers exactly the same portion of the texture plane as the set of triangles before the collapse.

Given a candidate point in the texture plane, we efficiently test the kernel criterion with a series of dot products to see if it lies on the inward side of each polygon edge. We first test some heuristic choices for the texture coordinates – the midpoint of the original edge in the texture plane or one of the edge vertices. If the heuristic choices fail we compute a point inside the kernel by averaging three corners, found using linear programming techniques [43].

6.2 Patch Borders & Continuity

Unlike an interior edge collapse, an edge collapse on a patch border can change the coverage in the texture plane, either by cutting off some of texture space or by extending into a portion of texture space for which we have no map data. Since neither of

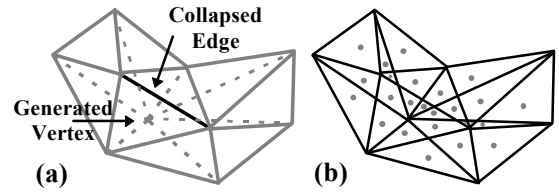


Figure 8: (a) An overlay in $\mathbb{P}$ determines the mapping between $\mathbb{M}_{i-1}$ and $\mathbb{M}_i$. (b) A set of polygonal *mapping cells*, each containing a dot.

these is acceptable, we add additional constraints on the choice of $\mathbf{v}_{gen}$ at patch borders.

We assume that the area of texture space for which we have map data is rectangular (though the method works for any map that covers a polygonal area in texture space), and that the edges of the patch are also the edges of the map. If the entire edge to be collapsed lies on a border of the map, we restrict $\mathbf{v}_{gen}$ to lie on the edge. If one of the vertices of the edge lies on a corner of the map, we further restrict $\mathbf{v}_{gen}$ to lie at that vertex. If only one vertex is on the border, we restrict $\mathbf{v}_{gen}$ to lie at that vertex. If one vertex of the edge lies on one border of the map and the other vertex lies on a different border, we do not allow the edge collapse.

The surface parameterization component typically breaks the input model into several connected patches. To preserve geometric and texture continuity across the boundary between them, we add further restrictions on the simplifications that are performed along the border. The shared border edges must be simplified on both patches, with matching choices of $\mathbf{V}_{gen}$ and $\mathbf{v}_{gen}$.

6.3 Measuring Texture Deviation

Texture deviation is a measure of the parametric distortion caused by the simplification process. We measure this deviation using a method similar to the one presented to measure surface deviation in [8]. The main difference is that we now measure the deviation using our mapping in the texture plane, rather than in the plane of some planar projection. While [8] presents an overview of this technique, we present it more formally.

Given the overlay (see Figure 8(a)) in the texture plane, $\mathbb{P}$, of two simplified versions of the surface, $\mathbb{M}_i$ and $\mathbb{M}_j$, we define the *incremental texture deviation* between them:

$$\mathbf{E}_{i,j}(\mathbf{x}) = \left\| \mathbf{F}_i^{-1}(\mathbf{x}) - \mathbf{F}_j^{-1}(\mathbf{x}) \right\| \tag{4}$$

This is the deviation between corresponding 3D points on the surfaces, both with texture coordinates, $\mathbf{x}$. Between any two sequential surfaces, $\mathbb{M}_i$ and $\mathbb{M}_{i-1}$, differing only by an edge collapse, the incremental deviation, $\mathbf{E}_{i,i-1}(\mathbf{x})$, is only non-zero in the neighborhood of the collapsed edge (i.e. only in the triangles that actually move).

The edges on the overlay in $\mathbb{P}$ partition the region into a set of convex, polygonal *mapping cells* (each identified by a dot in Figure 8(b)). Within each mapping cell, the incremental deviation function is linear, so the maximum incremental deviation for each cell occurs at one of its boundary points. Thus, we bound the incremental deviation using only the deviation at the cell vertices, $\mathbf{v}_k$:

$$\mathbf{E}_{i,i-1}(\mathbb{P}) = \max_{\mathbf{x}\in\mathbb{P}} \mathbf{E}_{i,i-1}(\mathbf{x}) = \max_{\mathbf{v}_k} \mathbf{E}_{i,i-1}(\mathbf{v}_k) \tag{5}$$

In terms of the incremental deviation, the *total texture deviation*, defined in (2) (the distance from points on $\mathbb{M}_i$ to corresponding points on the original surface, $\mathbb{M}_0$) is

$$\mathbf{T}_i(\mathbf{X}) = \mathbf{E}_{i,0}\left(\mathbf{F}_i(\mathbf{X})\right) \tag{6}$$

We approximate $\mathbf{E}_{i,0}(\mathbf{x})$ using a set of axis-aligned boxes. This provides a convenient representation of a bound on $\mathbf{T}_i(\mathbf{X})$, which

we can update from one simplified mesh to the next without having to refer to the original mesh. Each triangle, Δ_k in $\mathbf{M}_i$, has its own axis-aligned box, $\boldsymbol{b}_{i,k}$ such that at every point on the triangle, the Minkowski sum of the 3D point with the box gives a region that contains the point on the original surface with the same texture coordinates.

$$\forall \mathbf{X} \in \Delta_k,\ \mathbf{F}_0^{-1}\left(\mathbf{F}_i(\mathbf{X})\right) \in \mathbf{X} \oplus \boldsymbol{b}_{i,k} \tag{7}$$

Figure 9(a) shows an original surface (curve) in black and a simplification of it, consisting of the thick blue and green lines. The box associated with the blue line, $\boldsymbol{b}_{i,0}$, is shown in blue, while the box for the green line, $\boldsymbol{b}_{i,1}$, is shown in green. The blue box slides along the blue line; at every point of application, the point on the base mesh with the same texture coordinate is contained within the translated box. For example, one set of corresponding points is shown in red, with its box also in red.

From (2) and (7), we produce $\tilde{\mathrm{T}}_i(\mathbf{X})$, a bound on the total texture deviation, $\mathrm{T}_i(\mathbf{X})$. This our texture deviation output.

$$\mathrm{T}_i(\mathbf{X}) \le \tilde{\mathrm{T}}_i(\mathbf{X}) = \max_{\mathbf{X}' \in \mathbf{X} \oplus \boldsymbol{b}_{i,j}} \left\| \mathbf{X} - \mathbf{X}' \right\| \tag{8}$$

$\tilde{\mathrm{T}}_i(\mathbf{X})$ is the distance from $\mathbf{X}$ to the farthest corner of the box at $\mathbf{X}$. This will always bound the distance from $\mathbf{X}$ to $\mathbf{F}_0^{-1}(\mathbf{F}_i(\mathbf{X}))$. The maximum deviation over an edge collapse neighborhood is the maximum $\tilde{\mathrm{T}}_i(\mathbf{X})$ for any cell vertex.

The boxes, $\boldsymbol{b}_{i,k}$, are the only information we keep about the position of the original mesh as we simplify. We create a new set of boxes, $\boldsymbol{b}_{i+1,k}$, for mesh $\mathbf{M}_{i+1}$ using an incremental computation (described in Figure 10). Figure 9(b) shows the propagation from $\mathbf{M}_i$ to $\mathbf{M}_{i+1}$. The blue and green lines are simplified to the pink line. The new box, $\boldsymbol{b}_{i+1,0}$ is constant as it slides across the pink line. The size and offset is chosen so that, at every point of application, the pink box, $\boldsymbol{b}_{i+1,0}$, contains the corresponding blue or green boxes, $\boldsymbol{b}_{i,0}$ or $\boldsymbol{b}_{i,1}$.

If $\mathbf{X}$ is a point on $\mathbf{M}_i$ in triangle k, and $\mathbf{Y}$ is the point with the same texture coordinate on $\mathbf{M}_{i+1}$, the containment property of (7) holds:

$$\mathbf{F}_0^{-1}\left(\mathbf{F}_{i+1}(\mathbf{Y})\right) \in \mathbf{X} \oplus \boldsymbol{b}_{i,k} \subseteq \mathbf{Y} \oplus \boldsymbol{b}_{i+1,k'} \tag{9}$$

For example, all three red dots Figure 9(b) have the same texture coordinates. The red point on $\mathbf{M}_o$ is contained in the smaller red box, $\mathbf{X} \oplus \boldsymbol{b}_{i,0}$, which is contained in the larger red box, $\mathbf{Y} \oplus \boldsymbol{b}_{i+1,0}$.

Because each mapping cell in the overlay between $\mathbf{M}_i$ and $\mathbf{M}_{i+1}$ is linear, we compute the sizes of the boxes, $\boldsymbol{b}_{i+1,k'}$, by considering

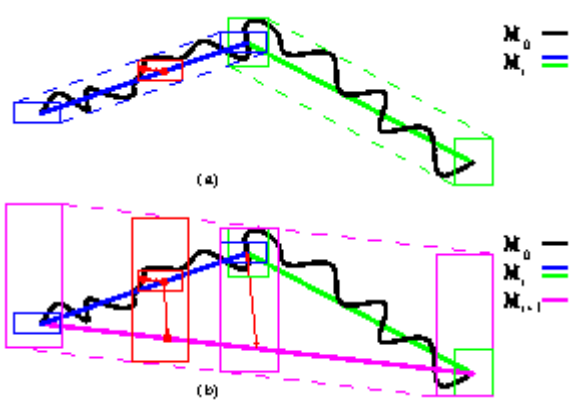


Figure 9: 2D illustration of the box approximation to total deviation error. a) A curve has been simplified to two segment, each with an associated box to bound the deviation. b) As we simplify one more step, the approximation is propagated to the newly created segment.

```
PropagateError():
foreach cell vertex, v
    foreach triangle, Told, in Mi-1 touching v
        foreach triangle, Tnew, in Mi touching v
            PropagateBox(v, Told, Tnew)

PropagateBox(v, Told, Tnew):
Pold = Fi-1^-1(v), Pnew = Fi^-1(v)
Enlarge Told.box so that Told.box applied at
    Pold contains Tnew.box applied at Pnew
```

Figure 10: Pseudo-code for the propagation of deviation error from mesh M_{i-1} to mesh M_i.

only the box correspondences at cell vertices. In Figure 9(b), there are three places we must consider. If the magenta box contains the blue and green boxes in all three places, it will contain them everywhere.

Together, the propagation rules, which are simple to implement, and the box-based approximation to the texture deviation, provide the tools we need to efficiently provide a texture deviation for the simplification process.

7 IMPLEMENTATION AND RESULTS

In this section we present some details of our implementation of the various components of our appearance-preserving simplification algorithm. These include methods for representation conversion, simplification and, finally, interactive display.

7.1 Representation Conversion

We have applied our technique to several large models, including those listed in Table 1. The bumpy torus model (Figure 1) was created from a parametric equation to demonstrate the need for greater sampling of the normals than of the surface position. The lion model (Figure 5) was designed from NURBS patches as part of a much larger garden environment, and we chose to decorate it with a marble texture (and a checkerboard texture to make texture deviation more apparent in static images). Neither of these models required the computation of a parameterization. The armadillo (Figure 12) was constructed by merging several laser-scanned meshes into a single, dense polygon mesh. It was decomposed into polygonal patches and parameterized using the algorithm presented in [32], which eventually converts the patches into a NURBS representation with associated displacement maps.

Because all these models were not only parameterized, but available in piecewise-rational parametric representations, we generated polygonal patches by uniformly sampling these representations in the parameter space. We chose the original tessellation of the models to be high enough to capture all the detail available in their smooth representations. Due to the uniform sampling, we were able to use the simpler method of map creation (described in Section 4.2), avoiding the need for a scan-conversion process.

7.2 Simplification

We integrated our texture deviation metric with the successive mapping algorithm for surface approximation [8]. The error metric for the successive mapping algorithm is simply a 3D surface deviation. We used this deviation only in the computation of $\mathbf{V}_{gen}$. Our total error metric for prioritizing edges and choosing switching distances is just the texture deviation. This is sensible because the texture deviation metric is also a measure of surface deviation, whose particular mapping is the parameterization. Thus, if the successive mapping metric is less than the texture deviation metric, we must apply the texture deviation metric, because it is the minimum bound we know that guarantees the bound on our texture deviation. On the other hand, if the successive mapping metric is greater than the texture deviation metric,

the texture deviation bound is still sufficient to guarantee a bound on both the surface deviation and the texture.

To achieve a simple and efficient run-time system, we apply a post-process to convert the progressive mesh output to a static set of levels of detail, reducing the mesh complexity by a factor of two at each level.

Our implementation can either treat each patch as an independent object or treat a connected set of patches as one object. If we simplify the patches independently, we have the freedom to switch their levels of detail independently, but we will see cracks between the patches when they are rendered at a sufficiently large error tolerance. Simplifying the patches together allows us to prevent cracks by switching the levels of detail simultaneously.

Table 1 gives the computation time to simplify several models,

Figure 11: Levels of detail of the armadillo model shown with 1.0 mm maximum image deviation. Triangle counts are: 7,809, 3,905, 1,951, 975, 488

Model	Patches	Input Tris	Time	Map Res.
Torus	1	44,252	4.4	512x128
Lion	49	86,844	7.4	N.A.
Armadillo	102	2,040,000	190	128x128

Table 1: Several models used to test appearance-preserving simplification. Simplification time is in minutes on a MIPS R10000 processor.

as well as the resolution of each map image. Figure 11 and Figure 12 show results on the armadillo model. It should be noted that the latter figure is not intended to imply equal computational costs for rendering models with per-vertex normals and normal maps. Simplification using the normal map representation provides measurable quality and reduced triangle overhead, with an additional overhead dependent on the screen resolution.

7.3 Interactive Display System

We have implemented two interactive display systems: one on top of SGI's IRIS Performer library, and one on top of a custom library running on a PixelFlow system. The SGI system supports color preservation using texture maps, and the PixelFlow system supports color and normal preservation using texture and normal maps, respectively. Both systems apply a bound on the distance from the viewpoint to the object to convert the texture deviation error in 3D to a number of pixels on the screen, and allow the user to specify a tolerance for the number of pixels of deviation. The tolerance is ultimately used to choose the primitives to render from among the statically generated set of levels of detail.

Our custom shading function on the PixelFlow implementation performs a mip-mapped look-up of the normal and applies a

249,924 triangles	62,480 triangles	7,809 triangles	975 triangles
0.05 mm max image deviation	0.25 mm max image deviation	1.3 mm max image deviation	6.6 mm max image deviation

Figure 12: Close-up of several levels of detail of the armadillo model. *Top*: normal maps *Bottom*: per-vertex normals

Phong lighting model to compute the output color of each pixel. The current implementation looks up normals with 8 bits per component, which seems sufficient in practice (using [44])

8 ONGOING WORK AND CONCLUSIONS

There are several directions to pursue to improve our system for appearance-preserving simplification. We would like to experiment more with techniques to generate parameterizations that allow efficient representations of the mapped attributes as well as guarantee a one-to-one mapping to the texture plane.

It would be nice for the simplification component to do a better job of optimizing the 3D and texture coordinates of the generated vertex for each edge collapse, both in 3D and the texture plane. Also, it may be interesting to allow the attribute data of a map to influence the error metric. We would also like to integrate our technique with a simplification algorithm like [6] that deals well with imperfect input meshes and allows some topological changes. Finally, we want to display our resulting progressive meshes in a system that performs dynamic, view-dependent management of LODs.

Our current system demonstrates the feasibility and desirability of our approach to appearance-preserving simplification. It produces high-fidelity images using a small number of high-quality triangles. This approach should complement future graphics systems well as we strive for increasingly realistic real-time computer graphics.

ACKNOWLEDGMENTS

We would like to thank Venkat Krishnamurthy and Marc Levoy at the Stanford Computer Graphics Laboratory and Peter Schröder for the armadillo model, and Lifeng Wang and Xing Xing Computer for the lion model. Our visualization system implementation was made possible by the UNC PixelFlow Project and the Hewlett Packard Visualize PxFl team. We also appreciate the assistance of the UNC Walkthrough Project. This work was supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract DAAH04-96-1-0257, NSF Grant CCR-9625217, ONR Young Investigator Award, Honda, Intel, NSF/ARPA Center for Computer Graphics and Scientific Visualization, and NIH/National Center for Research Resources Award 2P41RR02170-13 on Interactive Graphics for Molecular Studies and Microscopy.

REFERENCES

[1] L. Williams, "Pyramidal Parametrics," *SIGGRAPH 83 Conference Proceedings*, pp. 1--11, 1983.

[2] J. Rossignac and P. Borrel, "Multi-Resolution 3D Approximations for Rendering," in *Modeling in Computer Graphics*: Springer-Verlag, 1993, pp. 455--465.

[3] G. Schaufler and W. Sturzlinger, "Generating Multiple Levels of Detail from Polygonal Geometry Models," *Virtual Environments'95 (Eurographics Workshop)*, pp. 33-41, 1995.

[4] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen, "Decimation of Triangle Meshes," in *Proc. of ACM Siggraph*, 1992, pp. 65--70.

[5] G. Turk, "Re-tiling polygonal surfaces," *Comput. Graphics*, vol. 26, pp. 55--64, 1992.

[6] M. Garland and P. Heckbert, "Surface Simplification using Quadric Error Bounds," *SIGGRAPH'97 Conference Proceedings*, pp. 209-216, 1997.

[7] A. Gueziec, "Surface Simplification with Variable Tolerance," in *Second Annual Intl. Symp. on Medical Robotics and Computer Assisted Surgery (MRCAS '95)*, November 1995, pp. 132--139.

[8] J. Cohen, D. Manocha, and M. Olano, "Simplifying Polygonal Models Using Successive Mappings," *Proc. of IEEE Visualization'97*, pp. 395-402, 1997.

[9] R. Ronfard and J. Rossignac, "Full-range approximation of triangulated polyhedra," *Computer Graphics Forum*, vol. 15, pp. 67--76 and 462, August 1996.

[10] R. Klein, G. Liebich, and W. Straßer, "Mesh Reduction with Error Control," in *IEEE Visualization '96*: IEEE, October 1996.

[11] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright, "Simplification Envelopes," in *SIGGRAPH'96 Conference Proceedings*, 1996, pp. 119--128.

[12] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle, "Multiresolution Analysis of Arbitrary Meshes," in *SIGGRAPH'95 Conference Proceedings*, 1995, pp. 173--182.

[13] W. Schroeder, "A Topology Modifying Progressive Decimation Algorithm," *Proc. of IEEE Visualization'97*, pp. 205-212, 1997.

[14] J. El-Sana and A. Varshney, "Controlled Simplification of Genus for Polygonal Models," *Proc. of IEEE Visualization'97*, pp. 403-410, 1997.

[15] H. Hoppe, "Progressive Meshes," in *SIGGRAPH 96 Conference Proceedings*: ACM SIGGRAPH, 1996, pp. 99--108.

[16] H. Hoppe, "View-Dependent Refinement of Progressive Meshes," *SIGGRAPH'97 Conference Proceedings*, pp. 189-198, 1997.

[17] 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*, vol. 3, pp. 171--183, 1997.

[18] C. Bajaj and D. Schikore, "Error-bounded reduction of triangle meshes with multivariate data," *SPIE*, vol. 2656, pp. 34--45, 1996.

[19] M. Hughes, A. Lastra, and E. Saxe , "Simplification of Global-Illumination Meshes," *Proceedings of Eurographics '96, Computer Graphics Forum*, vol. 15, pp. 339-345, 1996.

[20] C. Erikson and D. Manocha, "Simplification Culling of Static and Dynamic Scene Graphs," UNC-Chapel Hill Computer Science TR98-009, 1998.

[21] P. Schroder and W. Sweldens, "Spherical Wavelets: Efficiently Representing Functions on the Sphere," *SIGGRAPH 95 Conference Proceedings*, pp. 161--172, August 1995.

[22] A. Certain, J. Popovic, T. Derose, T. Duchamp, D. Salesin, and W. Stuetzle, "Interactive Multiresolution Surface Viewing," in *Proc. of ACM Siggraph*, 1996, pp. 91--98.

[23] R. L. Cook, "Shade trees," in *Computer Graphics (SIGGRAPH '84 Proceedings)*, vol. 18, H. Christiansen, Ed., July 1984, pp. 223--231.

[24] J. Blinn, "Simulation of Wrinkled Surfaces," *SIGGRAPH '78 Conference Proceedings*, vol. 12, pp. 286--292, 1978.

[25] A. Fournier, "Normal distribution functions and multiple surfaces," *Graphics Interface '92 Workshop on Local Illumination*, pp. 45--52, 1992.

[26] M. Peercy, J. Airey, and B. Cabral, "Efficient Bump Mapping Hardware," *SIGGRAPH'97 Conference Proceedings*, pp. 303-306, 1997.

[27] M. Olano and A. Lastra, "A Shading Language on Graphics Hardware: The PixelFlow Shading System," *SIGGRAPH 98 Conference Proceedings*, 1998.

[28] J. Kajiya, "Anisotropic Reflection Models," *SIGGRAPH '85 Conference Proceedings*, pp. 15--21, 1985.

[29] B. Cabral, N. Max, and R. Springmeyer, "Bidirectional Reflection Functions From Surface Bump Maps," *SIGGRAPH '87 Conference Proceedings*, pp. 273--281, 1987.

[30] S. Westin, J. Arvo, and K. Torrance, "Predicting Reflectance Functions From Complex Surfaces," *SIGGRAPH '92 Conference Proceedings*, pp. 255--264, 1992.

[31] B. G. Becker and N. L. Max, "Smooth Transitions between Bump Rendering Algorithms," in *Computer Graphics (SIGGRAPH '93 Proceedings)*, vol. 27, J. T. Kajiya, Ed., August 1993, pp. 183--190.

[32] V. Krishnamurthy and M. Levoy, "Fitting Smooth Surfaces to Dense Polygon Meshes," *SIGGRAPH 96 Conference Proceedings*, pp. 313--324, 1996.

[33] D. G. Aliaga, "Visualization of Complex Models using Dynamic Texture-based Simplification," *Proc. of IEEE Visualization'96*, pp. 101--106, 1996.

[34] L. Darsa, B. Costa, and A. Varshney, "Navigating Static Environments using Image-space simplification and morphing," *Proc. of 1997 Symposium on Interactive 3D Graphics*, pp. 25-34, 1997.

[35] P. W. C. Maciel and P. Shirley, "Visual Navigation of Large Environments Using Textured Clusters," *Proc. of 1995 Symposium on Interactive 3D Graphics*, pp. 95--102, 1995.

[36] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder, "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments," *SIGGRAPH 96 Conference Proceedings*, pp. 75--82, August 1996.

[37] J. Kent, W. Carlson, and R. Parent, "Shape transformation for polyhedral objects," *SIGGRAPH '92 Conference Proceedings*, pp. 47--54, 1992.

[38] J. Maillot, H. Yahia, and A. Veroust, "Interactive Texture Mapping," *SIGGRAPH'93 Conference Proceedings*, pp. 27--34, 1993.

[39] H. Pedersen, "A Framework for Interactive Texturing Operations on Curved Surfaces," in *SIGGRAPH 96 Conference Proceedings*, *Annual Conference Series*, H. Rushmeier, Ed., August 1996, pp. 295--302.

[40] H. d. Fraysseix, J. Pach, and R. Pollack, "How to Draw a Planar Graph on a Grid," *Combinatorica*, vol. 10, pp. 41--51, 1990.

[41] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa, "A Linear Algorithm for Embedding Planar Graphs Using PQ-Trees," *J. Comput. Syst. Sci.*, vol. 30, pp. 54--76, 1985.

[42] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: an annotated bibliography," *Comput. Geom. Theory Appl.*, vol. 4, pp. 235--282, 1994.

[43] M. d. Berg, M. v. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*: Springer-Verlag, 1997.

[44] R. F. Lyon, "Phong Shading Reformulation for Hardware Renderer Simplification," Apple Computer #43, 1993.

Simplifying Polygonal Models Using Successive Mappings

Jonathan Cohen Dinesh Manocha Marc Olano
University of North Carolina at Chapel Hill
{cohenj,dm,olano}@cs.unc.edu

Abstract:
We present the use of mapping functions to automatically generate levels of detail with known error bounds for polygonal models. We develop a piece-wise linear mapping function for each simplification operation and use this function to measure deviation of the new surface from both the previous level of detail and from the original surface. In addition, we use the mapping function to compute appropriate texture coordinates if the original map has texture coordinates at its vertices. Our overall algorithm uses edge collapse operations. We present rigorous procedures for the generation of local planar projections as well as for the selection of a new vertex position for the edge collapse operation. As compared to earlier methods, our algorithm is able to compute tight error bounds on surface deviation and produce an entire continuum of levels of detail with mappings between them. We demonstrate the effectiveness of our algorithm on several models: a Ford Bronco consisting of over 300 parts and $70,000$ triangles, a textured lion model consisting of 49 parts and $86,000$ triangles, and a textured, wrinkled torus consisting of $79,000$ triangles.

CR Categories and Subject Descriptors: I.3.5 **[Computer Graphics]:** Computational Geometry and Object Modeling — Curve, surface, solid, and object representations.
Additional Key Words and Phrases: model simplification, levels-of-detail, surface approximation, projection, linear programming.

1 Introduction

Automatic generation of levels of detail for polygonal data sets has become a task of fundamental importance for real-time rendering of large polygonal environments on current graphics systems. Many detailed models are obtained by scanning physical objects using range scanning systems or created by modeling systems. Besides surface geometry these models, at times, contain additional information such as normals, texture coordinates, color etc. As the field of model simplification continues to mature, many applications desire high quality simplifications, with tight error bounds of various types across the surface being simplified.

Most of the literature on simplification has focused purely on surface approximation. Many of these techniques give guaranteed error bounds on the deviation of the simplified surface from the original surface. Such bounds are useful for providing a measure of the screen-space deviation from the original surface. A few techniques have been proposed to preserve other attributes such as color or overall appearance. However, they are not able to give tight error bounds on these parameters. At times the errors accumulated in all these domains may cause visible artifacts, even though the surface deviation itself is properly constrained. We believe the most promising approach to measuring and bounding these attribute errors is to have a mapping between the original surface and the simplified surface. With such a mapping in hand, we are free to devise suitable methods for measuring and bounding each type of error.

Main Contribution: In this paper we present a new simplification algorithm, which computes a piece-wise linear mapping between the original surface and the simplified surface. The algorithm uses the edge collapse operation due to its simplicity, local control, and suitability for generating smooth transitions between levels of detail. We also present rigorous and complete algorithms for collapsing an edge to a vertex such that there are no local self-intersections. The algorithm keeps track of surface deviation from both the current level of detail as well as from the original surface. The main features of our approach are:

1. **Successive Mapping:** This mapping between the levels of detail is a useful tool. We currently use the mapping in several ways: to measure the distance between the levels of detail before an edge collapse, to choose a location for the generated vertex that minimizes this distance, to accumulate an upper bound on the distance between the new level of detail and the original surface, and to map surface attributes to the simplified surface.

2. **Tight Error Bounds:** Our approach can measure and minimize the error for surface deviation and is extendible to other attributes. These error bounds give guarantees on the shape of the simplified object and screen-space deviation.

3. **Generality:** Portions of our approach can be easily combined with other algorithms, such as simplification envelopes [5]. Furthermore, the algorithm for collapsing an edge into a vertex is rather general and does not restrict the vertex to lie on the original edge.

4. **Surface Attributes:** Given an original surface with texture coordinates, our algorithm uses the successive mapping to compute appropriate texture coordinates for the simplified

mesh. Other attributes such as color or surface normal can also be maintained with the mapping.

5. **Continuum of Levels of Details:** The algorithm incrementally produces an entire spectrum of levels-of-details as opposed to a few discrete levels. Furthermore, the algorithm incrementally stores the error bounds for each level. Thus, the simplified model can be stored as a progressive mesh [12] if desired.

The algorithm has been successfully applied to a number of models. These models consist of hundreds of parts and tens of thousands of polygons, including a Ford Bronco with 300 parts, a textured lion model and a textured wrinkled torus.

Organization: The rest of the paper is organized as follows. In Section 2, we survey related work on model simplification. We give an overview of our algorithm in Section 3. Section 4 discusses the types of mappings computed by the algorithm and describes the algorithm in detail. In Section 5, we present applications of these mapping. The implementation is discussed in Section 6 and its performance in Section 7. Finally, in Section 8 we compare our approach to other algorithms.

2 Previous Work

Automatic simplification has been studied in both the computational geometry and computer graphics literature for several years [1, 3, 5, 6, 7, 8, 9, 10, 12, 11, 15, 16, 17, 18, 19, 21, 22, 24]. Some of the earlier work by Turk [22] and Schroeder [19] employed heuristics based on curvature to determine which parts of the surface to simplify to achieve a model with the desired polygon count. Other work include that of Rossignac and Borrel [16] where vertices close to each other are clustered and a vertex is generated to represent them. This algorithm has been used in the *Brush* walkthrough system [18]. A dynamic view-dependent simplification algorithm has been presented in [24].

Hoppe et al. [12, 11] posed the model simplification problem into a global optimization framework, minimizing the least-squares error from a set of point-samples on the original surface. Later, Hoppe extended this framework to handle other scalar attributes, explicitly recognizing the distinction between smooth gradients and sharp discontinuities. He also introduced the progressive mesh [12], which is essentially a stored sequence of simplification operations, allowing quick construction of any desired level of detail along the continuum of simplifications. However, the algorithm in [12] provides no guaranteed error bounds.

There is considerable literature on model simplification using error bounds. Cohen and Varshney et al. [5, 23] have used envelopes to preserve the model topology and obtain tight error bounds for a simple simplification. But they do not produce an entire spectrum of levels of detail. Guéziec [9] has presented an algorithm for computing local error bounds inside the simplification process by maintaining tolerance volumes. However, it does not produce a suitable mapping between levels of detail. Bajaj and Schikore [1, 17] have presented an algorithm for producing a mapping between approximations and measure the error of scalar fields across the surface based on vertex-removals. Some of the results presented in this paper extend this work non-trivially to edge collapse operation. A detailed comparison with these approaches is presented in Section 8.

An elegant solution to the polygon simplification problem has been presented in [7, 8] where arbitrary polygonal meshes are first subdivided into patches with *subdivision connectivity* and then multiresolution wavelet analysis is used over each patch. These methods preserve global topology, give error bounds on the simplified object and provide a mapping between levels of detail. In [3] they have been further extended to handle colored meshes. However, the initial mesh is not contained in the level of detail hierarchy, but can only be recovered to within an ϵ-tolerance. In some cases this is undesirable. Furthermore, the wavelet based approach can be somewhat conservative and for a given error bound, algorithms based on vertex removal and edge collapses [5, 12] have been *empirically* able to simplify more (in terms of reducing the polygon count).

3 Overview

Our simplification approach may be seen as a high-level algorithm which controls the simplification process with a lower-level cost function based on local mappings. Next we describe this high-level control algorithm and the idea of using local mappings for cost evaluation.

3.1 High-level Algorithm

At a broad level, our simplification algorithm is a generic greedy algorithm. Our simplification operation is the edge collapse. We initialize the algorithm by measuring the cost of all possible edge collapses, then we perform the edge collapses in order of increasing cost. The cost function tries to minimize local error bounds on surface deviation and other attributes. After performing each edge collapse, we locally re-compute the cost functions of all edges whose neighborhoods were affected by the collapse. This process continues until none of the remaining edges can be collapsed.

The output of our algorithm is the original model plus an ordered list of edge collapses and their associated cost functions. This *progressive mesh* [12] represents an entire *continuum* of levels of detail for the surface. A graphics application can choose to dynamically create levels of detail or to statically allocate a set of levels of detail to render the model with the desired quality or speed-up.

3.2 Local Mappings

The edge collapse operation we perform to simplify the surface contracts an edge (the *collapsed edge*) to a single, new vertex (the *generated vertex*). Most of the earlier algorithms position the generated vertex to one of the end vertices or mid-point of the collapse edge. However, these choices for generated vertex position may not minimize the deviation or error bound and can result in a local self-intersection. We choose a vertex position in two dimensions to avoid self-intersections and optimize in the third dimension to minimize error. This optimization of the generated vertex position and measurement of the error are the keys to simplifying the surface without introducing significant error.

For each edge collapse, we consider only the neighborhood of the surface that is modified by the operation (i.e. those faces, edges and vertices adjacent to the collapsed edge). There is a *natural mapping* between the neighborhood of the collapsed edge and the neighborhood of the generated vertex. Most of the triangles incident to the collapsed edge are stretched into corresponding triangles incident to the generated vertex. However, the two triangles that share the collapsed edge are themselves collapsed to edges (see Figure 1). These natural correspondences are one form of mapping

This natural mapping has two weaknesses.

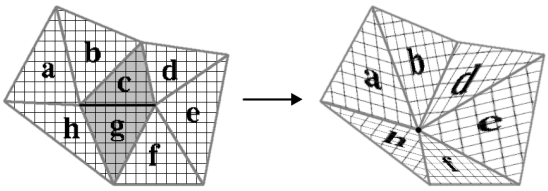


Figure 1: *The natural mapping primarily maps triangles to triangles. The two grey triangles map to edges, and the collapsed edge maps to the generated vertex*

1. The degeneracy of the triangles mapping to edges prevents us from mapping points of the simplified surface back to unique points on the original surface. This also implies that if we have any sort of attribute field across the surface, a portion of it disappears as a result of the operation.
2. The error implied by this mapping may be larger than necessary.

We measure the surface deviation error of the operation by the distances between corresponding points of our mapping. If we use the natural mapping, the maximum distance between any pair of points is defined as:

$$max(distance(\mathbf{v}_1, \mathbf{v}_{generated}), distance(\mathbf{v}_2, \mathbf{v}_{generated})),$$

where the collapsed edge corresponds to $(\mathbf{v}_1, \mathbf{v}_2)$ and $\mathbf{v}_{generated}$ is the generated vertex.

If we place the generated vertex at the midpoint of the collapsed edge, this distance error will be half the length of the edge. If we place the vertex at any other location, the error will be even greater.

We can create mappings that are free of degeneracies and often imply less error than the natural mapping. For simplicity, and to guarantee no self-intersections, we perform our mappings using planar projections of our local neighborhood. We refer to them as *successive mappings*.

4 Successive Mapping

In this section we present an algorithm to compute the mappings and their error bounds, which guide the simplification process. We present efficient and complete algorithms for computing a planar projection, finding a generated vertex in the plane, creating a mapping in the plane, and finally placing the generated vertex in 3D. The resulting algorithms utilize a number of techniques from computational geometry and are efficient in practice.

4.1 Computing a Planar Projection

Given a set of triangles in 3D, we present an efficient algorithm to compute a planar projection which is one-to-one to the set of triangles. The algorithm is guaranteed to find a plane, if it exists.

The projection we seek should be *one-to-one* to guarantee that the operations we perform in the plane are meaningful. For example, suppose we project a connected set of triangles onto a plane and then re-triangulate the polygon described by their boundary. The resulting set of triangles will contain no self-intersections, so long as the projection is one-to-one. Many other simplification algorithms, such as those by Turk [22], Schroeder [19] and Cohen, Varshney et al. [5], also used such projections for vertex removal. However, they would choose a likely direction, such as the average of the normal vectors of the triangles of interest. To test the validity of the resulting projection, these earlier algorithms would project all the triangles onto the plane and check for self-intersections. This process can be relatively expensive and is not guaranteed to find a one-to-one projecting plane.

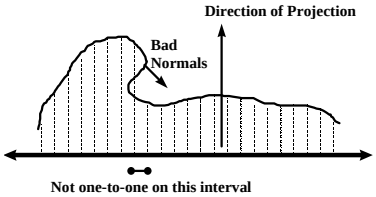


Figure 2: *A 2D example of an invalid projection*

We improve on earlier brute-force approaches in two ways. First, we present a simple, linear-time algorithm for testing the validity of a given direction. Second, we present a slightly more complex, but still expected linear-time, algorithm which will find a valid direction if one exists, or report that no such direction exists for the given set of triangles.

4.1.1 Validity Test for Planar Projection

In this section, we briefly describe the algorithm which checks whether a given set of triangles have a one-to-one planar projection. Assume that we can calculate a consistent set of normal vectors for the set of triangles in question (if we cannot, the surface is non-orientable and cannot be mapped onto a plane in a one-to-one fashion). If the angle between a given direction of projection and the normal vector of each of the triangles is less than 90°, then the direction of projection is valid, and defines a one-to-one mapping from the 3D triangles to a set of triangles in the plane of projection (any plane perpendicular to the direction of projection). Note that for a given direction of projection and a given set of triangles, this test involves only a single dot product and a sign test for each triangle in the set.

The correctness of the validity test can be established rigorously [4]. Due to space limitations, we do not present the detailed proof here. Rather, we give a short overview of the proof.

Figure 2 illustrates our problem in 2D. We would like to determine if the projection of the curve onto the line is one-to-one. Without loss of generality, assume the direction of projection is the y-axis. Each point on the curve projects to its x-coordinate on the line. If we traverse the curve from its left-most endpoint, we can project onto a previously projected location if and only if we reverse our direction along the x-axis. This can only occur when the y-component of the curve's normal vector goes from a positive value to a negative value. This is equivalent to our statement that the normal will be more than 90° from the direction of projection. With a little more work, we can show that this characterization generalizes to 3D.

4.1.2 Finding a valid direction

The validity test in the previous section provides a quick method of testing the validity of a likely direction as a one-to-one mapping projection. But the wider the spread of the normal vectors of our set of triangles, the less likely we are to find a valid direction by using any sort of heuristic. It is possible, in fact, to

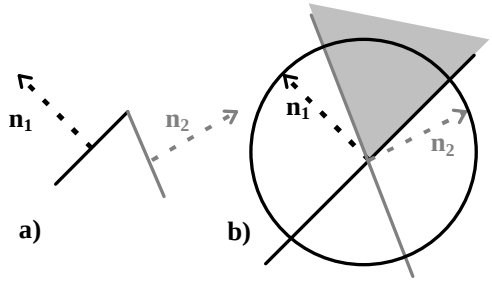


Figure 3: *A 2D example of the valid projection space. a) Two line segments and their normals. b) The 2D Gaussian circle, the planes corresponding to each segment, and the space of valid projection directions.*

compute the set of all valid directions of projection for a given set of triangles. However, to achieve greater efficiency and to reduce the complexity of the software system we choose to find only a single valid direction, which is typically all we require.

The *Gaussian sphere* [2] is the unit sphere on which each point corresponds to a unit normal vector with the same coordinates. Given a triangle, we define a plane through the origin with the same normal as the triangle. For a direction of projection to be valid with respect to this triangle, its point on the Gaussian sphere must lie on the correct side of this plane (i.e. within the correct hemisphere). If we consider two triangles simultaneously (shown in 2D in Figure 3) the direction of projection must lie on the correct side of the planes determined by the normal vectors of both triangles. This is equivalent to saying that the valid directions lie within the intersection of half-spaces defined by these two planes. Thus, the valid directions of projection for a set of N triangles lie within the intersection of N half-spaces.

This intersection of half-spaces forms a convex polyhedron. This polyhedron is a cone, with its apex at the origin and an unbounded base (shown as a triangular region in Figure 3). We can force this polyhedron to be bounded by adding more half-spaces (we use the six faces of a cube containing the origin). By finding a point on the interior of this cone and normalizing its coordinates, we shall construct a unit vector in the direction of projection.

Rather than explicitly calculating the boundary of the cone, we simply find a few corners (vertices) and average them to find a point that is strictly inside. By construction, the origin is definitely such a corner, so we just need to find three more *unique* corners to calculate an interior point. We can find each of these corners by solving a 3D *linear programming* problem. Linear programming allows us to find a point that maximizes a linear objective function subject to a collection of linear constraints [13]. The equations of the half-spaces serve as our linear constraints. We maximize in the direction of a vector to find the corner of our cone that lies the farthest in that direction.

As stated above, the origin is our first corner. To find the second corner, we try maximizing in the positive-x direction. If the resulting point is the origin, we instead maximize in the negative-x direction. To find the third corner, we maximize in a direction orthogonal to the line containing the first two corners. If the resulting point is one of the first two corners, we maximize in the opposite direction. Finally, we maximize in a direction orthogonal to the plane containing the first three corners. Once again, we may need to maximize in the opposite direction instead. Note that it is possible to reduce the worst-case number of optimizations from six to four by using the triangle normals to guide the selection of optimization vectors.

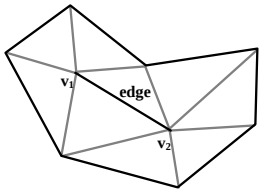


Figure 4: *The neighborhood of an edge as projected into 2D*

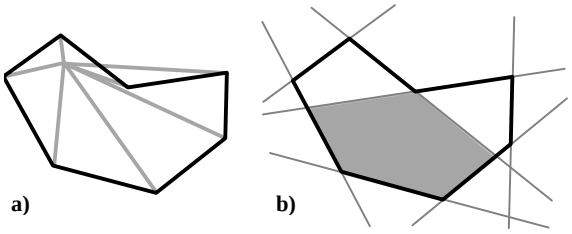


Figure 5: *a) An invalid 2D vertex position. b) The kernel of a polygon is the set of valid positions for a single, interior vertex to be placed. It is the intersection of a set of inward half-spaces.*

We used Seidel's linear time randomized algorithm [20] to solve each linear programming problem. A public domain implementation of this algorithm by Hohmeyer is available. It is very fast in practice.

4.2 Placing the Vertex in the Plane

In the previous section, we presented an algorithm to compute a valid plane. The edge collapse, which we use as our simplification operation, entails merging the two vertices of a particular edge into a single vertex. The topology of the resulting mesh is completely determined, but we are free to choose the position of the vertex, which will determine the geometry of the resulting mesh.

When we project the triangles neighboring the given edge onto a valid plane of projection, we get a triangulated polygon with two interior vertices, as shown in Figure 4. The edge collapse will reduce this edge to a single vertex. There will be edges connecting this generated vertex to each of the vertices of the polygon. In the context of this mapping approach, we would like the set of triangles around the generated vertex to have a one-to-one mapping with our chosen plane of projection, and thus to have a one-to- one mapping with the original edge neighborhood as well.

In this section, we present linear time algorithms both to test a candidate vertex position for validity, and to find a valid vertex position, if one exists.

4.2.1 Validity test for Vertex Position

The edge collapse operation leaves the boundary of the polygon in the plane unchanged. For the neighborhood of the generated vertex to have a one-to-one mapping with the plane, its edges must lie entirely within the polygon, ensuring that no edge crossings occur.

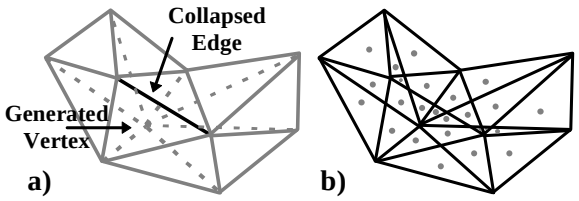


Figure 6: *a) Edge neighborhood and generated vertex neighborhood superimposed. b) A mapping in the plane, composed of 25 polygonal cells (each cell contains a dot). Each cell maps between a pair of planar elements in 3D.*

This 2D visibility problem has been well-studied in the computational geometry literature [14]. The generated vertex must have an unobstructed line of sight to each of the surrounding polygon vertices (unlike the vertex shown in Figure 5a). This condition holds if and only if the generated vertex lies within the polygon's *kernel*, shown in Figure 5b. This kernel is the intersection of inward-facing half-planes defined polygon's edges.

Given a potential vertex position in 2D, we test its validity by plugging it into the implicit-form equation for each of the polygon edges' line. If the position is on the interior with respect to each line, the position is valid, otherwise it is invalid.

4.2.2 Finding a Valid Position

The validity test highlighted above is useful if we wish to test out a likely candidate for the generated vertex position, such as the midpoint of the edge being collapsed. If such a heuristic choice succeeds, we can avoid the work necessary to compute a valid position directly.

Given the kernel definition for valid points, it is straightforward to find a valid vertex position using 2D linear programming. Each of the lines provides one of the constraints for the linear programming problem. Using the same methods as in Section 4.1.2, we can find a point in the kernel with no more than four calls to the linear programming routine. The first and second corners are found by maximizing in the positive- and negative-x directions. The final corner is found using a vector orthogonal to the first two corners.

4.3 Creating a Mapping in the Plane

After mapping the edge neighborhood to a valid plane and choosing a valid position for the generated vertex, we must define a mapping between the edge neighborhood and the generated vertex neighborhood. We shall map to each other the pairs of 3D points which project to identical points on the plane. These correspondences are shown in Figure 6a.

We can represent the mapping by a set of map cells, shown in Figure 6b. Each cell is a convex polygon in the plane and maps a piece of a triangle from the edge neighborhood to a similar piece of a triangle from the generated vertex neighborhood. The mapping represented by each cell is linear.

The vertices of the polygonal cells fall into *four* categories: vertices of the overall neighborhood polygon, vertices of the collapsed edge, the generated vertex itself, and edge-edge intersection points. We already know the locations of the first three categories of cell vertices, but we must calculate the edge-edge intersection points explicitly. Each such point is the intersection of an edge adjacent to the collapsed edge with an edge adjacent to the generated vertex. The number of such points can be quadratic (in the worst case) in the number of neighborhood edges. If we choose to construct the actual cells, we may do so by sorting the intersection points along each neighborhood edge and then walking the boundary of each cell.

4.4 Optimizing the 3D Vertex Position

Up to this point, we have projected the original edge neighborhood onto a plane, performed an edge collapse in this plane, and computed a mapping in the plane between these two local meshes. We are now ready to choose the position of the generated vertex in 3D. This 3D position will completely determine the geometry of the triangles surrounding the generated vertex.

To preserve our one-to-one mapping, it is necessary that all the points of the generated vertex neighborhood, including the generated vertex itself, project back into 3D along the direction of projection (the normal to the plane of projection). This restricts the 3D position of the generated vertex to the line parallel to the direction of projection and passing through the generated vertex's 2D position in the plane. We choose the vertex's position along this line such that it introduces as small a surface deviation as possible, that is it minimizes the maximum distance between any two corresponding points of the edge collapse neighborhood and the generated vertex neighborhood.

4.4.1 Distance function of the map

Each cell of our mapping determines a correspondence between a pair of planar elements. The maximum distance between any pair of planar functions must be at the boundary. For these pairs of polygons, the maximum distance must occur at a vertex. So the maximum distance for the entire mapping will always be at one of the interior cell vertices (because the cell vertices along the boundary do not move).

We parameterize the position of the generated vertex along its line of projection by a single parameter, t. As t varies, the distance between the corresponding cell vertices in 3D varies linearly. Note that these distances will always be along the direction of projection, because the distance between corresponding cell vertices is zero in the other two dimensions (those of the plane of projection). Because the distance is always positive, the distance function of each cell vertex is actually a pair of lines intersecting on the x-axis (shaped like a "V").

4.4.2 Minimizing the distance function

Given the distance function, we would like to choose the parameter t that minimizes the maximum distance between any pair of mapped points. This point is the minimum of the so-called *upper envelope*. For a set of k linear functions, we define the upper envelope function as follows:

$$U(t) = \{f_i(t) \mid f_i(t) \ > \ f_j(t) \, \forall \, i, j \ \ 1 \leq i, \ j \leq k; \ \ i \neq j\}.$$

For linear functions with no boundary conditions, this function is convex. Again we use linear programming to find the t value at which the minima occurs. We use this value of t to calculate the position of the generated vertex in 3D.

4.5 Accommodating Bordered Surfaces

Bordered surface are those containing edges adjacent to only a single triangle, as opposed to two triangles. Such surfaces are

quite common in practice. Borders create some complications for the creation of a mapping in the plane. The problem is that the total shape of the neighborhood projected into the plane changes as a result of the edge collapse.

Bajaj and Schikore [1], who employ a vertex-removal approach, deal with this problem by mapping the removed vertex to a length-parameterized position along the border. This solution can be employed for the edge-collapse operation as well. In their case, a single vertex maps to a point on an edge. In ours, three vertices map to points on a chain of edges.

5 Applying Mappings

The previous section described the steps required to compute a mapping using planar projections. Given such a mapping, we would now like to apply it to the problem of computing high-quality surface approximations. We will next discuss how to bound the distance from the current simplified surface to the original surface, and how to compute new values for scalar surface attributes at the generated vertex.

5.1 Approximation of Original Surface Position

In the process of creating a mapping, we have measured the distance between the current surface and the surface resulting from the application of one more simplification operation. What we eventually desire is the distance between this new surface and the *original* surface. One possible solution would be to incorporate the information from all the previous mappings into an increasingly complex mapping as the simplification process proceeds. While this approach has the potential for a high degree of accuracy, the increasing complexity of the mappings is undesirable.

Instead, we associate with every point on the current surface a volume that is guaranteed to contain the corresponding point on the original surface. This volume is chosen conservatively so we can use the same volume for all points in a triangle. Thus the portion of the original surface corresponding to the triangle lies within the convolution of the triangle and the volume.

Possible volume choices include axis-aligned boxes, triangle-aligned prisms and sphere. For computational efficiency, we use axis-aligned boxes. To improve the error bounds, we do not require the box to be centered at the point of application.

The initial box at every triangle has zero size and displacement. After computing the mapping in the plane and choosing the 3D vertex position, we propagate the error by adjusting the size and displacement of the box associated with each new triangle.

For each cell vertex, we create a box that contains the boxes of the old triangles that meet there. The box for each new triangle is then constructed to contain the boxes of all of its cell vertices. By maintaining this containment property at the cell vertices, we guarantee it for all the interior points of the cells.

The maximum error for each triangle is the distance between a point on the triangle and the farthest corner of its associated box. The error of the entire current mesh is the largest error of any of its triangles.

5.2 Computing Texture Coordinates

The use of texture maps has become common over the last several years, as the hardware support for texture mapping has increased. Texture maps provide visual richness to computer-rendered models without adding more polygons to the scene.

Texture mapping requires two *texture coordinates* at every vertex of the model. These coordinates provide a parameterization of the texture map over the surface.

As we collapse an edge, we must compute texture coordinates for the generated vertex. These coordinates should reflect the original parameterization of the texture over the surface. We use linear interpolation to find texture coordinates for the corresponding point on the old surface, and assign these coordinates to the generated vertex.

This approach works well in many cases, as demonstrated in Section 7. However, there can still be some sliding of the texture across the surface. We can extend our mapping approach to also measure and bound the deviation of the texture. This extension, currently under development, will provide more guarantees about the smoothness of transitions between levels of detail.

As we add more error measures to our system, it becomes necessary to decide how to weight these errors to determine the overall cost of an edge collapse. Each type of error at an edge mandates a particular viewing distance based on a user-specified screen-space tolerance (e.g. number of allowable pixels of surface or texel deviation). We conservatively choose the farthest of these. At run-time, the user can still adjust the overall screen-space tolerance, but the relationships between the types of error are fixed.

6 System Implementation

All the algorithms described in this paper have been implemented and applied to various models. While the simplification process itself is only a pre-process with respect to the graphics application, we would still like it to be as efficient as possible. The most time-consuming part of our implementation is the re-computation of edge costs as the surface is simplified (Section 3.1). To reduce this computation time, we allow our approach to be slightly less greedy. Rather than recompute all the local edge costs after a collapse, we simply set a *dirty flag* for these edges. If the next minimum-cost edge we pick to collapse is dirty, we re-compute it's cost and pick again. This *lazy evaluation* of edge costs significantly speeds up the algorithm without much effect on the error across the progressive mesh.

More important than the cost of the simplification itself is the speed at which our graphics application runs. To maximize graphics performance, our display application renders simplified objects only with display lists. After loading the progressive mesh, it takes snapshots to use as levels of detail every time the triangle count decreases by a factor of two. These choices limit the memory usage to twice the original number of triangles, and virtually eliminate any run-time cost of simplification.

7 Results

We have applied our simplification algorithm to four distinct objects: a bunny rabbit, a wrinkled torus, a lion, and a Ford Bronco, with a total of 390 parts. Table 1 shows the total input complexity of each of these objects as well as the time needed to generate a progressive mesh representation. All simplifications were performed on a Hewlett-Packard 735/125 workstation.

Figure 7 graphs the complexity of each object vs. the number of pixels of screen-space error for a particular viewpoint. Each set

Model	Parts	Orig. Triangles	CPU Time (Min:Sec)
Bunny	1	69,451	9:05
Torus	1	79,202	10:53
Lion	49	86,844	8:52
Bronco	339	74,308	6:55

Table 1: *Simplifications performed. CPU time indicates time to generate a progressive mesh of edge collapses until no more simplification is possible.*

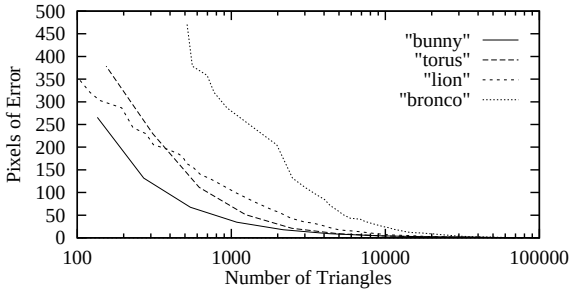


Figure 7: *Continuum of levels of detail for four models*

of data was measured with the object centered in the foreground of a 1000x1000-pixel viewport, with a 45° field-of-view, like the Bronco in Plates 2 and 3. This was the easiest way for us to measure the continuum. Conveniently, this function of complexity vs. error at a fixed distance is proportional to the function of complexity vs. viewing distance with a fixed error. The latter is typically the function of interest.

Plate 1 shows the typical way of viewing levels of detail – with a fixed error bound and levels of detail changing as a function of distance. Plates 2 and 3 show close-ups of the Bronco model at full and reduced resolution.

Plates 4 and 5 show the application of our algorithm to the texture-mapped lion and wrinkled torus models. If you know how to free-fuse stereo image pairs, you can fuse the torii or any of the adjacent pairs of textured lion. Because the torii are rendered at an appropriate distance for switching between the two levels of detail, the images are nearly indistinguishable, and fuse to a sharp, clear image. The lions, however, are not rendered at their appropriate viewing distances, so certain discrepancies will appear as fuzzy areas. Each of the lion's 49 parts is individually colored in the wire-frame rendering to indicate which of its levels of detail is currently being rendered.

7.1 Applications of Projection Algorithm

We have also applied the technique of finding a one-to-one planar projection to the simplification envelopes algorithm [5]. The simplification envelopes method requires the calculation of a vertex normal at each vertex that may be used as a direction to offset the vertex. The criterion for being able to move a vertex without creating a local self-intersection is the same as the criterion for being able to project to a plane. The algorithm presented in [5] used a heuristic based on averaging the face normals.

By applying the projection algorithm based on linear programming (presented in Section 4.1) to the computation of the offset directions, we were able to perform more drastic simplifications. The simplification envelopes method could previously only reduce the bunny model to about 500 triangles, without resulting in any self-intersections. Using the new approach, the algorithm can reduce the bunny to 129 triangles, with no self-intersections.

7.2 Video Demonstration

We have produced a video demonstrating the capabilities of the algorithm and smooth switching between different levels-of-details for different models. It shows the speed-up in the frame rate for eight circling Bronco models (about a factor of six) with almost no degradation in image quality. This is based on mapping the object space error bounds to screen space, which can measure the maximum error in number of pixels. The video also highlights the performance on simplifying textured models, showing smooth switching between levels of detail. The texture coordinates were computed using the algorithm in Section 5.2.

8 Comparison to Previous Work

While concrete comparisons are difficult to make without careful implementation of all the related approaches readily available, we compare some of the features of our algorithm with that of others. The efficient and complete algorithms for computing the planar projection and placing the generated vertex after edge collapse should improve the performance of all the earlier algorithms that use vertex removals or edge collapses.

We compared our implementation with that of the simplification envelopes approach [5]. We generated levels of detail of the Stanford bunny model (70,000 triangles) using the simplification envelopes method, then generated levels of detail with the same number of triangles using the successive mapping approach. Visually, the models were comparable. The error bounds for the simplification envelopes method were smaller by about a factor of two, but the error bounds for the two methods measure different things. Simplification envelopes only bounds the surface deviation in the direction normal to the original surface, while the mapping approach prevents the surface from sliding around as well. Also, simplification envelopes created local creases in the bunnies, resulting in some shading artifacts. The successive mapping approach discourages such creases by its use of planar projections. At the same time, the performance of the simplification envelopes approach (in terms complexity vs. error) has been improved by our new projection algorithm.

Hoppe's progressive mesh [12] implementation is more complete than ours in its handling of colors, textures, and discontinuities. However, this technique provides no guaranteed error bounds, so there is no simple way to automatically choose switching distances that guarantee some visual quality.

The multi-resolution analysis approach to simplification [7, 8] does, in fact, provide strict error bounds as well as a mapping between surfaces. However, the requirements of its subdivision topology and the coarse granularity of its simplification operation do not provide the local control of the edge collapse. In particular, it does not deal well with sharp edges. Hoppe [12] has compared his progressive meshes with the multi-resolution analysis meshes. For a given number of triangles, his progressive meshes provide much higher visual quality. Therefore, for a given error bound, we expect our mapping algorithm to be able to simplify more than the multi-resolution approach.

Guéziec's tolerance volume approach [9] also uses edge collapses with local error bounds. Unlike the boxes used by the successive mapping approach, Guéziec's error volume can grow as the simplified surface fluctuates closer to and farther away from the original surface. This is due to the fact that it uses spheres which always remain centered at the vertices, and the newer spheres must always contain the older spheres. The boxes used by our successive mapping approach are not centered on the surface and do not grow as a result of such fluctuations. Also, the tolerance volume approach does not generate mappings between the surfaces for use with other attributes.

We have made several significant improvements over the simplification algorithm presented by Bajaj and Schikore [1, 17]. First, we have replaced their projection heuristic with a robust algorithm for finding a valid direction of projection. Second, we have generalized their approach to handle more complex operations, such as the edge collapse. Finally, we have presented an error propagation algorithm which correctly bounds the error in the surface deviation. Their approach represented error as infinite slabs surrounding each triangle. Because there is no information about the extent of these slabs, it is impossible to correctly propagate the error from a slab with one orientation to a new slab with a different orientation.

9 Future Work

We are currently working on bounding the screen-space deviation of the texture coordinates. By bounding the error of the texture coordinates, we will provide one type of bound on the deviation of surface colors (from a texture map) or normals (from a bump map). We also plan to measure and bound the deviation of colors and normals specified directly at the polygon vertices.

There are cases where the projection onto a plane produces mappings with unnecessarily large error. We only optimize surface position in the direction orthogonal to the plane of projection. It would be useful to generate and optimize mappings directly in 3D to produce better simplifications.

Our system currently handles non-manifold topologies by breaking them into independent surfaces, which does not maintain connectivity between the components. Handling such non-manifold regions directly may provide higher visual fidelity for large screen-space tolerances.

10 Acknowledgments

We would like to thank Stanford Computer Graphics Laboratory for the bunny model, Stefan Gottschalk for the wrinkled torus model, Lifeng Wang and Xing Xing Computer for the lion model from the Yuan Ming Garden, and Division and Viewpoint for the Ford Bronco model. Thanks to Michael Hohmeyer for the linear programming library. We would also like to thank the UNC Walkthrough Group and Carl Mueller. This work was supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract DAAH04-96-1-0257, NSF Grant CCR-9319957, NSF Grant CCR-9625217, ONR Young Investigator Award, Intel, DARPA Contract DABT63-93-C-0048, NSF/ARPA Center for Computer Graphics and Scientific Visualization, and NIH/National Center for Research Resources Award 2 P41RR02170-13 on Interactive Graphics for Molecular Studies and Microscopy.

References

[1] C. Bajaj and D. Schikore. Error-bounded reduction of triangle meshes with multivariate data. *SPIE*, 2656:34–45, 1996.

[2] M.P. Do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice Hall, 1976.

[3] A. Certain, J. Popovic, T. Derose, T. Duchamp, D. Salesin, and W. Stuetzle. Interactive multiresolution surface viewing. In *Proc. of ACM Siggraph*, pages 91–98, 1996.

[4] J. Cohen, D. Manocha, and M. Olano. Simplifying polygonal models using successive mappings. Technical Report TR97-011, Department of Computer Science, UNC Chapel Hill, 1997.

[5] J. Cohen, A. Varshney, D. Manocha, and G. Turk et al. Simplification envelopes. In *Proc. of ACM Siggraph'96*, pages 119–128, 1996.

[6] M.J. Dehaemer and M.J. Zyda. Simplification of objects rendered by polygonal approximations. *Computer and Graphics*, 15(2):175–184, 1981.

[7] T. Derose, M. Lounsbery, and J. Warren. Multiresolution analysis for surfaces of arbitrary topology type. Technical Report TR 93-10-05, Department of Computer Science, University of Washington, 1993.

[8] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proc. of ACM Siggraph*, pages 173–182, 1995.

[9] A. Guéziec. Surface simplification with variable tolerance. In *Second Annual Intl. Symp. on Medical Robotics and Computer Assisted Surgery (MRCAS '95)*, pages 132–139, November 1995.

[10] P. Heckbert and M. Garland. Multiresolution modeling for fast rendering. *Proceedings of Graphics Interface '94*, pages 43–50, May 1994.

[11] H. Hoppe, T. Derose, T. Duchamp, J. Mcdonald, and W. Stuetzle. Mesh optimization. In *Proc. of ACM Siggraph*, pages 19–26, 1993.

[12] Hugues Hoppe. Progressive meshes. In *SIGGRAPH 96 Conference Proceedings*, pages 99–108. ACM SIGGRAPH, Addison Wesley, August 1996.

[13] B. Kolman and R. Beck. *Elementary Linear Programming with Applications*. Academic Press, New York, 1980.

[14] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1994.

[15] R. Ronfard and J. Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3):67–76, 462, Aug. 1996. Proc. Eurographics '96.

[16] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993.

[17] D. Schikore and C. Bajaj. Decimation of 2d scalar data with error control. Technical report, Computer Science Report CSD-TR-95-004, Purdue University, 1995.

[18] B. Schneider, P. Borrel, J. Menon, J. Mittleman, and J. Rossignac. Brush as a walkthrough system for architectural models. In *Fifth Eurographics Workshop on Rendering*, pages 389–399, July 1994.

[19] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen. Decimation of triangle meshes. In *Proc. of ACM Siggraph*, pages 65–70, 1992.

[20] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

[21] D. C. Taylor and W. A. Barrett. An algorithm for continuous resolution polygonalizations of a discrete surface. In *Proc. Graphics Interface '94*, pages 33–42, Banff, Canada, May 1994.

[22] G. Turk. Re-tiling polygonal surfaces. In *Proc. of ACM Siggraph*, pages 55–64, 1992.

[23] A. Varshney. *Hierarchical Geometric Approximations*. PhD thesis, University of N. Carolina, 1994.

[24] J.C. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, June 1997.

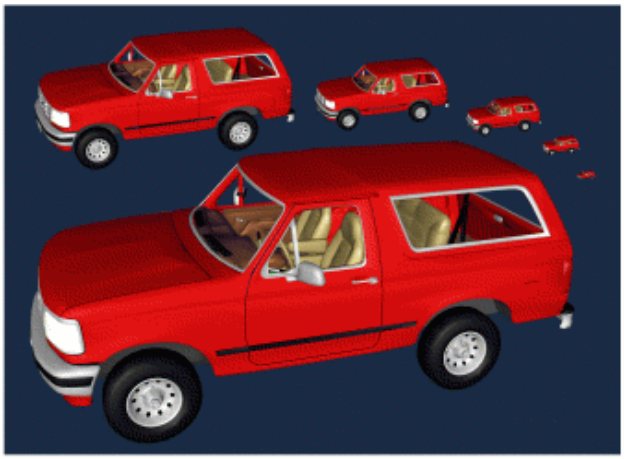


Plate 1: 6 views of the Ford Bronco model, all at 2 pixels of error (0.17 mm)

Triangle counts:	41,855	12,939
	27,970	8,385
	20,922	4,766

Plate 2: Bronco at full resolution 74,000 triangles

Plate 3: 26 pixels of error (4.4 mm) 9,000 triangles

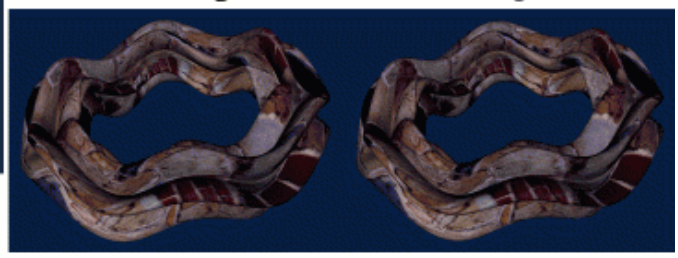


Plate 4: Wrinkled torus model at a transitional distance for 1 pixel of error (0.085 mm)

39,600 triangles 19,800 triangles

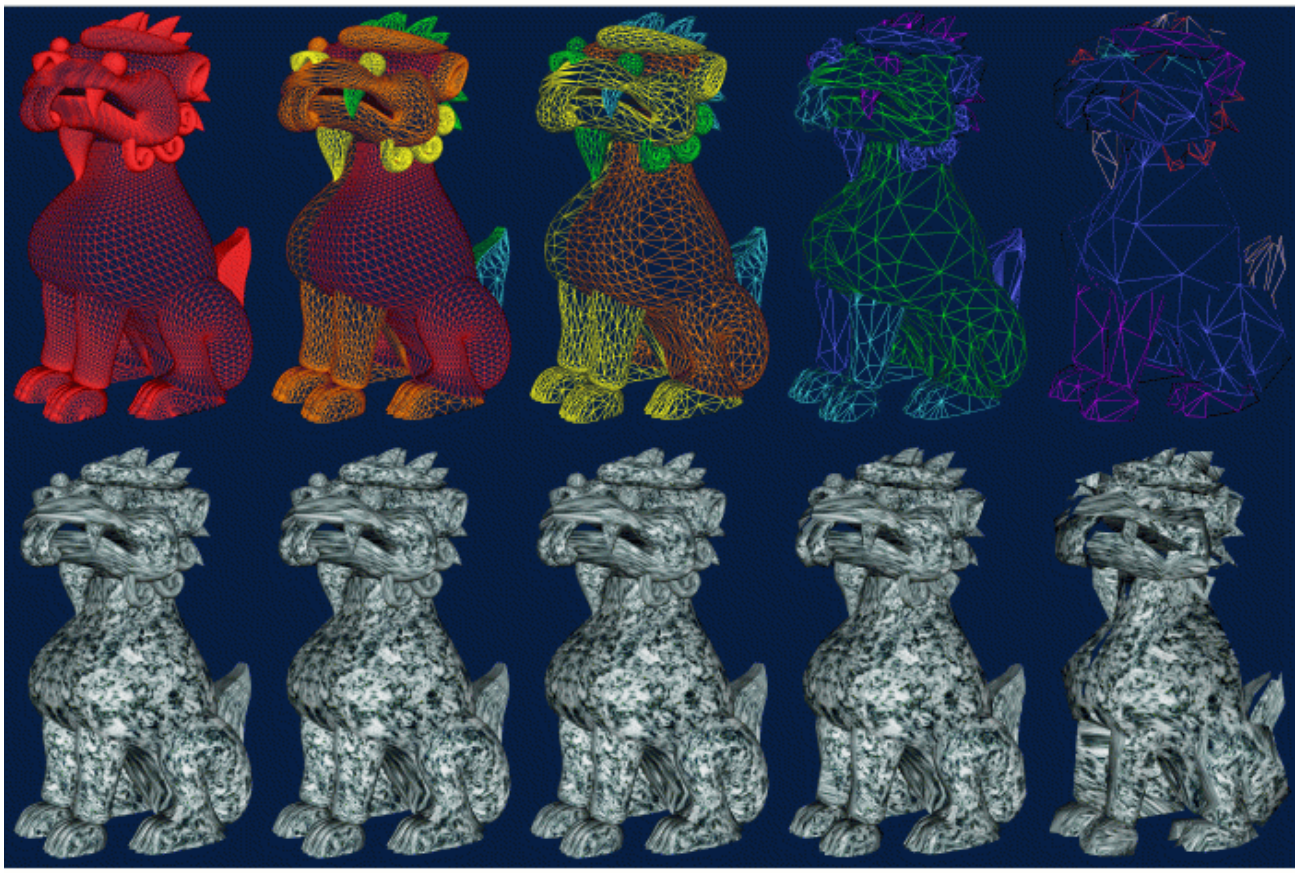


Plate 5: 6 levels of detail for the lion (colors indicate levels of detail of individual parts)

0 pixels of error	1 pixel of error	3 pixels of error	23 pixels of error	76 pixels of error
0 mm of error	0.085 mm of error	0.25 mm of error	1.9 mm of error	6.4 mm of error
86,000 triangles	29,000 triangles	14,000 triangles	4,000 triangles	1,000 triangles

Simplification Envelopes

Jonathan Cohen* Amitabh Varshney† Dinesh Manocha* Greg Turk* Hans Weber*
Pankaj Agarwal‡ Frederick Brooks* William Wright*
http://www.cs.unc.edu/~geom/envelope.html

Abstract

We propose the idea of *simplification envelopes* for generating a hierarchy of level-of-detail approximations for a given polygonal model. Our approach guarantees that all points of an approximation are within a user-specifiable distance ϵ from the original model and that all points of the original model are within a distance ϵ from the approximation. Simplification envelopes provide a general framework within which a large collection of existing simplification algorithms can run. We demonstrate this technique in conjunction with two algorithms, one local, the other global. The local algorithm provides a fast method for generating approximations to large input meshes (at least hundreds of thousands of triangles). The global algorithm provides the opportunity to avoid local "minima" and possibly achieve better simplifications as a result.

Each approximation attempts to minimize the total number of polygons required to satisfy the above ϵ constraint. The key advantages of our approach are:

- General technique providing guaranteed error bounds for genus-preserving simplification
- Automation of both the simplification process and the selection of appropriate viewing distances
- Prevention of self-intersection
- Preservation of sharp features
- Allows variation of approximation distance across different portions of a model

CR Categories and Subject Descriptors: I.3.3 **[Computer Graphics]:** Picture/Image Generation — Display algorithms; I.3.5 **[Computer Graphics]:** Computational Geometry and Object Modeling — Curve, surface, solid, and object representations.
Additional Key Words and Phrases: hierarchical approximation, model simplification, levels-of-detail generation, shape approximation, geometric modeling, offsets.

*Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175.
{cohenj,weberh,manocha,turk,brooks,wright}@cs.unc.edu
†Department of Computer Science, State University of New York, Stony Brook, NY 11794-4400. varshney@cs.sunysb.edu
‡Department of Computer Science, Duke University, Durham, NC 27708-0129. pankaj@cs.duke.edu

1 Introduction

We present the framework of *simplification envelopes* for computing various levels of detail of a given polygonal model. These hierarchical representations of an object can be used in several ways in computer graphics. Some of these are:

- Use in a level-of-detail-based rendering algorithm for providing desired frame update rates [4, 9].
- Simplifying traditionally over-sampled models such as those generated from volume datasets, laser scanners, and satellites for storage and reducing CPU cycles during processing, with relatively few or no disadvantages [10, 11, 13, 15, 21, 23].
- Using low-detail approximations of objects for illumination algorithms, especially radiosity [19].

Simplification envelopes are a generalization of *offset surfaces*. Given a polygonal representation of an object, they allow the generation of minimal approximations that are guaranteed not to deviate from the original by more than a user-specifiable amount while preserving global topology. We surround the original polygonal surface with two envelopes, then perform simplification within this volume. A sample application of the algorithms we describe can be seen in Figure 1.

Figure 1: *A level-of-detail hierarchy for the rotor from a brake assembly.*

Such an approach has several benefits in computer graphics. First, one can very precisely quantify the amount of approximation that is tolerable under given circumstances. Given a user-specified error in number of pixels of deviation of an object's silhouette, it is possible to choose which level of detail to view from a particular distance to maintain that pixel error bound. Second, this approach allows one a fine control over which regions of an object should be approximated more and which ones less. This could be used for selectively preserving those features of an object that are *perceptually* important. Third, the user-specifiable tolerance for approximation is the only parameter required to obtain the approximations; fine tweaking of several parameters depending upon the object to be approximated is not required. Thus, this approach is quite useful for *automating the process* of topology-preserving simplifications of a large number of objects. This problem of *scalability* is important for any simplification algorithm. One of our main goals is to create a method for simplification which is not only automatic for large datasets, but tends to preserve the shapes of the original models.

The rest of the paper is organized in the following manner: we survey the related work in Section 2, explain our assumptions and terminology in Section 3, describe the envelope and approximation computations in Sections 4 and 5, present some useful extentions to and properties of the approximation algorithms in Section 6, and explain our implementation and results in Section 7.

2 Background

Approximation algorithms for polygonal models can be classified into two broad categories:

- **Min-# Approximations**: For this version of the approximation problem, given some error bound ϵ, the objective is to minimize the number of vertices such that no point of the approximation $\mathcal{A}$ is farther than ϵ distance away from the input model $\mathcal{I}$.

- **Min-ϵ Approximations**: Here we are given the number of vertices of the approximation $\mathcal{A}$ and the objective is to minimize the error, or the difference, between $\mathcal{A}$ and $\mathcal{I}$.

Previous work in the area of min-# approximations has been done by [6, 20] where they adaptively subdivide a series of bicubic patches and polygons over a surface until they fit the data within the tolerance levels.

In the second category, work has been done by several groups. Turk [23] first distributes a given number of vertices over the surface depending on the curvature and then re-triangulates them to obtain the final mesh. Schroeder et al. [21] and Hinker and Hansen [13] operate on a set of local rules — such as deleting edges or vertices from almost coplanar adjacent faces, followed by local re-triangulation. These rules are applied iteratively till they are no longer applicable. A somewhat different local approach is taken in [18] where vertices that are close to each other are clustered and a new vertex is generated to represent them. The mesh is suitably updated to reflect this.

Hoppe et al. [14] proceed by iteratively optimizing an energy function over a mesh to minimize both the distance of the approximating mesh from the original, as well as the number of approximating vertices. An interesting and elegant solution to the problem of polygonal simplification by using wavelets has been presented in [7, 8] where arbitrary polygonal meshes are first subdivided into patches with *subdivision connectivity* and then multiresolution wavelet analysis is used over each patch. This wavelet approach preserves global topology.

In computational geometry, it has been shown that computing the minimal-facet ϵ-approximation is NP-hard for both convex polytopes [5] and polyhedral terrains [1]. Thus, algorithms to solve these problems have evolved around finding polynomial-time approximations that are *close* to the optimal.

Let k_o be the size of a min-# approximation. An algorithm has been given in [16] for computing an ϵ-approximation of size $O(k_o \log n)$ for convex polytopes. This has recently been improved by Clarkson in [3]; he proposes a randomized algorithm for computing an approximation of size $O(k_o \log k_o)$ in expected time $O(k_o n^{1+\delta})$ for any $\delta > 0$ (the constant of proportionality depends on δ, and tends to $+\infty$ as δ tends to 0). In [2] Brönnimann and Goodrich observed that a variant of Clarkson's algorithm yields a polynomial-time deterministic algorithm that computes an approximation of size $O(k_0)$. Working with polyhedral terrains, [1] present a polynomial-time algorithm that computes an ϵ-approximation of size $O(k_o \log k_o)$ to a polyhedral terrain.

Our work is different from the above in that it allows adaptive, genus-preserving, ϵ-approximation of arbitrary polygonal objects. Additionally, we can simplify bordered meshes and meshes with holes. In terms of direct comparison with the global topology preserving approach presented in [7, 8], for a given ϵ our algorithm has been *empirically* able to obtain "reduced" simplifications, which are much smaller in output size (as demonstrated in Section 7). The algorithm in [18] also guarantees a bound in terms of the Hausdorff metric. However, it is not guaranteed to preserve the genus of the original model.

3 Terminology and Assumptions

Let us assume that $\mathcal{I}$ is a three-dimensional compact and orientable object whose polygonal representation $\mathcal{P}$ has been given to us. Our objective is to compute a piecewise-linear approximation $\mathcal{A}$ of $\mathcal{P}$. Given two piecewise linear objects $\mathcal{P}$ and $\mathcal{Q}$, we say that $\mathcal{P}$ and $\mathcal{Q}$ are *ϵ-approximations* of each other iff every point on $\mathcal{P}$ is within a distance ϵ of some point of $\mathcal{Q}$ and every point on $\mathcal{Q}$ is within a distance ϵ of some point of $\mathcal{P}$. Our goal is to outline a method to generate two envelope surfaces surrounding $\mathcal{P}$ and demonstrate how the envelopes can be used to solve the following polygonal approximation problem. Given a polygonal representation $\mathcal{P}$ of an object and an approximation parameter ϵ, generate a genus-preserving ϵ-approximation $\mathcal{A}$ with minimal number of polygons such that the vertices of $\mathcal{A}$ are a subset of vertices of $\mathcal{P}$.

We assume that all polygons in $\mathcal{P}$ are triangles and that $\mathcal{P}$ is a well-behaved polygonal model, i.e., every edge has either one or two adjacent triangles, no two triangles interpenetrate, there are no unintentional "cracks" in the model, no T-junctions, etc.

We also assume that each vertex of $\mathcal{P}$ has a single normal vector, which must lie within 90° of the normal of each of its surrounding triangles. For the purpose of rendering, each vertex may have either a single normal or multiple normals. For the purpose of generating envelope surfaces, we shall compute a single vertex normal as a combination of the normals of the surrounding triangles.

The three-dimensional ϵ-offset surface for a parametric surface

$$\mathbf{f}(s,t) = (f_1(s,t), f_2(s,t), f_3(s,t)),$$

whose unit normal to $\mathbf{f}$ is

$$\mathbf{n}(s,t) = (n_1(s,t), n_2(s,t), n_3(s,t)),$$

is defined as $\mathbf{f}^\epsilon(s,t) = (f_1^\epsilon(s,t), f_2^\epsilon(s,t), f_3^\epsilon(s,t))$, where

$$f_i^\epsilon(s,t) = f_i(s,t) + \epsilon n_i(s,t).$$

Note that offset surfaces for a polygonal object can self-intersect and may contain non-linear elements. We define a simplification envelope $\mathcal{P}(+\epsilon)$ (respectively $\mathcal{P}(-\epsilon)$) for an object $\mathcal{I}$ to be a *polygonal* surface that lies *within* a distance of ϵ from every point p on $\mathcal{I}$ in the same (respectively opposite) direction as the normal to $\mathcal{I}$ at p. Thus, the simplification envelopes can be thought of as an approximation to offset surfaces. Henceforth we shall refer to simplification envelope by simply envelope.

Let us refer to the triangles of the given polygonal representation $\mathcal{P}$ as the *fundamental triangles*. Let $e = (v_1, v_2)$ be an edge of $\mathcal{P}$. If the normals $\mathbf{n}_1, \mathbf{n}_2$ to $\mathcal{I}$ at both v_1 and v_2, respectively, are identical, then we can construct a plane π_e that passes through the edge e and has a normal that is perpendicular to that of v_1. Thus v_1, v_2 and their normals all lie along π_e. Such a plane defines two half-spaces for edge e, say π_e^+ and π_e^- (see Fig 2(a)). However, in general the normals $\mathbf{n}_1$ and $\mathbf{n}_2$ at the vertices v_1 and v_2 need not be identical, in which case it is not clear how to define the two half-spaces for an edge. One choice is to use a *bilinear patch* that passes through v_1 and v_2 and has a tangent $\mathbf{n}_1$ at v_1 and $\mathbf{n}_2$ at v_2. Let us call such a bilinear patch for e as the *edge half-space* β_e. Let the two half-spaces for the edge e in this case be β_e^+ and β_e^-. This is shown in Fig 2(b).

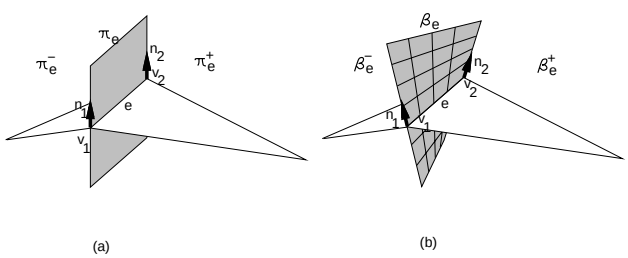


Figure 2: *Edge Half-spaces*

Let the vertices of a fundamental triangle be v_1, v_2, and v_3. Let the coordinates and the normal of each vertex v be represented as $\mathbf{c}(v)$ and $\mathbf{n}(v)$, respectively. The coordinates and the normal of a $(+\epsilon)$-offset vertex v_i^+ for a vertex v_i are: $\mathbf{c}(v_i^+) = \mathbf{c}(v_i) + \epsilon\mathbf{n}(v_i)$, and $\mathbf{n}(v_i^+) = \mathbf{n}(v_i)$. The $(-\epsilon)$-offset vertex can be similarly defined in the opposite direction. These offset vertices for a fundamental triangle are shown in Figure 3.

Now consider the closed object defined by v_i^+ and v_i^-, $i = 1, 2, 3$. It is defined by two triangles, at the top and bottom, and three edge half-spaces. This object contains the fundamental triangle (shown shaded in Figure 3) and we will henceforth refer to it as the *fundamental prism*.

4 Envelope Computation

In order to preserve the input topology of $\mathcal{P}$, we desire that the simplification envelopes do not self-intersect. To meet this criterion we reduce our level of approximation at certain places. In other words, to guarantee that no intersections amongst the envelopes occur, we have to be

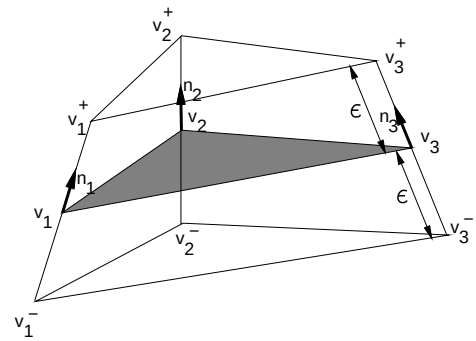


Figure 3: *The Fundamental Prism*

content at certain places with the distance between $\mathcal{P}$ and the envelope being smaller than ϵ. This is how simplification envelopes differ from offset surfaces.

We construct our envelope such that each of its triangles corresponds to a fundamental triangle. We offset each vertex of the original surface in the direction of its normal vector to transform the fundamental triangles into those of the envelope.

If we offset each vertex v_i by the same amount ϵ, to get the offset vertices v_i^+ and v_i^-, the resulting envelopes, $\mathcal{P}(+\epsilon)$ and $\mathcal{P}(-\epsilon)$, may contain self-intersections because one or more offset vertices are closer to some non-adjacent fundamental triangle. In other words, if we define a Voronoi diagram over the fundamental triangles of the model, the condition for the envelopes to intersect is that there be at least one offset vertex lying in the Voronoi region of some non-adjacent fundamental triangle. This is shown in Figure 4 by means of a two-dimensional example. In the figure, the offset vertices b^+ and c^+ are in the Voronoi regions of edges other than their own, thus causing self-intersection of the envelope.

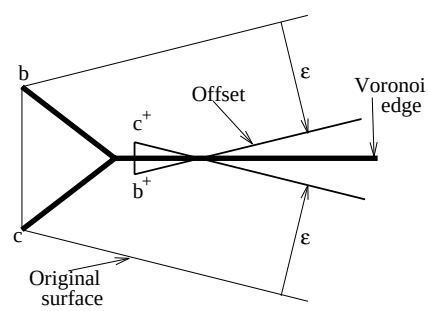


Figure 4: *Offset Surfaces*

Once we make this observation, the solution to avoid self-intersections becomes quite simple — just do not allow an offset vertex to go beyond the Voronoi regions of its adjacent fundamental triangles. In other words, determine the positive and negative ϵ for each vertex v_i such that the vertices v_i^+ and v_i^- determined with this new ϵ do not lie in the Voronoi regions of the non-adjacent fundamental triangles.

While this works in theory, efficient and robust computation of the three-dimensional Voronoi diagram of the fundamental triangles is non-trivial. We now present two methods for computing the reduced ϵ for each vertex, the first method analytical, and the second numerical.

4.1 Analytical ϵ Computation

We adopt a conservative approach for recomputing the ϵ at each vertex. This approach underestimates the values for the positive and negative ϵ. In other words, it guarantees the envelope surfaces not to intersect, but it does not guarantee that the ϵ at each vertex is the largest permissible ϵ. We next discuss this approach for the case of computing the positive ϵ for each vertex. Computation of negative ϵ follows similarly.

Consider a fundamental triangle t. We define a prism t_p for t, which is conceptually the same as its fundamental prism, but uses a value of 2ϵ instead of ϵ for defining the envelope vertices. Next, consider all triangles Δ_i that do not share a vertex with t. If Δ_i intersects t_p above t (the directions above and below t are determined by the direction of the normal to t, above is in the same direction as the normal to t), we find the point on Δ_i that lies within t_p and is closest to t. This point would be either a vertex of Δ_i, or the intersection point of one of its edges with the three sides of the prism t_p. Once we find the point of closest approach, we compute the distance δ_i of this point from t. This is shown in Figure 5.

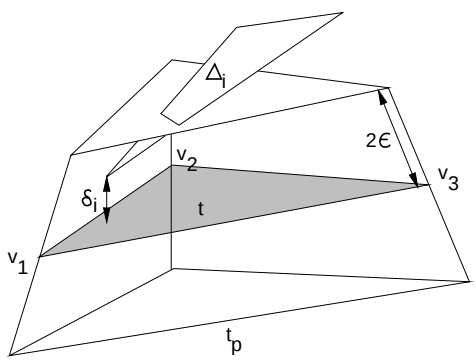


Figure 5: *Computation of δ_i*

Once we have done this for all Δ_i, we compute the new value of the positive ϵ for the triangle t as $\epsilon_{new} = \frac{1}{2}\min_i \delta_i$. If the vertices for this triangle t have this value of positive ϵ, their positive envelope surface will not self-intersect. Once the $\epsilon_{new}(t)$ values for all the triangles t have been computed, the $\epsilon_{new}(v)$ for each vertex v is set to be the minimum of the $\epsilon_{new}(t)$ values for all its adjacent triangles.

We use an octree in our implementation to speed up the identification of triangles Δ_i that intersect a given prism.

4.2 Numerical ϵ Computation

To compute an envelope surface numerically, we take an iterative approach. Our envelope surface is initially identical to the input model surface. In each iteration, we sequentially attempt to move each envelope vertex a fraction of the ϵ distance in the direction of its normal vector (or the opposite direction, for the inner envelope). This effectively stretches or contracts all the triangles adjacent to the vertex. We test each of these adjacent triangles for intersection with each other and the rest of the model. If no such intersections are found, we accept the step, leaving the vertex in this new position. Otherwise we reject it, moving the vertex back to its previous position. The iteration terminates when all vertices have either moved ϵ or can no longer move.

In an attempt to guarantee that each vertex gets to move a reasonable amount of its potential distance, we use an adaptive step size. We encourage a vertex to move at least K (an arbitrary constant which is scaled with respect to ϵ and the size of the object) steps by allowing it to reduce its step size. If a vertex has moved less than K steps and its move is been rejected, it divides its step size in half and tries again (with some maximum number of divides allowed on any particular step). Notice that if a vertex moves i steps and is rejected on the $(i+1)$st step, we know it has moved at least $i/(i+1)$ % of its potential distance, so $K/(K+1)$ which is a lower bound of sorts. It is possible, though rare, for a vertex to move less than K steps, if its current position is already quite close to another triangle.

Each vertex also has its own initial step size. We first choose a global, maximum step size based on a global property: either some small percentage of the object's bounding box diagonal length or ϵ/K, whichever is smaller. Now for each vertex, we calculate a local step size. This local step size is some percentage of the vertex's shortest incident edge (only those edges within 90° of the offset direction are considered). We set the vertex's step size to the minimum of the global step size and its local step size. This makes it likely that each vertex's step size is appropriate for a step given the initial mesh configuration.

This approach to computing an envelope surface is robust, simple to implement (if difficult to explain), and fair to all the vertices. It tends to maximize the minimum offset distance amongst the envelope vertices. It works fairly well in practice, though there may still be some room for improvement in generating maximal offsets for thin objects. Figure 6 shows internal and external envelopes computed for three values of ϵ using this approach.

As in the analytical approach, a simple octree data structure makes these intersection tests reasonably efficient, especially for models with evenly sized triangles.

5 Generation of Approximation

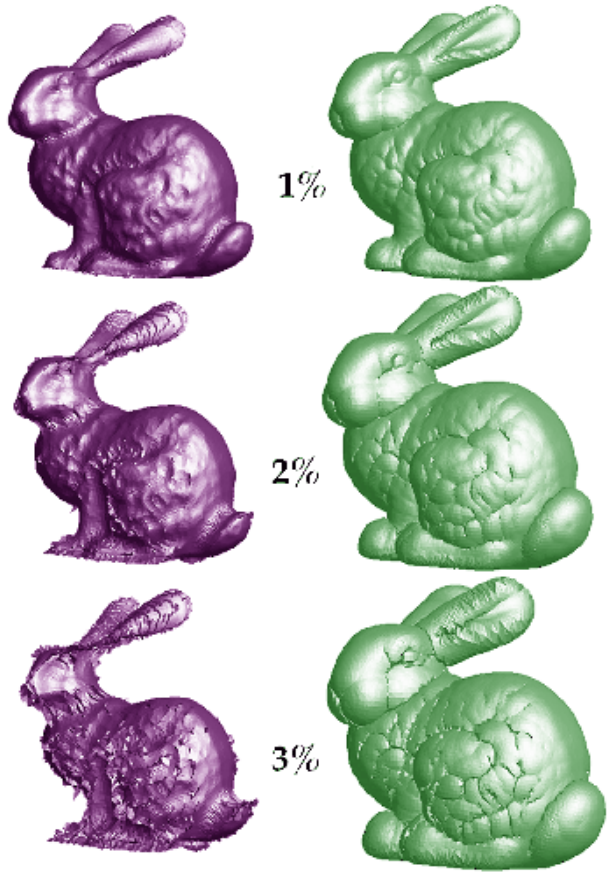
Generating a surface approximation typically involves starting with the input surface and iteratively making modifications to ultimately reduce its complexity. This process may be broken into two main stages: *hole creation*, and *hole filling*. We first create a hole by removing some connected set of triangles from the surface mesh. Then we fill the hole with a smaller set of triangles, resulting in some reduction of the mesh complexity.

We demonstrate the generality of the simplification envelope approach by designing two algorithms. The hole filling stages of these algorithms are quite similar, but their hole creation stages are quite different. The first algorithm makes only local choices, creating relatively small holes, while the second algorithm uses global information about the surface to create maximally-sized holes. These design choices produce algorithms with very different properties.

We begin by describing the envelope validity test used to determine whether a *candidate triangle* is valid for inclusion in the approximation surface. We then proceed to the two example simplification algorithms and a description of their relative merits.

5.1 Validity Test

A *candidate triangle* is one which we are considering for inclusion in an approximation to the input surface. Valid candidate triangles must lie between the two envelopes. Because we construct candidate triangles from the vertices of the original model, we know its vertices lie between the two envelopes. Therefore, it is sufficient to test the candidate triangle for intersections with the two envelope



Inner Envelopes ϵ **Outer Envelopes**

Figure 6: *Simplification envelopes for various* ϵ

surfaces. We can perform such tests efficiently using a space-partitioning data structure such as an octree.

A valid candidate triangle must also not cause a self-intersection in our surface, Therefore, it must not intersect any triangle of the current approximation surface.

5.2 Local Algorithm

To handle large models efficiently within the framework of simplification envelopes we construct a vertex-removal-based local algorithm. This algorithm draws heavily on the work of [21], [23], and [14]. Its main contributions are the use of envelopes to provide global error bounds as well as topology preservation and non-self-intersection. We have also explored the use of a more exhaustive hole-filling approach than any previous work we have seen.

The local algorithm begins by placing all vertices in a queue for removal processing. For each vertex in the queue, we attempt to remove it by creating a hole (removing the vertex's adjacent triangles) and attempting to fill it. If we can successfully fill the hole, the mesh modification is accepted, the vertex is removed from the queue, and its neighbors are placed back in the queue. If not, the vertex is removed from the queue and the mesh remains unchanged. This process terminates when the global error bounds eventually prevent the removal of any more vertices. Once the vertex queue is empty we have our simplified mesh.

For a given vertex, we first create a hole by removing all adjacent triangles. We begin the hole-filling process by generating all possible triangles formed by combinations of the vertices on the hole boundary. This is not strictly necessary, but it allows us to use a greedy strategy to favor triangles with nice aspect ratios. We fill the hole by choosing a triangle, testing its validity, and recursively filling the three (or fewer) smaller holes created by adding that triangle into the hole (see figure 7). If a hole cannot be filled at any level of the recursion, the entire hole filling attempt is considered a failure. Note that this is a single-pass hole filling strategy; we do not backtrack or undo selection of a triangle chosen for filling a hole. Thus, this approach does not guarantee that if a triangulation of a hole exists we will find it. However, it is quite fast and works very well in practice.

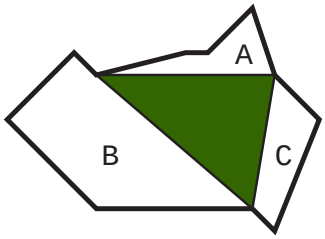


Figure 7: *Hole filling: adding a triangle into a hole creates up to three smaller holes*

We have compared the above approach with an exhaustive approach in which we tried all possible hole-filling triangulations. For simplifications resulting in the removal of hundreds of vertices (like highly oversampled laser-scanned models), the exhaustive pass yielded only a small improvement over the single-pass heuristic. This sort of confirmation reassures us that the single-pass heuristic works well in practice.

5.3 Global Algorithm

This algorithm extends the algorithm presented in [3] to non-convex surfaces. Our major contribution is the use of simplification envelopes to bound the error on a non-convex polygonal surface and the use of fundamental prisms to provide a generalized projection mechanism for testing for regions of multiple covering (overlaps). We present only a sketch of the algorithm here ; see [24] for the full details.

We begin by generating all possible candidate triangles for our approximation surface. These triangles are all 3-tuples of the input vertices which do not intersect either of the offset surfaces. Next we determine how many vertices each triangle *covers*. We rank the candidate triangles in order of decreasing covering.

We then choose from this list of candidate triangles in a greedy fashion. For each triangle we choose, we create a large hole in the current approximation surface, removing all triangles which *overlap* this candidate triangle. Now we begin the recursive hole-filling process by placing this candidate triangle into the hole and filling all the subholes with other triangles, if possible. One further restriction in this process is that the candidate triangle we are testing should not overlap any of the candidate triangles we have previously accepted.

5.4 Algorithm Comparison

The local simplification algorithm is fast and robust enough to be applied to large models. The global strategy is theoretically elegant. While the global algorithm works well for small models, its complexity rises at least quadratically,

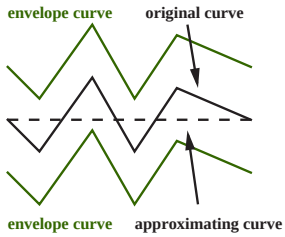


Figure 8: *Curve at local minimum of approximation*

making it prohibitive for larger models. We can think of the simplification problem as an optimization problem as well. A purely local algorithm may get trapped in a local "minimum" of simplification, while an ideal global algorithm will avoid all such minima.

Figure 8 shows a two-dimensional example of a curve for which a local vertex removal algorithm might fail, but an algorithm that globally searches the solution space will succeed in finding a valid approximation. Any of the interior vertices we remove would cause a new edge to penetrate an envelope curve. But if we remove all of the interior vertices, the resulting edge is perfectly acceptable.

This observation motivates a wide range of algorithms of which our local and global examples are the two extremes. We can easily imagine an algorithm that chooses nearby groups of vertices to remove simultaneously rather than sequentially. This could potentially lead to increased speed and simplification performance. However, choosing such sets of vertices remains a challenging problem.

6 Additional Features

Envelope surfaces used in conjunction with simplification algorithms are powerful, general-purpose tools. As we will now describe, they implicitly preserve sharp edges and can be extended to deal with bordered surfaces and perform adaptive approximations.

6.1 Preserving Sharp Edges

One of the important properties in any approximation scheme is the way it preserves any sharp edges or normal discontinuities present in the input model. Simplification envelopes deal gracefully with sharp edges (those with a small angle between their adjacent faces). When the ϵ tolerance is small, there is not enough room to simplify across these sharp edges, so they are automatically preserved. As the tolerance is increased, it will eventually be possible to simplify across the edges (which should no longer be visible from the appropriate distance). Notice that it is not necessary to explicitly recognize these sharp edges.

6.2 Bordered Surfaces

A bordered surface is one containing points that are homeomorphic to a half-disc. For polygonal models, this corresponds to edges that are adjacent to a single face rather than two faces. Depending on the context, we may naturally think of this as the boundary of some plane-like piece of a surface, or a hole in an otherwise closed surface.

The algorithms described in 5 are sufficient for closed triangle meshes, but they will not guarantee our global error bound for meshes with borders. While the envelopes constrain our error with respect to the normal direction of the surface, bordered surfaces require some additional constraints to hold the approximation border close to the original border. Without such constraints, the border of the approximation surface may "creep in," possibly shrinking the surface out of existence.

In many cases, the complexity of a surface's border curves may become a limiting factor in how much we can simplify the surface, so it is unacceptable to forgo simplifying these borders.

We construct a set of border tubes to constrain the error in deviation of the border curves. Each border is actually a cyclic polyline. Intuitively speaking, a border tube is a smooth, non-self-intersecting surface around one of these polylines. Removing a border vertex causes a pair of border edges to be replaced by a single border edge. If this new border edge does not intersect the relevant border tube, we may safely attempt to remove the border vertex.

To construct a tube we define a plane passing through each vertex of the polyline. We choose a coordinate system on this plane and use that to define a circular set of vertices. We connect these vertices for consecutive planes to construct our tube. Our initial tubes have a very narrow radius to minimize the likelihood of self-intersections. We then expand these narrow tubes using the same technique we used previously to construct our simplification envelopes.

The difficult task is to define a coordinate system at each polyline vertex which encourages smooth, non-self-intersecting tubes. The most obvious approach might be to use the idea of Frenet frames from differential geometry to define a set of coordinate systems for the polyline vertices. However, Frenet frames are meant for smooth curves. For a jagged polyline, a tube so constructed often has many self-intersections.

Instead, we use a projection method to minimize the twist between consecutive frames. Like the Frenet frame method, we place the plane at each vertex so that the normal to the plane approximates the tangent to the polyline. This is called the *normal plane*.

At the first vertex, we choose an arbitrary orthogonal pair of axes for our coordinate system in the normal plane. For subsequent vertices, we project the coordinate system from the previous normal plane onto the current normal frame. We then orthogonalize this projected coordinate system in the plane. For the normal plane of the final polyline vertex, we average the projected coordinate systems of the previous normal plane and the initial normal plane to minimize any twist in the final tube segment.

6.3 Adaptive Approximation

For certain classes of objects it is desirable to perform an adaptive approximation. For instance, consider large terrain datasets, models of spaceships, or submarines. One would like to have more detail near the observer and less detail further away. A possible solution could be to subdivide the model into various spatial cells and use a different ϵ-approximation for each cell. However, problems would arise at the boundaries of such cells where the ϵ-approximation for one cell, say at a value ϵ_1 need not necessarily be continuous with the ϵ-approximation for the neighboring cell, say at a different value ϵ_2.

Since all candidate triangles generated are constrained to lie within the two envelopes, manipulation of these envelopes provides one way to smoothly control the level of approximation. Thus, one could specify the ϵ at a given vertex to be a function of its distance from the observer — the larger the distance, the greater is the ϵ.

As another possibility, consider the case where certain

features of a model are very important and are not to be approximated beyond a certain level. Such features might have human perception as a basis for their definition or they might have mathematical descriptions such as regions of high curvature. In either case, a user can vary the ϵ associated with a region to increase or decrease the level of approximation. The bunny in Figure 9 illustrates such an approximation.

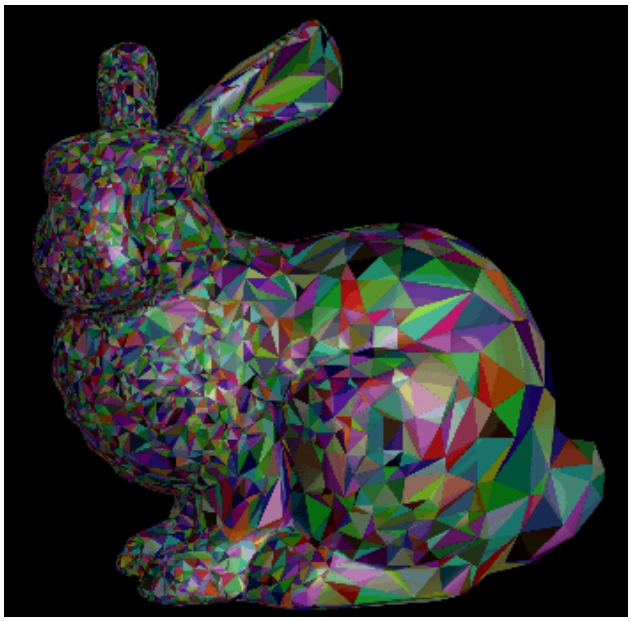


Figure 9: *An adaptive simplification for the bunny model. ϵ varies from 1/64% at the nose to 1% at the tail.*

7 Implementation and Results

We have implemented both algorithms and tried out the local algorithm on several thousand objects. We will first discuss some of the implementation issues and then present some results.

7.1 Implementation Issues

The first important implementation issue is what sort of input model to accept. We chose to accept only manifold triangle meshes (or bordered manifolds). This means that each edge is adjacent to two (one in the case of a border) triangles and that each vertex is surrounded by a single ring of triangles.

We also do not accept other forms of degenerate meshes. Many mesh degeneracies are not apparent on casual inspection, so we have implemented an automatic degeneracy detection program. This program detects non-manifold vertices, non-manifold edges, sliver triangles, coincident triangles, T-junctions, and intersecting triangles in a proposed input mesh. Note that correcting these degeneracies is more difficult than detecting them.

Robustness issues are important for implementations of any geometric algorithms. For instance, the analytical method for envelope computation involves the use of bilinear patches and the computation of intersection points. The computation of intersection points, even for linear elements, is difficult to perform robustly. The numerical method for envelope computation is much more robust because it involves only linear elements. Furthermore, it requires an intersection test but not the calculation of intersection points. We perform all such intersection tests in a conservative manner, using fuzzy intersection tests that may report intersections even for some close but non-intersecting elements.

Another important issue is the use of a space-partitioning scheme to speed up intersection tests. We chose to use an octree because of its simplicity. Our current octree implementation deals only with the bounding boxes of the elements stored. This works well for models with triangles that are evenly sized and shaped. For CAD models, which may contain long, skinny, non-axis-aligned triangles, a simple octree does not always provide enough of a speed-up, and it may be necessary to choose a more appropriate space-partitioning scheme.

7.2 Results

We have simplified a total of 2636 objects from the auxiliary machine room (AMR) of a submarine dataset, pictured in Figure 10 to test and validate our algorithm. We reproduce the timings and simplifications achieved by our implementation for the AMR and a few other models in Table 1. All simplifications were performed on a Hewlett-Packard 735/125 with 80 MB of main memory. Images of these simplifications appear in Figures 11 and 12. It is interesting to compare the results on the bunny and phone models with those of [7, 8]. For the same error bound, we are able to obtain much improved solutions.

We have automated the process which sets the ϵ value for each object by assigning it to be a percentage of the diagonal of its bounding box. We obtained the reductions presented in Table 1 for the AMR and Figures 11 and 12 by using this heuristic.

For the rotor and AMR models in the above results, the ith level of detail was obtained by simplifying the $i-1^{th}$ level of detail. This causes to total ϵ to be the sum of all previous ϵ's, so choosing $\epsilon's$ of 1, 2, 4, and 8 percent results in total $\epsilon's$ of 1, 3, 7, and 15 percent. There are two advantages to this scheme:
(a) It allows one to proceed incrementally, taking advantage of the work done in previous simplifications.
(b) It builds a hierarchy of detail in which the vertices at the ith level of detail are a subset of the vertices at the $i-1^{th}$ level of detail.

One of the advantages of the setting ϵ to a percent of the object size is that it provides an a way to automate the selection of switching points used to transition between the various representations. To eliminate visual artifacts, we switch to a more faithful representation of an object when ϵ projects to more than some user-specified number of pixels on the screen. This is a function of the ϵ for that approximation, the output display resolution, and the corresponding maximum tolerable visible error in pixels.

8 Future Work

There are still several areas to be explored in this research. We believe the most important of these to be the generation of correspondences between levels of detail and the moving of vertices within the envelope volume.

Bunny			Phone			Rotor			AMR		
ϵ %	# Polys	Time	ϵ %	# Polys	Time	ϵ %	# Polys	Time	ϵ %	# Polys	Time
0	69,451	N/A	0	165,936	N/A	0	4,735	N/A	0	436,402	N/A
1/64	44,621	9	1/64	43,537	31	1/8	2,146	3	1	195,446	171
1/32	23,581	10	1/32	12,364	35	1/4	1,514	2	3	143,728	61
1/16	10,793	11	1/16	4,891	38	3/4	1,266	2	7	110,090	61
1/8	4,838	11	1/8	2,201	32	1 3/4	850	1	15	87,476	68
1/4	2,204	11	1/4	1,032	35	3 3/4	716	1	31	75,434	84
1/2	1,004	11	1/2	544	33	7 3/4	688	1			
1	575	11	1	412	30	15 3/4	674	1			

Table 1: *Simplification ϵ's and run times in minutes*

8.1 Generating Correspondences

A true geometric hierarchy should contain not only representations of an object at various levels of detail, but also some correspondence information about the relationship between adjacent levels. These relationships are necessary for propagating local information from one level to the next. For instance, this information would be helpful for using the hierarchical geometric representation to perform radiosity calculations. It is also necessary for performing geometric interpolation between the models when using the levels of detail for rendering. Note that the envelope technique preserves silhouettes when rendering, so it is also a good candidate for alpha blending rather than geometric interpolation to smooth out transitions between levels of detail.

We can determine which elements of a higher level of detail surface are covered by an element of a lower level of detail representation by noting which fundamental prisms this element intersects. This is non-trivial only because of the bilinear patches that are the sides of a fundamental prism. We can approximate these patches by two or more triangles and then tetrahedralize each prism. Given this tetrahedralization of the envelope volume, it is possible to stab each edge of the lower level-of-detail model through the tetrahedrons to determine which ones they intersect, and thus which triangles are covered by each lower level-of-detail triangle.

8.2 Moving Vertices

The output mesh generated by either of the algorithms we have presented has the property that its set of vertices is a subset of the set of vertices of the original mesh. If we can afford to relax this constraint somewhat, we may be able to reduce the output size even further. If we allow the vertices to slide along their normal vectors, we should be able to simplify parts of the surface that might otherwise be impossible to simplify for some choices of epsilon. We are currently working on a goal-based approach to moving vertices within the envelope volume. For each vertex we want to remove, we slide its neighboring vertices along their normals to make them lie as closely as possible to a tangent plane of the original vertex. Intuitively, this should increase the likelihood of successfully removing the vertex. During this whole process, we must ensure that none of the neighboring triangles ever violates the envelopes. This approach should make it possible to simplify surfaces using smaller epsilons than previously possible. In fact, it may even enable us to use the original surface and a single envelope as our constraint surfaces rather than two envelopes. This is important for objects with areas of high maximal curvature, like thin cylinders.

9 Conclusion

We have outlined the notion of simplification envelopes and how they can be used for generation of multiresolution hierarchies for polygonal objects. Our approach guarantees non-self-intersecting approximations and allows the user to do adaptive approximations by simply editing the simplification envelopes (either manually or automatically) in the regions of interest. It allows for a global error tolerance, preservation of the input genus of the object, and preservation of sharp edges. Our approach requires only one user-specifiable parameter, allowing it to work on large collections of objects with no manual intervention if so desired. It is rotationally and translationally invariant, and can elegantly handle holes and bordered surfaces through the use of cylindrical tubes. Simplification envelopes are general enough to permit both simplification algorithms with good theoretical properties such as our global algorithm, as well as fast, practical, and robust implementations like our local algorithm. Additionally, envelopes permit easy generation of correspondences across several levels of detail.

10 Acknowledgements

Thanks to Greg Angelini, Jim Boudreaux, and Ken Fast at Electric Boat for the submarine model, Rich Riesenfeld and Elaine Cohen of the Alpha_1 group at the University of Utah for the rotor model, and the Stanford Computer Graphics Laboratory for the bunny and telephone models.Thanks to Carl Mueller, Marc Olano, and Bill Yakowenko for many useful suggestions, and to the rest of the UNC Simplification Group (Rui Bastos, Carl Erikson, Merlin Hughes, and David Luebke) for providing a great forum for discussing ideas. The funding for this work was provide by a Link Foundation Fellowship, Alfred P. Sloan Foundation Fellowship, ARO Contract P-34982-MA, ARO MURI grant DAAH04-96-1-0013, NSF Grant CCR-9319957, NSF Grant CCR-9301259, NSF Career Award CCR-9502239, ONR Contract N00014-94-1-0738, ARPA Contract DABT63-93-C-0048, NSF/ARPA Center for Computer Graphics and Scientific Visualization, NIH Grant RR02170, an NYI award with matching funds from Xerox Corp, and a U.S.-Israeli Binational Science Foundation grant.

References

[1] P. Agarwal and S. Suri. Surface approximation and geometric partitions. In *Proceedings Fifth Symposium on Discrete Algorithms*, pages 24–33, 1994.

[2] H. Brönnimann and M. Goodrich. Almost optimal set covers in finite VC-dimension. In *Proceedings Tenth ACM Symposium on Computational Geometry*, pages 293–302, 1994.

[3] K. L. Clarkson. Algorithms for polytope covering and approximation. In *Proc. 3rd Workshop Algorithms Data Struct.*, Lecture Notes in Computer Science, 1993.

[4] M. Cosman and R. Schumacker. System strategies to optimize CIG image content. In *Proceedings of the Image II Conference*, Scottsdale, Arizona, June 10–12 1981.

[5] G. Das and D. Joseph. The complexity of minimum convex nested polyhedra. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 296–301, 1990.

[6] M. J. DeHaemer, Jr. and M. J. Zyda. Simplification of objects rendered by polygonal approximations. *Computers & Graphics*, 15(2):175–184, 1991.

[7] T. D. DeRose, M. Lounsbery, and J. Warren. Multiresolution analysis for surface of arbitrary topological type. Report 93-10-05, Department of Computer Science, University of Washington, Seattle, WA, 1993.

[8] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. *Computer Graphics: Proceedings of SIGGRAPH'95*, pages 173–182, 1995.

[9] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 247–254, August 1993.

[10] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Computer Graphics: Proceedings of SIGGRAPH 1993*, pages 231–238. ACM SIGGRAPH, 1993.

[11] T. He, L. Hong, A. Kaufman, A. Varshney, and S. Wang. Voxel-based object simplification. In G. M. Nielson and D. Silver, editors, *IEEE Visualization '95 Proceedings*, pages 296–303, 1995.

[12] P. Heckbert and M. Garland. Multiresolution modeling for fast rendering. *Proceedings of Graphics Interface*, 1994.

[13] P. Hinker and C. Hansen. Geometric optimization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings Visualization '93*, pages 189–195, October 1993.

[14] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 19–26, August 1993.

[15] A. D. Kalvin and R. H. Taylor. Superfaces: Polyhedral approximation with bounded error. Technical Report RC 19135 (#82286), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10958, 1993.

[16] J. Mitchell and S. Suri. Separation and approximation of polyhedral surfaces. In *Proceedings of 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 296–306, 1992.

[17] Kevin J. Renze and J. H. Oliver. Generalized surface and volume decimation for unstructured tessellated domains. In *Proceedings of SIVE'95*, 1995.

[18] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993.

[19] H. E. Rushmeier, C. Patterson, and A. Veerasamy. Geometric simplification for indirect illumination calculations. In *Proceedings Graphics Interface '93*, pages 227–236, 1993.

[20] F. J. Schmitt, B. A. Barsky, and W. Du. An adaptive subdivision method for surface-fitting from sampled data. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):179–188, 1986.

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

[22] G. Taubin. A signal processing approach to fair surface design. In *Proc. of ACM Siggraph*, pages 351–358, 1995.

[23] G. Turk. Re-tiling polygonal surfaces. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 55–64, July 1992.

[24] A. Varshney. Hierarchical geometric approximations. Ph.D. Thesis TR-050-1994, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1994.

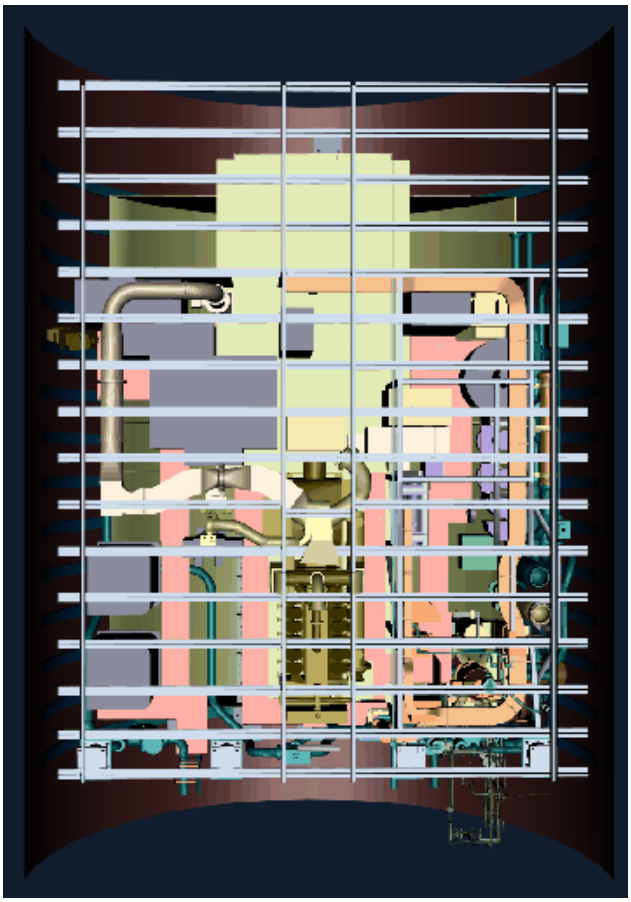


Figure 10: *Looking down into the auxiliary machine room (AMR) of a submarine model. This model contains nearly 3,000 objects, for a total of over half a million triangles. We have simplified over 2,600 of these objects, for a total of over 430,000 triangles.*

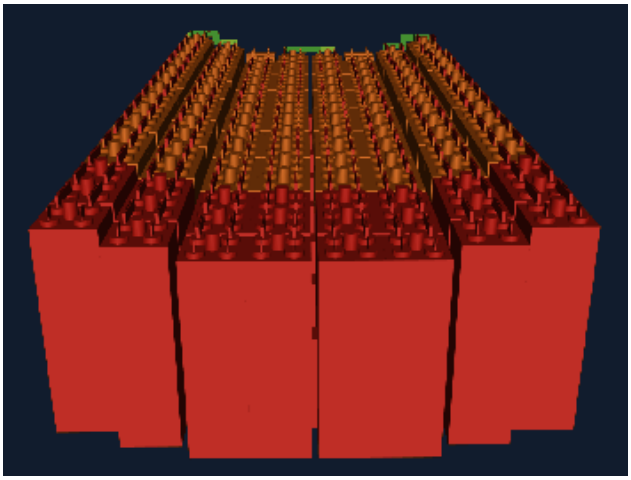
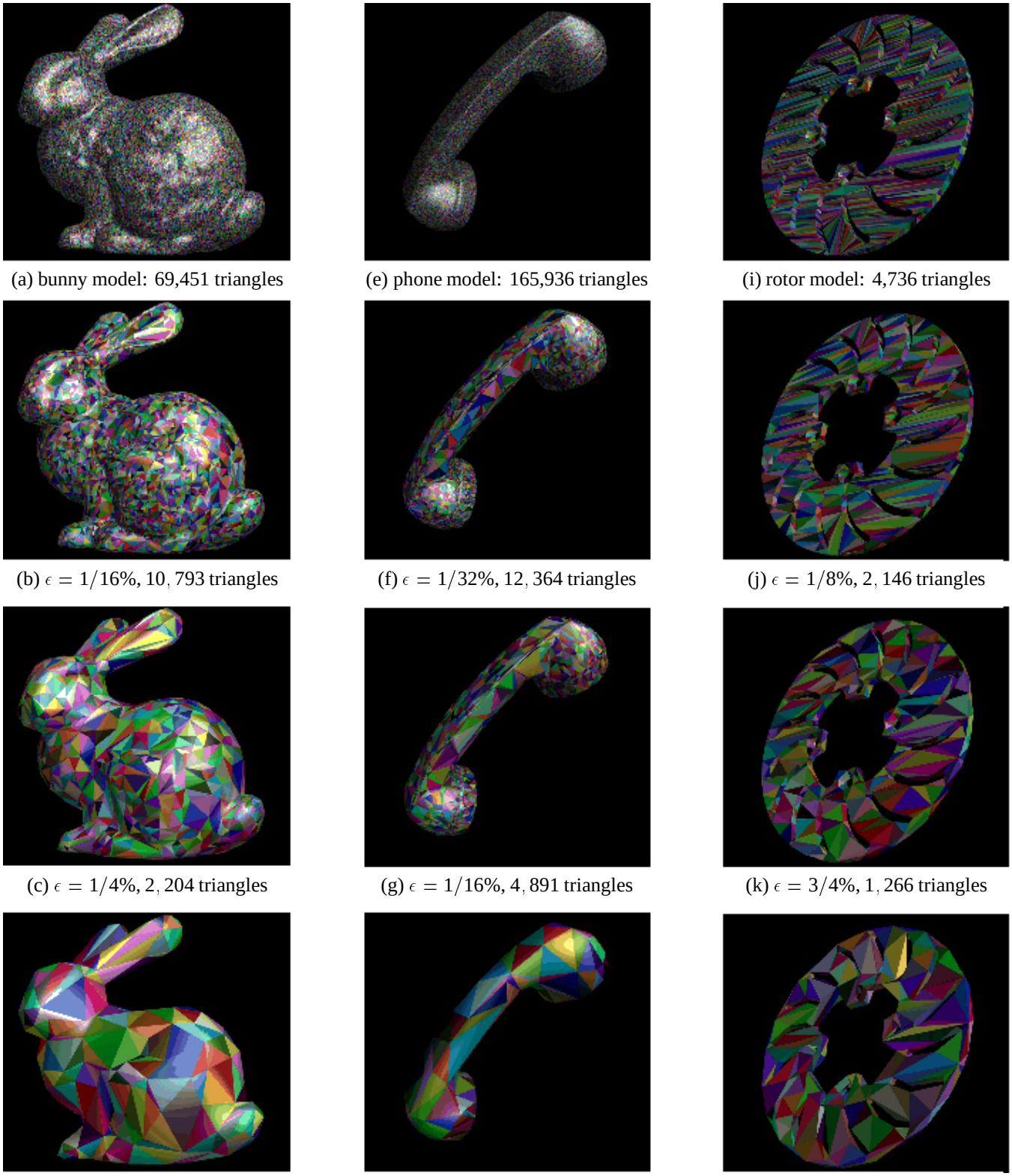


Figure 11: *An array of batteries from the AMR. All parts but the red are simplified representations. At full resolution, this array requires 87,000 triangles. At this distance, allowing 4 pixels of error in screen space, we have reduced it to 45,000 triangles.*



(a) bunny model: 69,451 triangles

(e) phone model: 165,936 triangles

(i) rotor model: 4,736 triangles

(b) $\epsilon = 1/16\%$, $10,793$ triangles

(f) $\epsilon = 1/32\%$, $12,364$ triangles

(j) $\epsilon = 1/8\%$, $2,146$ triangles

(c) $\epsilon = 1/4\%$, $2,204$ triangles

(g) $\epsilon = 1/16\%$, $4,891$ triangles

(k) $\epsilon = 3/4\%$, $1,266$ triangles

(d) $\epsilon = 1\%$, 575 triangles

(h) $\epsilon = 1\%$, 412 triangles

(l) $\epsilon = 3\ 3/4\%$, 716 triangles

Figure 12: *Level-of-detail hierarchies for three models. The approximation distance, ϵ, is taken as a percentage of the bounding box diagonal.*