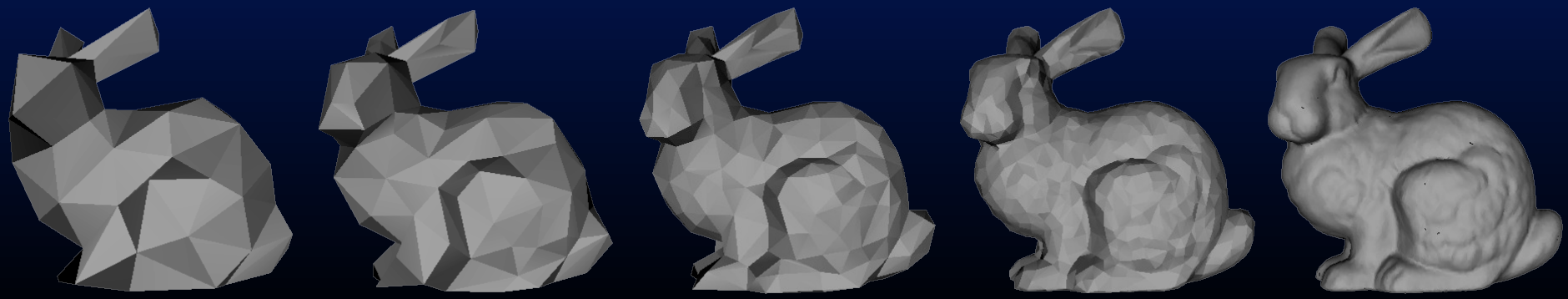
SAN ANTONIO

SIGGRAPH

2002

Advanced Issues In Level Of Detail



Course Introduction

- *Level of detail* (*LOD*) methods provide a powerful way to manage scene complexity
- A standard tool for the graphics developer to control rendering speed
- This course will adress advanced issues in using and developing LOD algorithms, with a focus on polygonal mesh simplification

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

Course Topics

- Generation

  LOD frameworks & creation

- Theory

  Measuring & controlling fidelity

- Applications

  Important real-world applications

Course Schedule

8:30	Welcome, Introductions	Luebke
8:50	Frameworks	Luebke
	Discrete, continuous, & view-dependent LOD	
10:15	Break	
10:30	Algorithms	Varshney, Cohen
	Algorithms and approaches for simplification	
	Appearance-preserving simplification	
12:15	Lunch	

Course Schedule

1:30	Fidelity	Cohen, Reddy, Watson

Measuring geometric and attribute error
Understanding and applying visual perception
Balancing fidelity and performance

3:15	Break	
3:30	Applications	Huebner, Reddy, Watson

Gaming optimizations
Terrain visualization
Out-of-core simplification

5	Conclusion	Luebke

Speakers
In Order of Appearance

- David Luebke, University of Virginia
- Jon Cohen, Johns Hopkins University
- Amitabh Varshney, University of Maryland
- Martin Reddy, SRI International
- Ben Watson, Northwestern University
- Rob Huebner, Nihilistic Software

Frameworks for LOD

- I will discuss basic LOD frameworks:
  - Discrete LOD: the traditional approach
  - Continuous LOD: encoding a continuous spectrum of detail from coarse to fine
  - View-dependent LOD: adjusting detail across the model in response to viewpoint
- I will focus on view-dependent LOD

Frameworks for 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?*

Motivation:
Preaching To The Choir

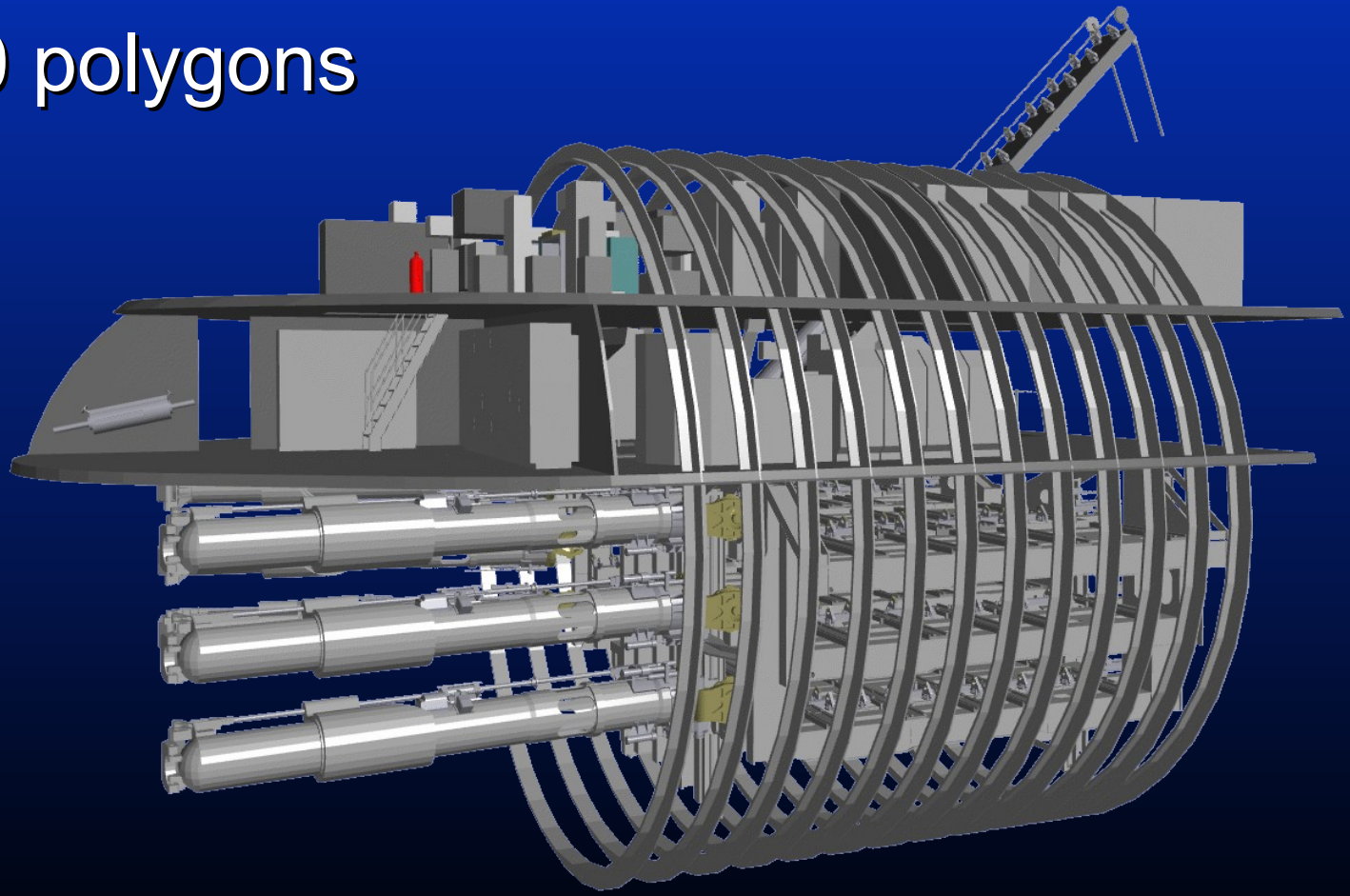
- Interactive rendering of large-scale geometric datasets is important
  - Scientific and medical visualization
  - Architectural and industrial CAD
  - Training (military and otherwise)
  - Entertainment

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…

Big Models:
Submarine Torpedo Room

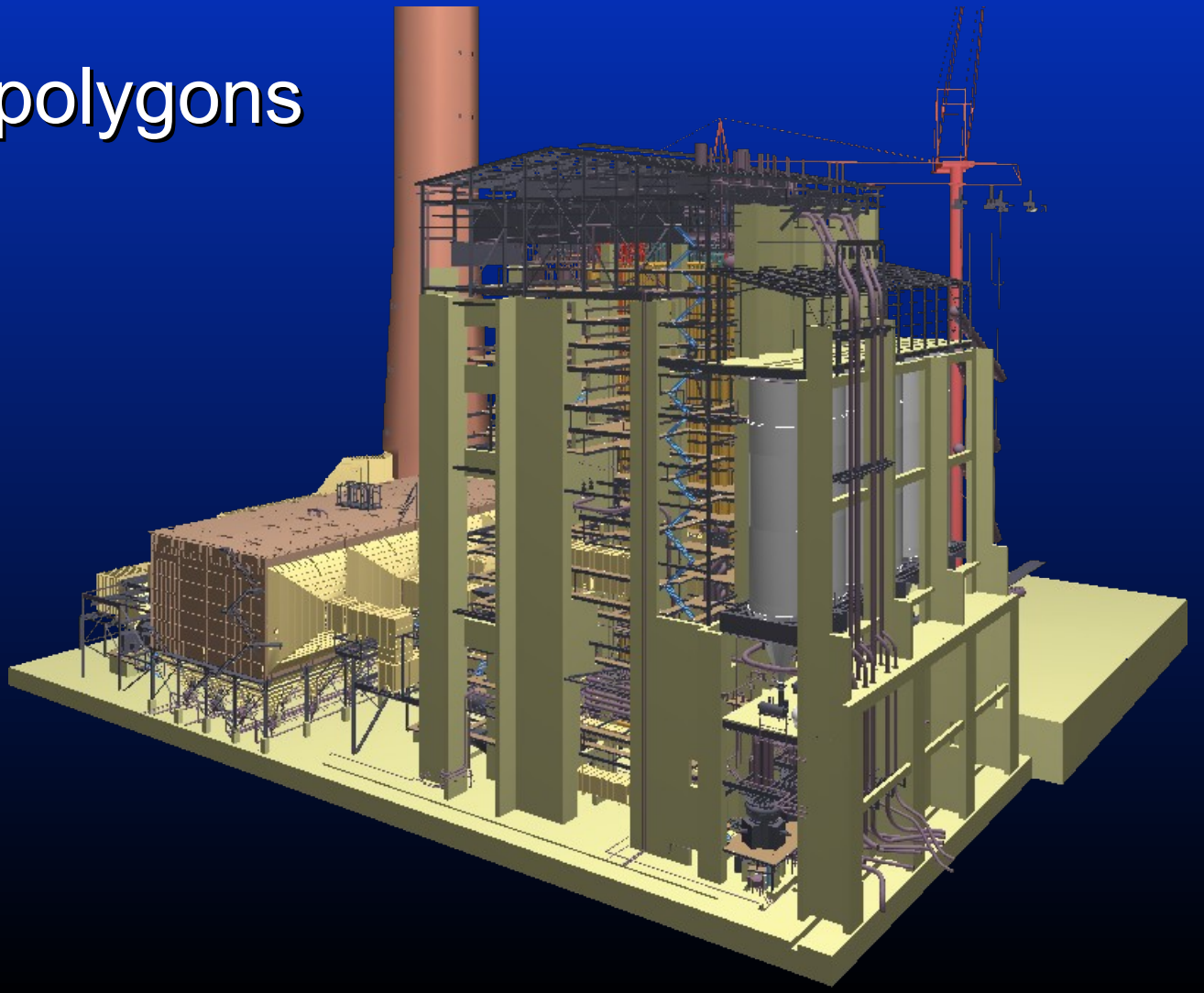
- 700,000 polygons



Courtesy General Dynamics, Electric Boat Div.

Big Models:
Coal-fired Power Plant

- 13 million polygons



(Anonymous)

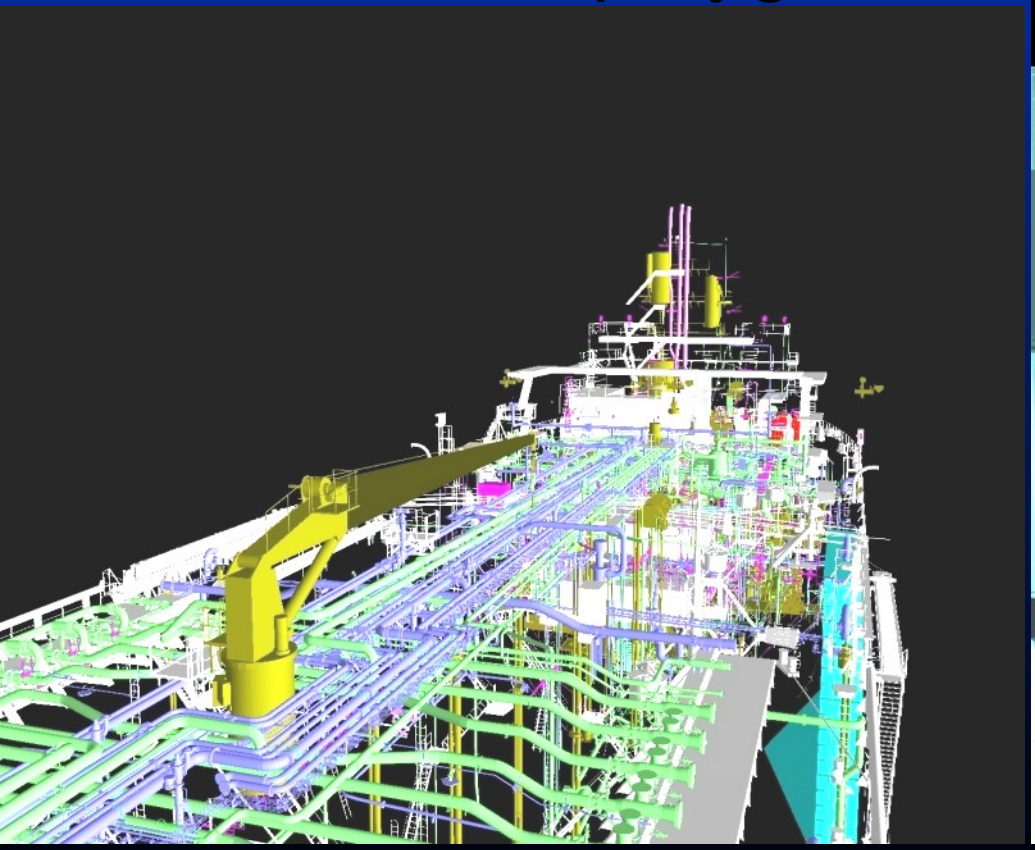
Big Models:
Plant Ecosystem Simulation

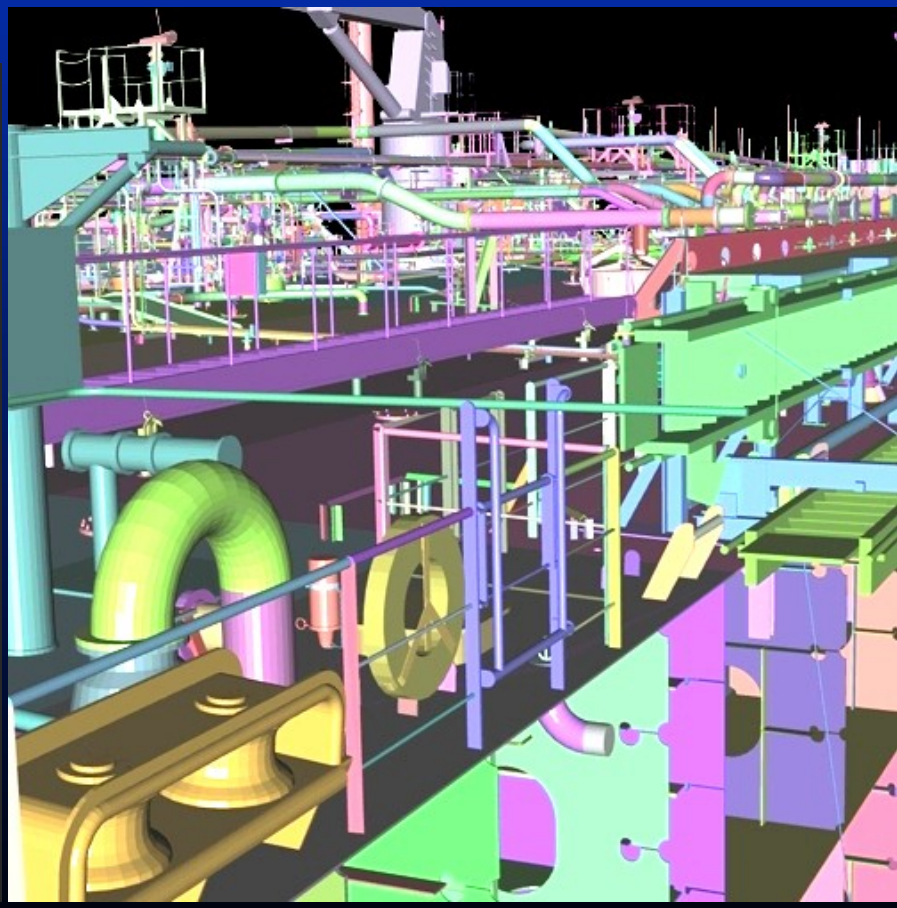
- 16.7 million polygons (sort of)

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

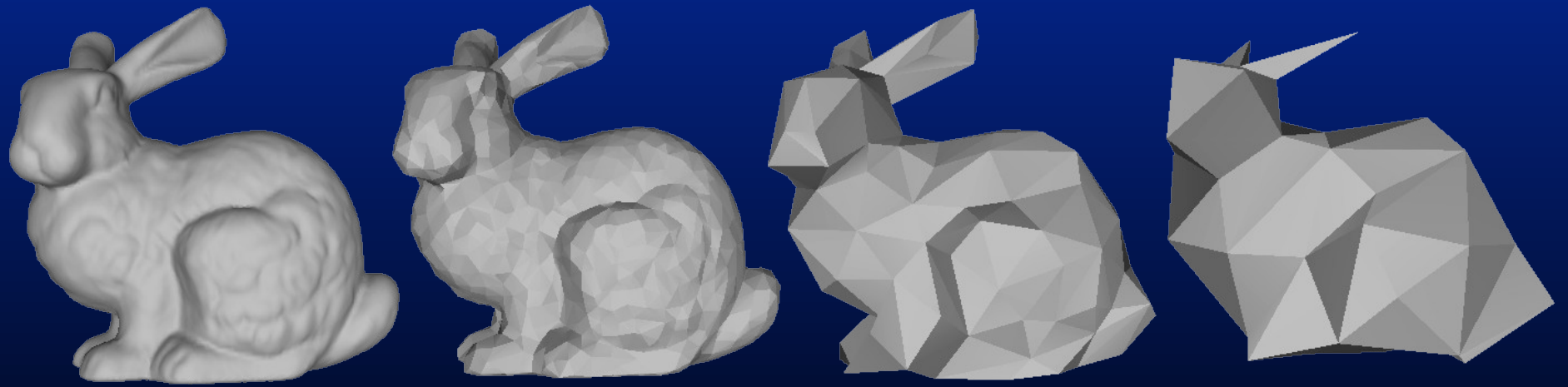
Courtesy Digital Michelangelo Project, Stanford University

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, …

Level of Detail: Traditional Approach

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

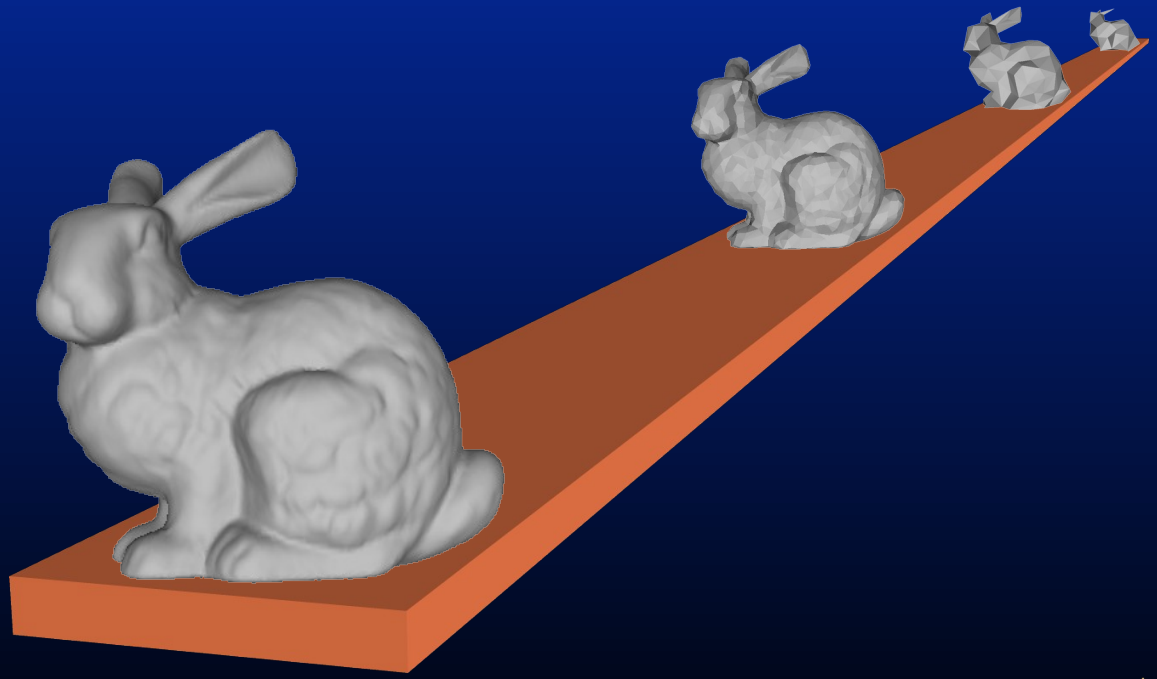


69,451 polys 2,502 polys 251 polys 76 polys

Courtesy Stanford 3D Scanning Repository

Level of Detail: Traditional Approach

- Distant objects use coarser LODs:



Traditional Approach:
Discrete 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 *Discrete LOD*

Discrete LOD:
Advantages

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

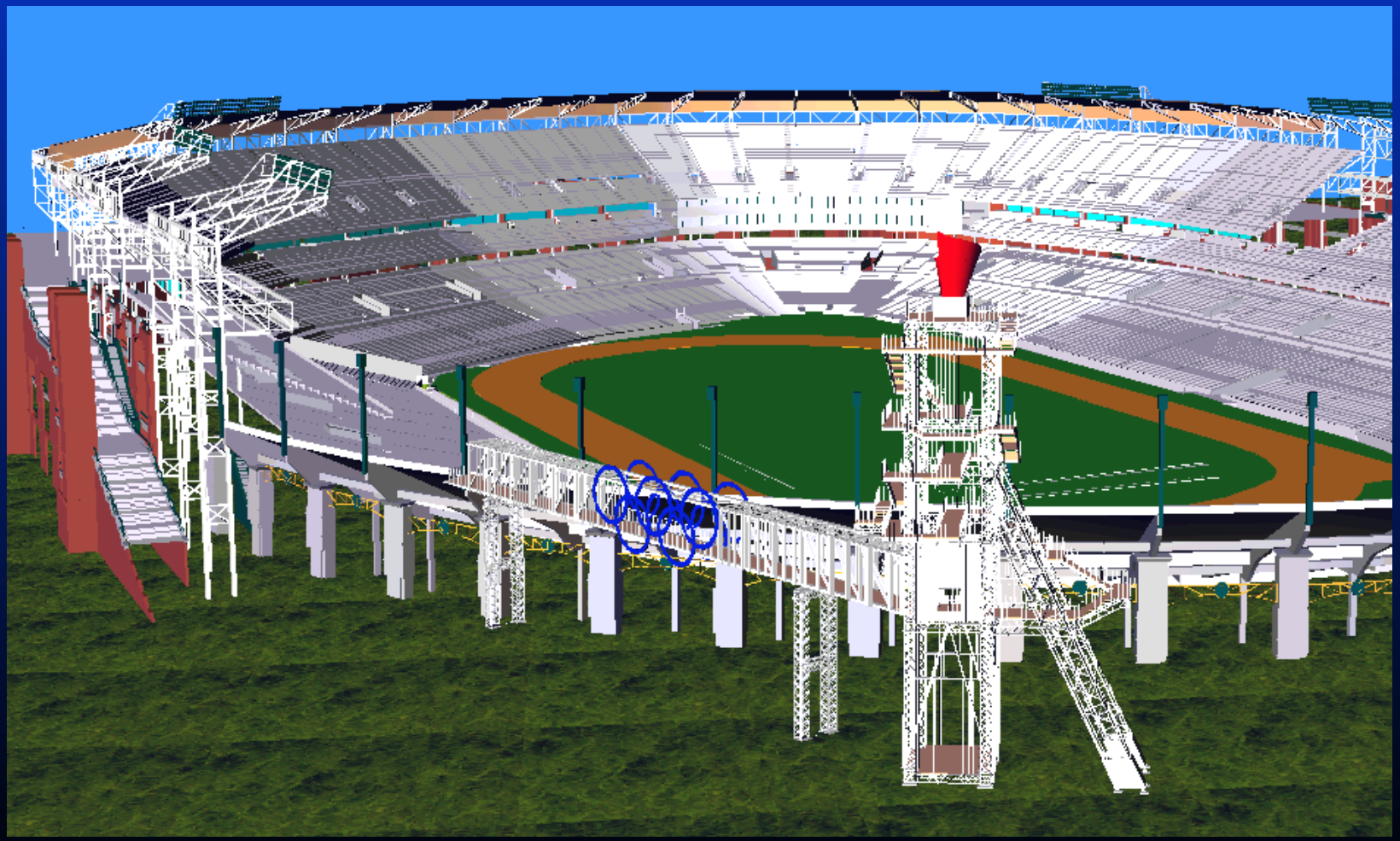
Discrete LOD:
Advantages

- 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)

Discrete LOD: Disadvantages

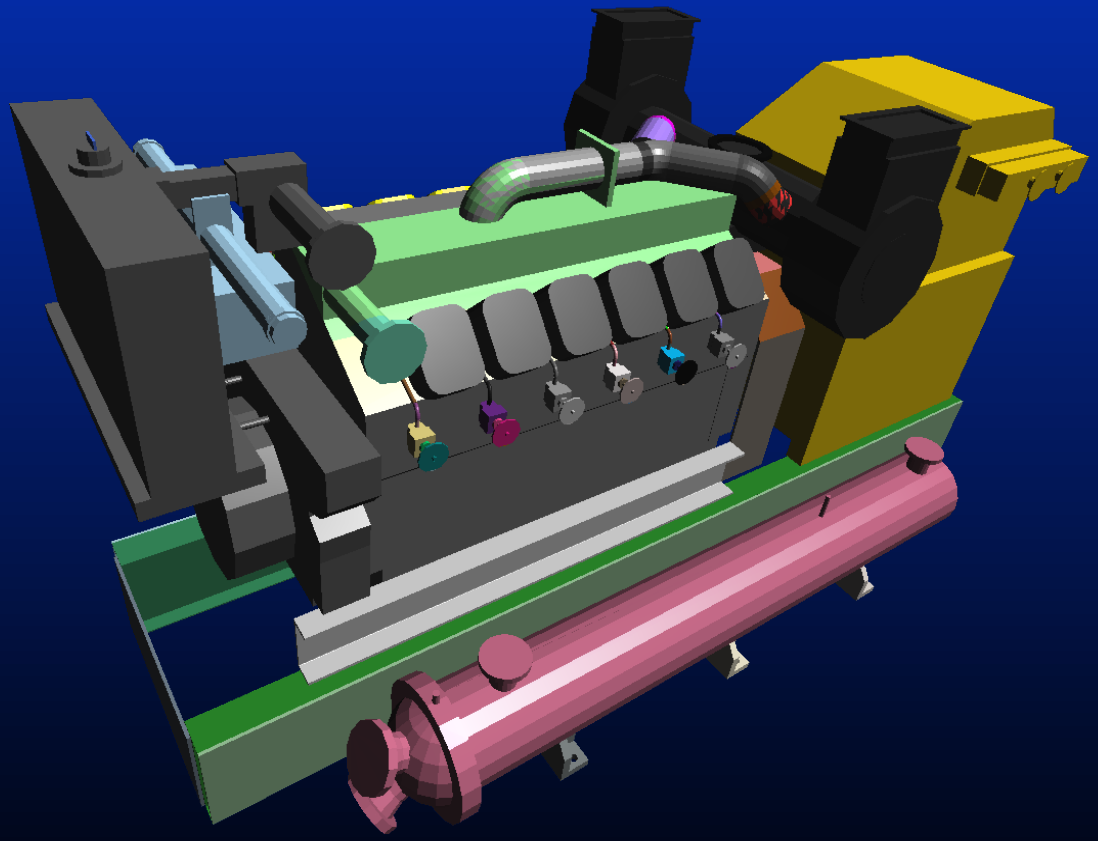
- So why use anything but discrete LOD?
- Answer: sometimes discrete LOD not suited for *drastic simplification*
- Some problem cases:
  - Terrain flyovers
  - Volumetric isosurfaces
  - Super-detailed range scans
  - Massive CAD models

Drastic Simplification:
The Problem With Large Objects



Courtesy IBM and ACOG

Drastic Simplification:
The Problem With Small Objects



Courtesy Electric Boat

Drastic Simplification

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

Continuous Level of Detail

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

Continuous 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

Continuous LOD: Advantages

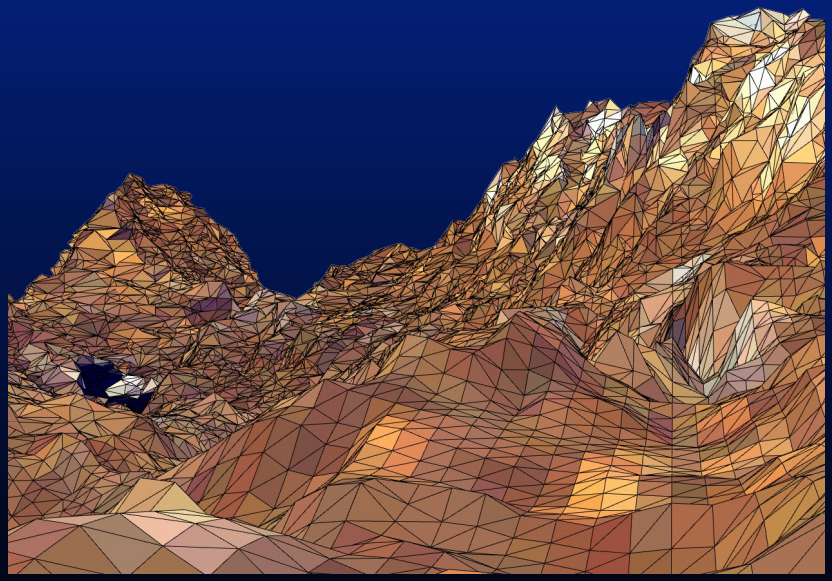
- Better granularity → smoother transitions
  - Switching between traditional LODs can introduce visual “popping” effect
  - Continuous LOD can adjust detail gradually and incrementally, reducing visual pops
    - Can even *geomorph* the fine-grained simplification operations over several frames to eliminate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

Continuous LOD: Advantages

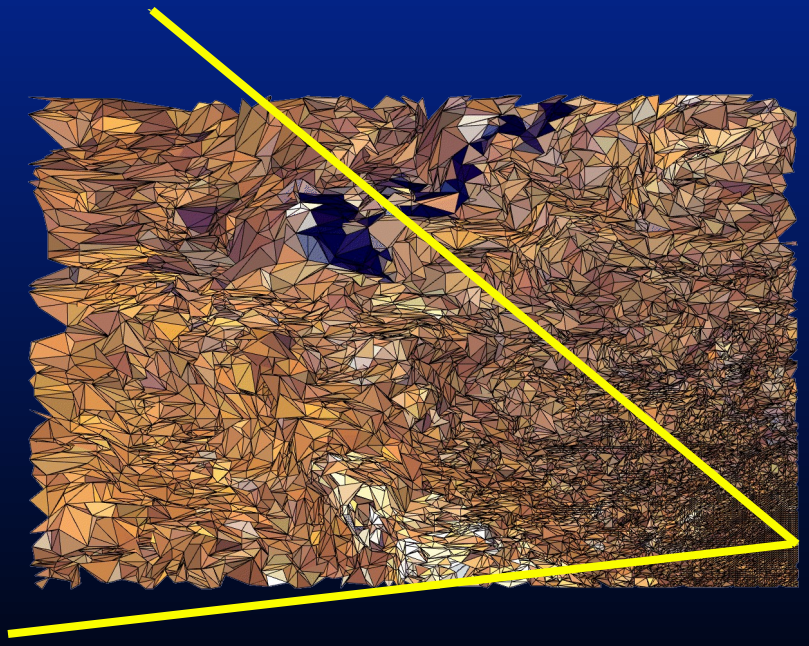
- Supports progressive transmission
  - *Progressive Meshes [Hoppe 97]*
  - *Progressive Forest Split Compression [Taubin 98]*
- Leads to *view-dependent LOD*
  - Use current view parameters to select best representation *for the current view*
  - Single objects may thus span several levels of detail

View-Dependent LOD: Examples

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



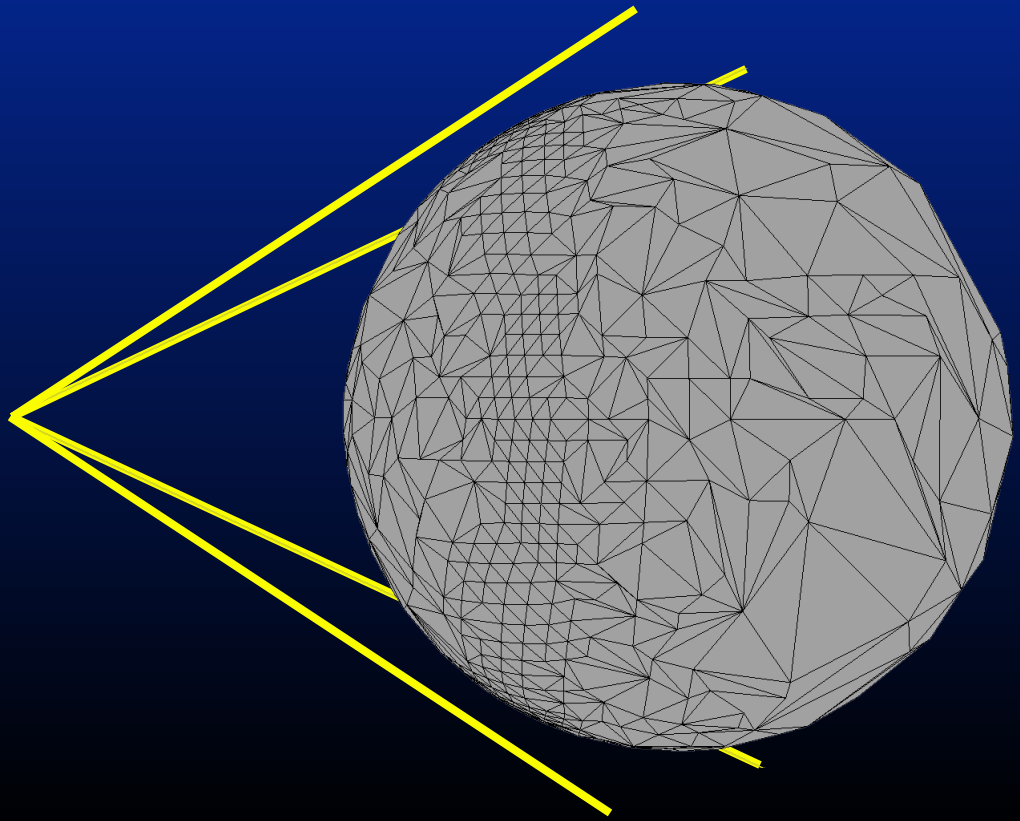
View from eyepoint



Birds-eye view

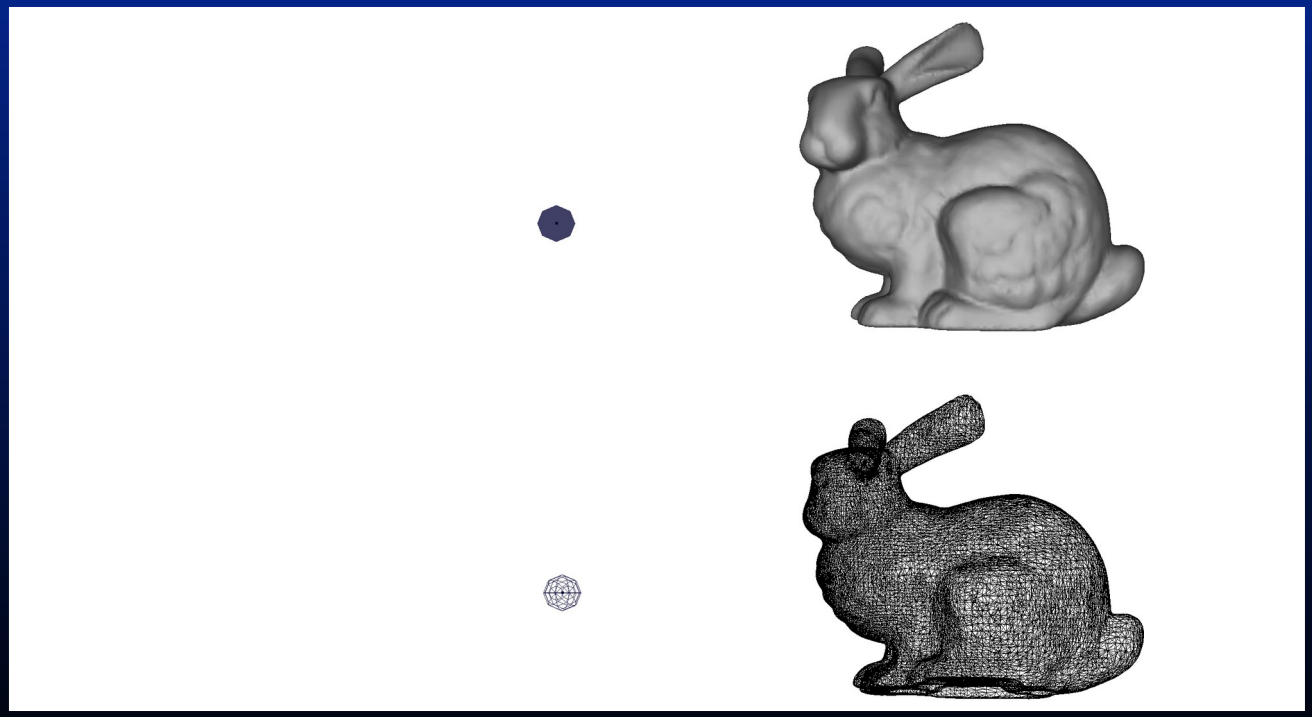
View-Dependent LOD: Examples

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



View-Dependent LOD: Examples

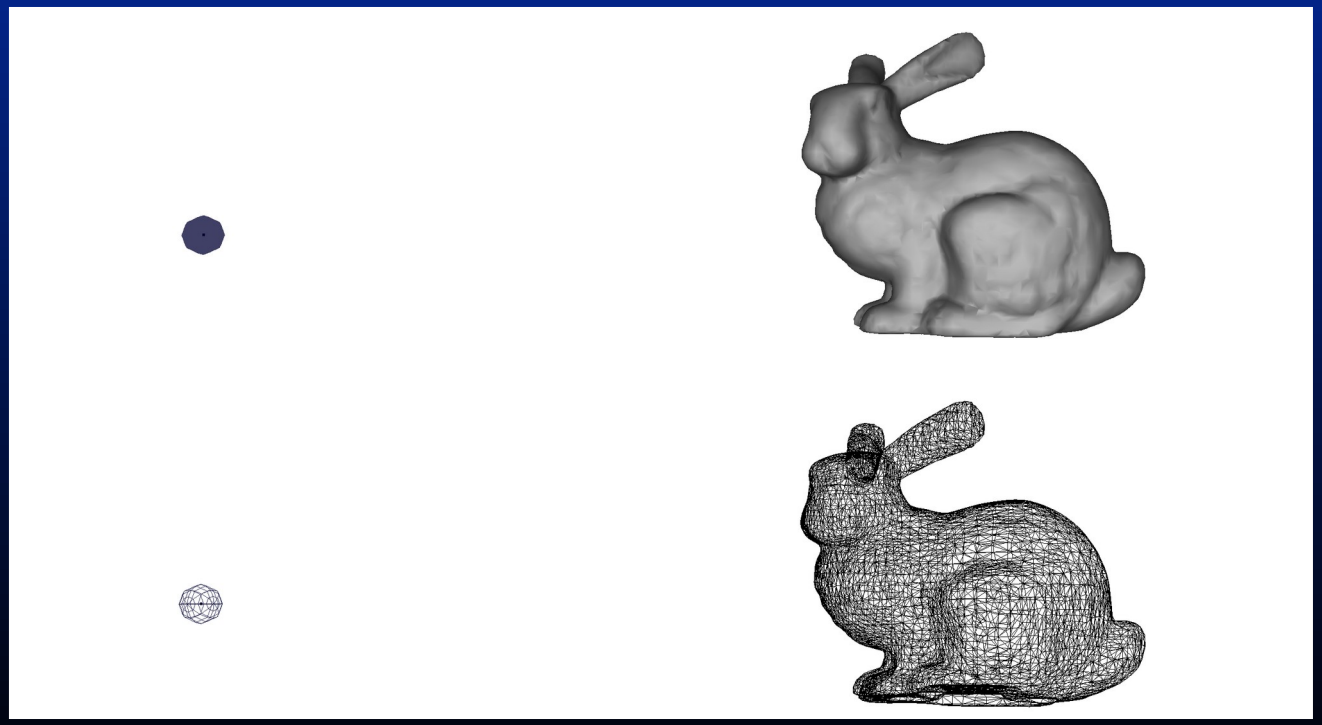
- Show more detail where the user is looking than in their peripheral vision:



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

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

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 discrete 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
- Amitabh will give his take on some related issues later

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

Data Structures

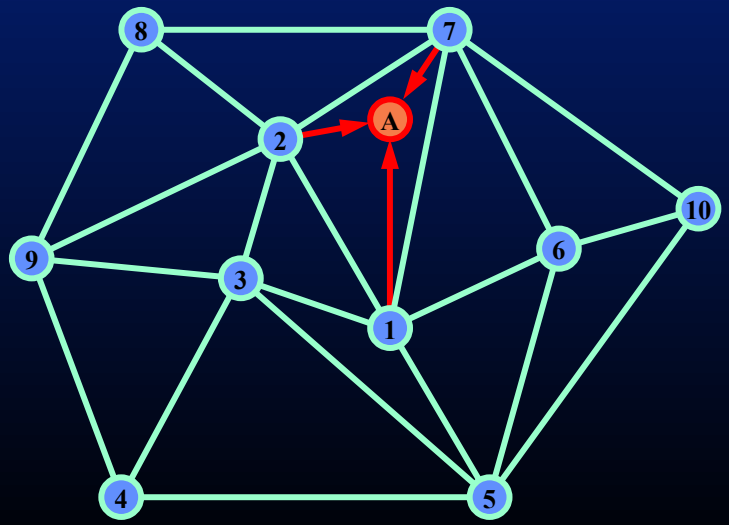
- The *vertex tree*
  - Represents the entire model
  - Hierarchy of all vertices in model
  - Queried each frame for updated scene
- The *active triangle list*
  - Represents the current simplification
  - List of triangles to be displayed
  - Triangles added and deleted by operations on vertex tree

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*

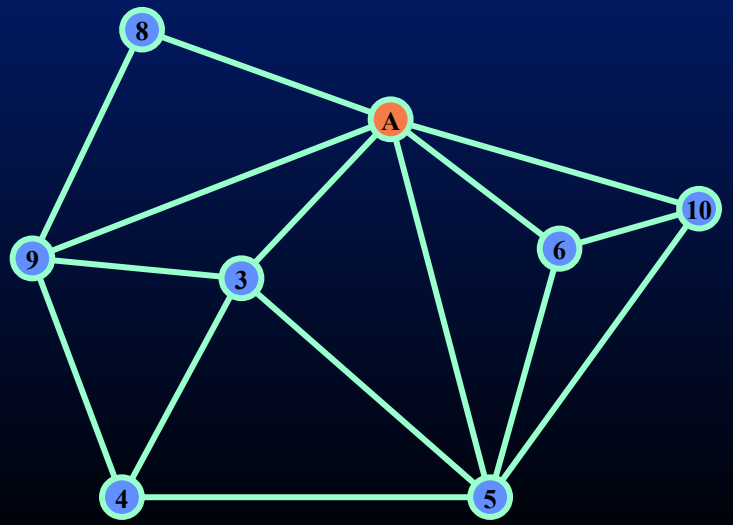
The Vertex Tree:
Folding And Unfolding

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

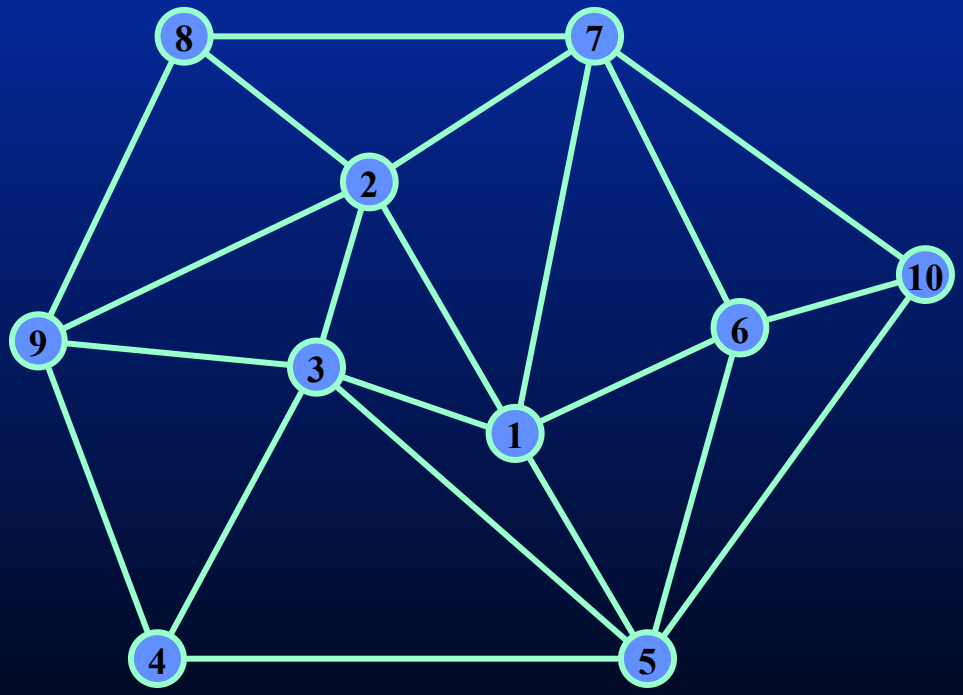


Fold Node A

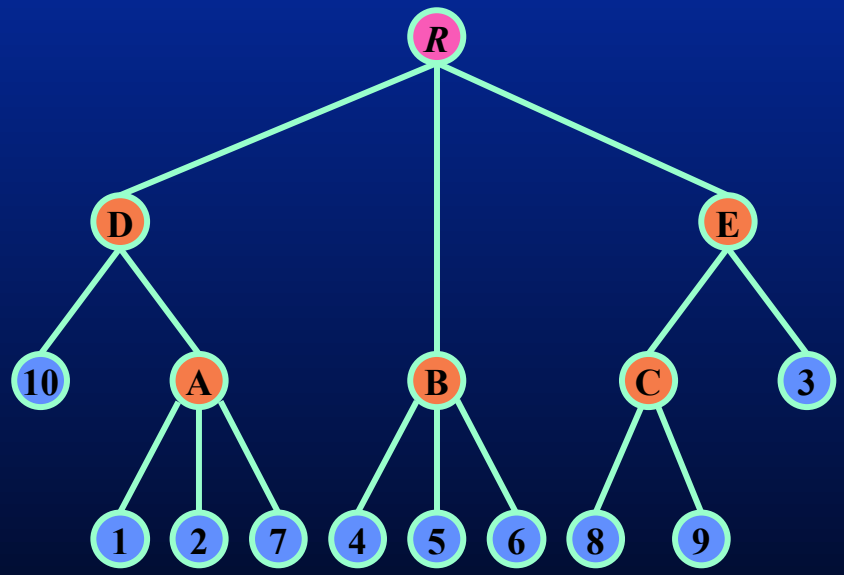
Unfold Node A



Vertex Tree Example

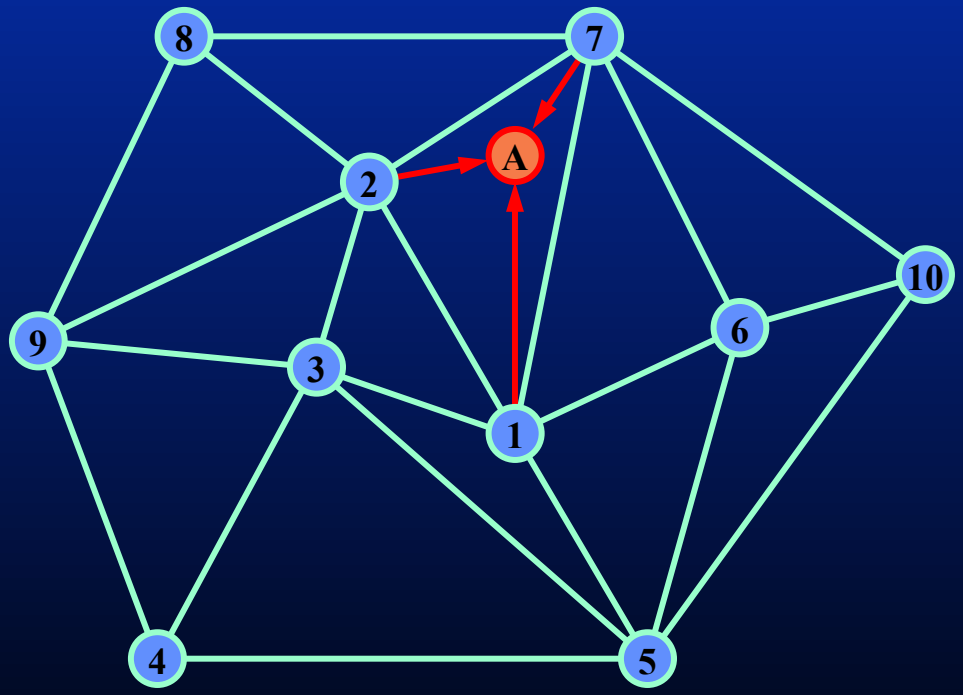


Triangles in active list

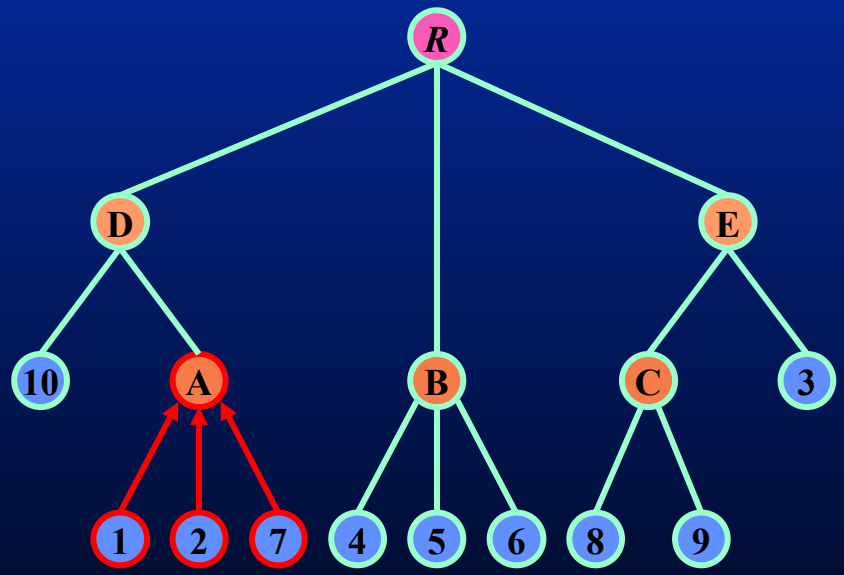


Vertex tree

Vertex Tree Example

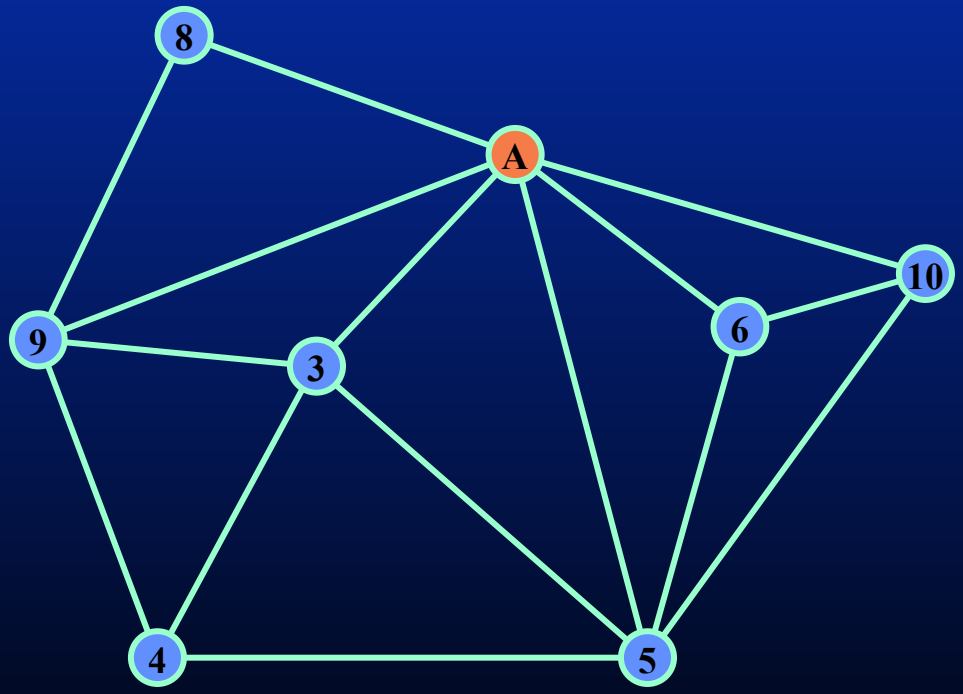


Triangles in active list

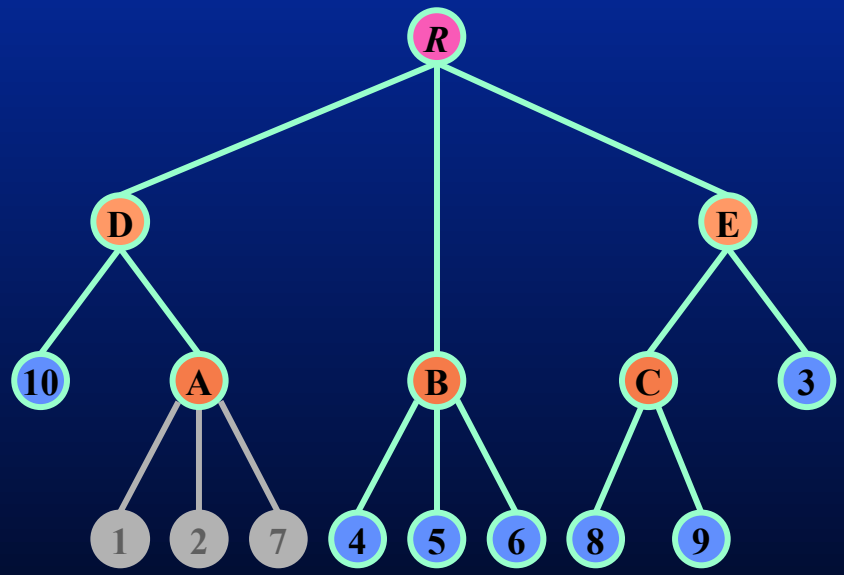


Vertex tree

Vertex Tree Example

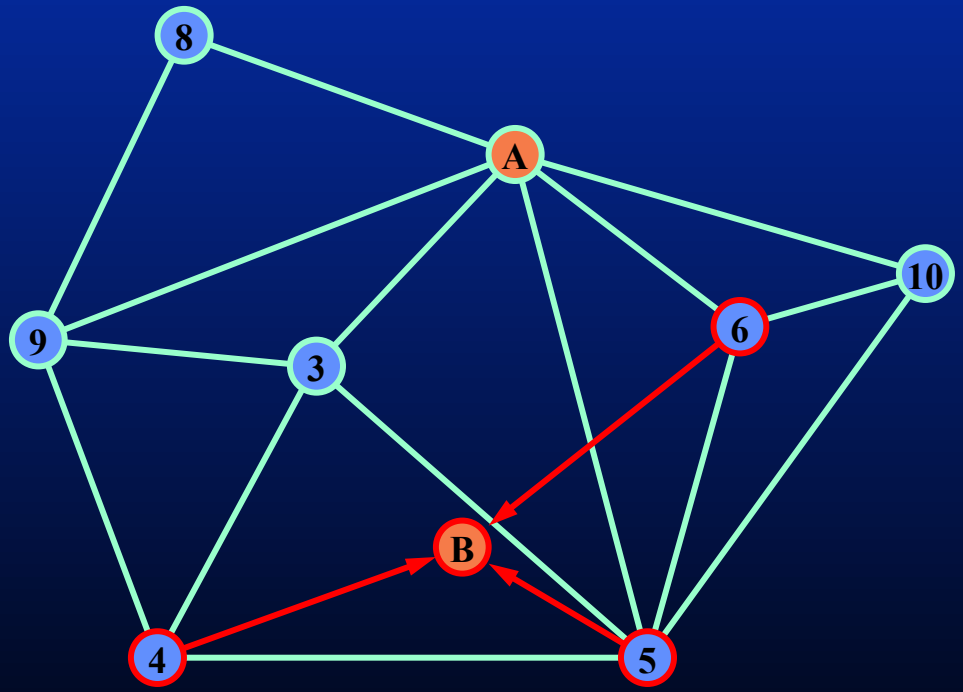


Triangles in active list

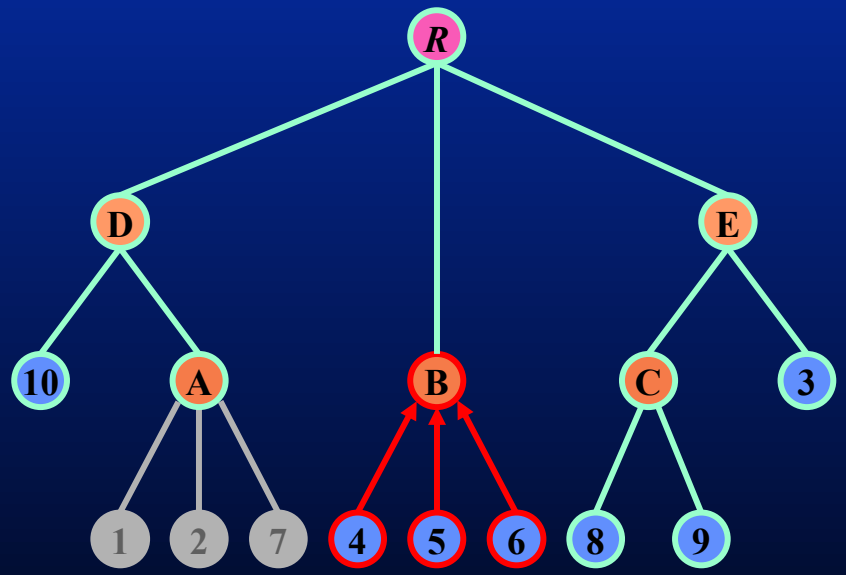


Vertex tree

Vertex Tree Example

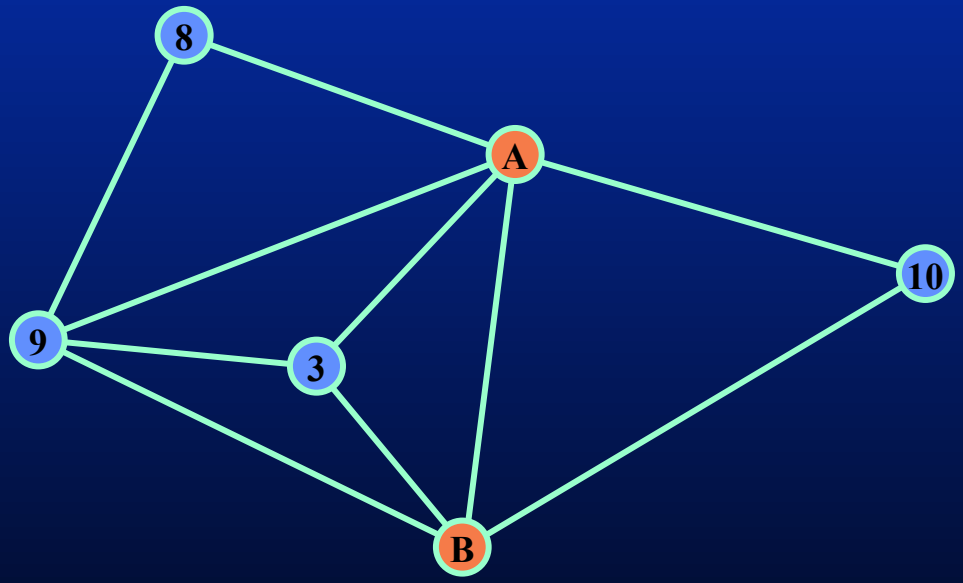


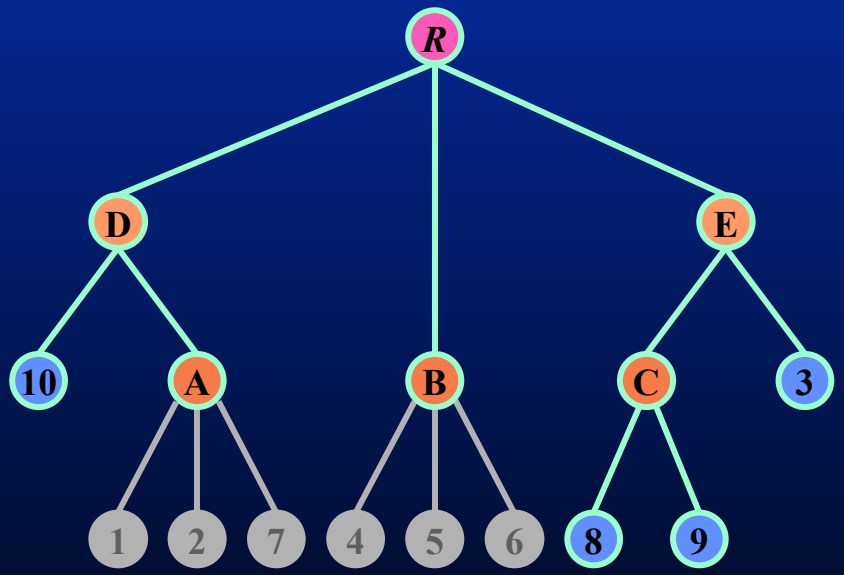
Triangles in active list



Vertex tree

Vertex Tree Example

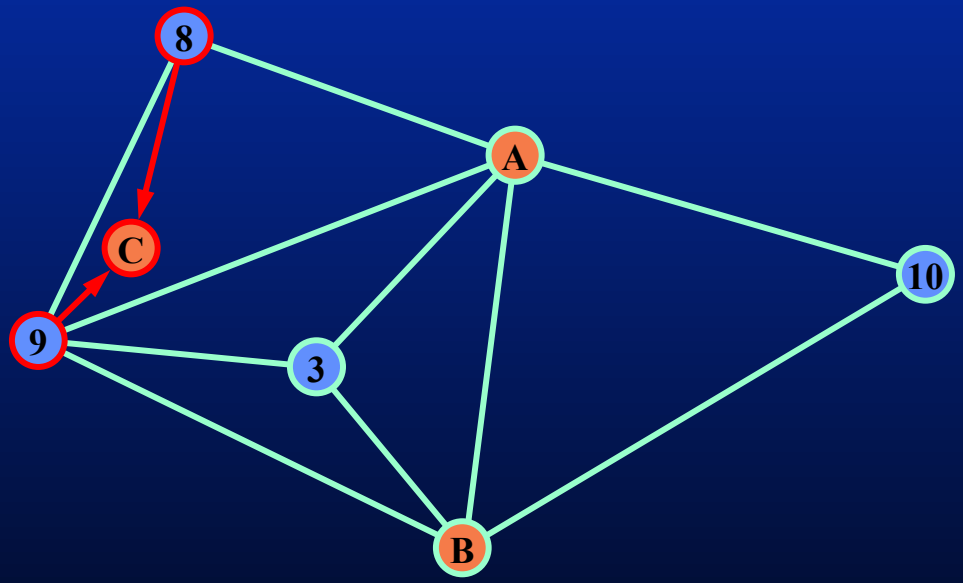


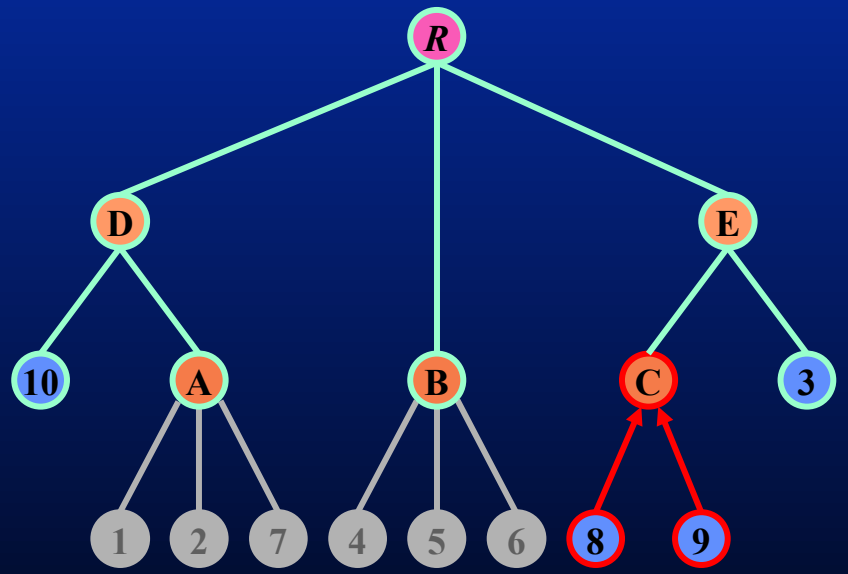


Triangles in active list

Vertex tree

Vertex Tree Example

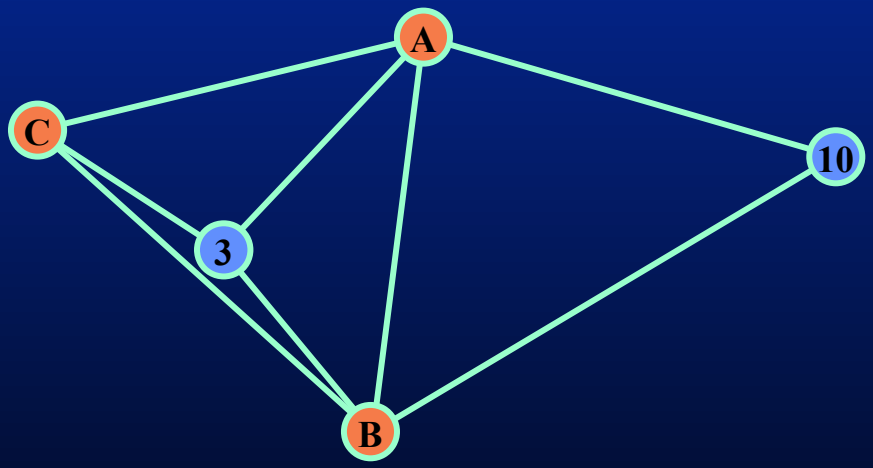


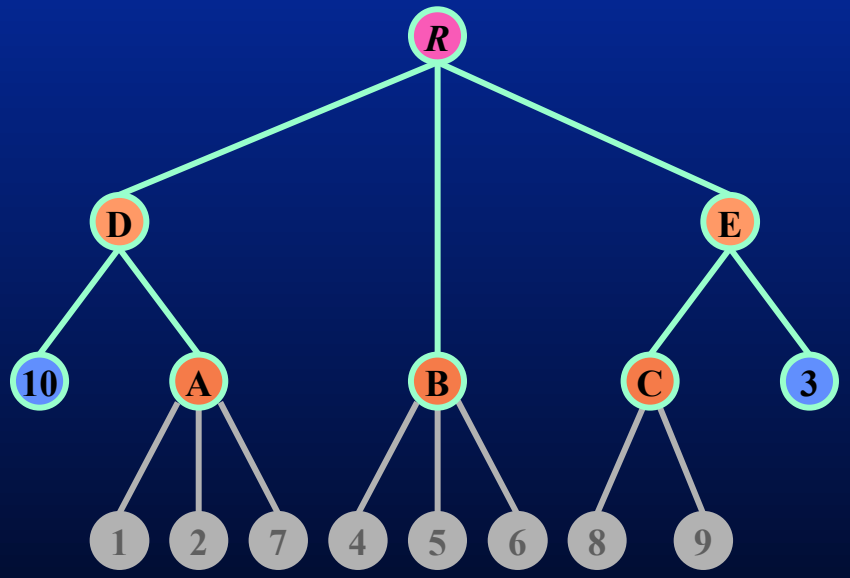


Triangles in active list

Vertex tree

Vertex Tree Example

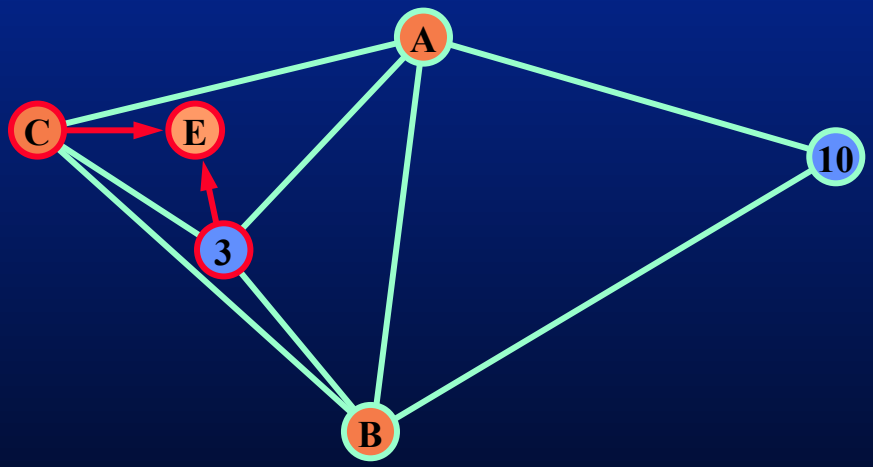


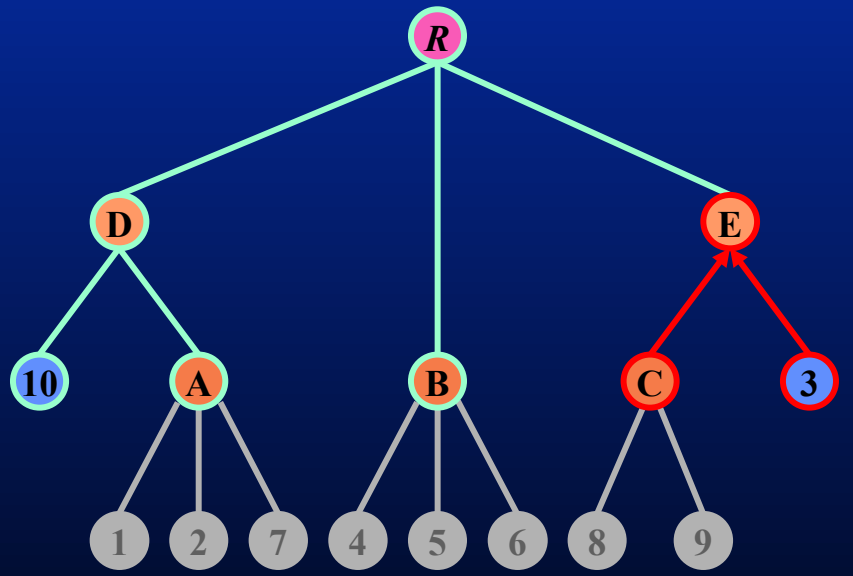


Triangles in active list

Vertex tree

Vertex Tree Example

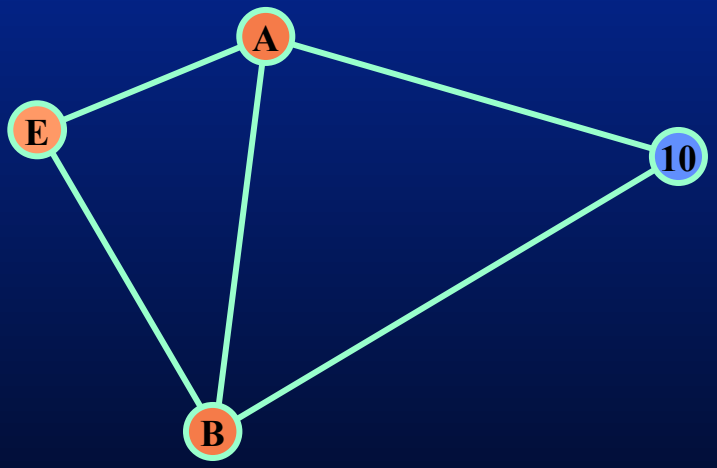


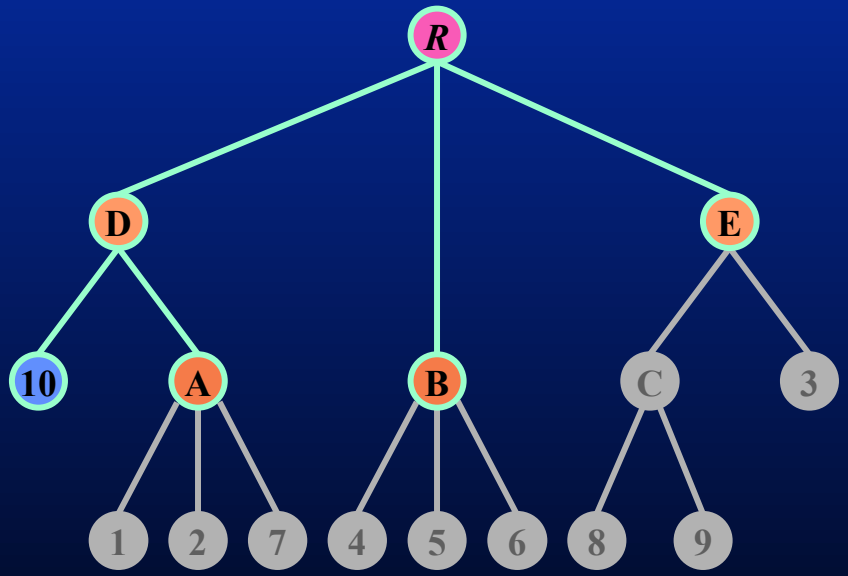


Triangles in active list

Vertex tree

Vertex Tree Example

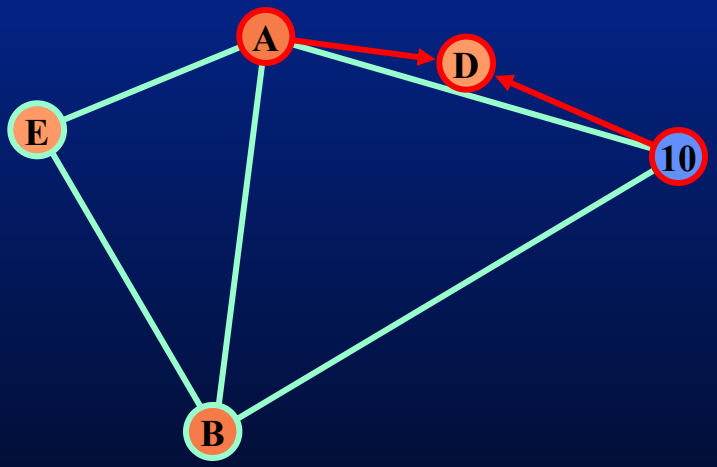


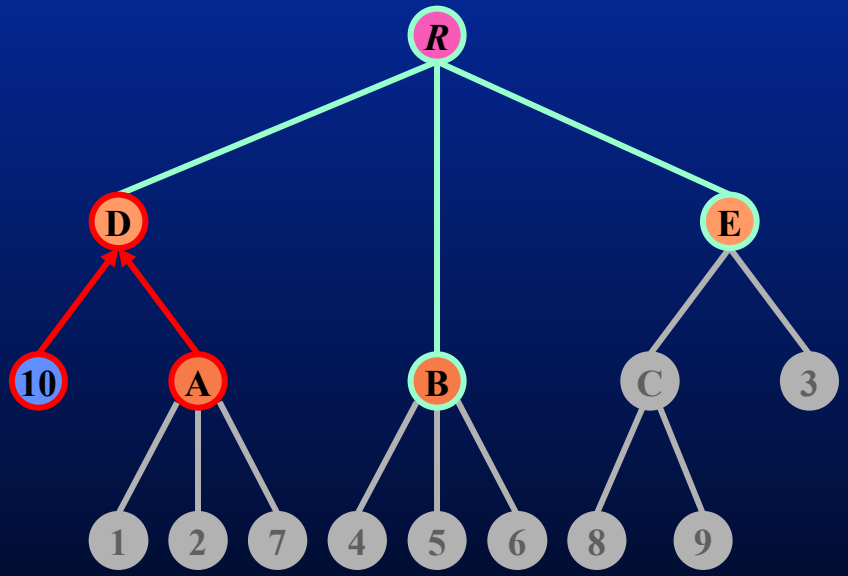


Triangles in active list

Vertex tree

Vertex Tree Example

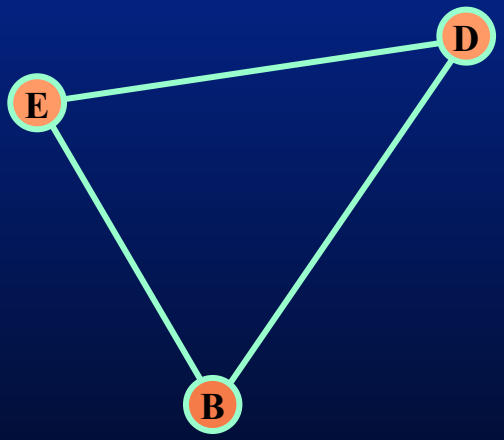


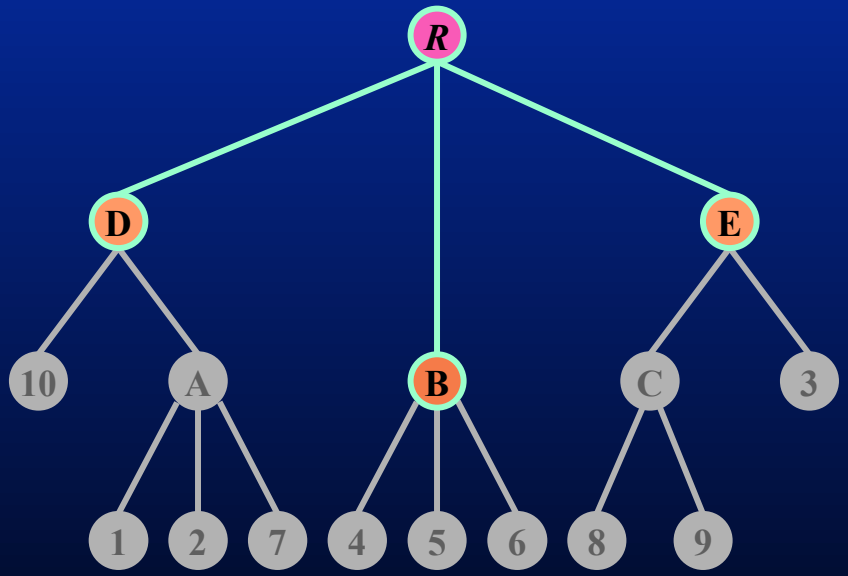


Triangles in active list

Vertex tree

Vertex Tree Example

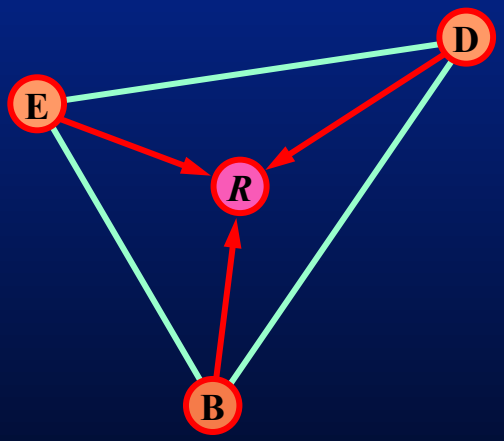


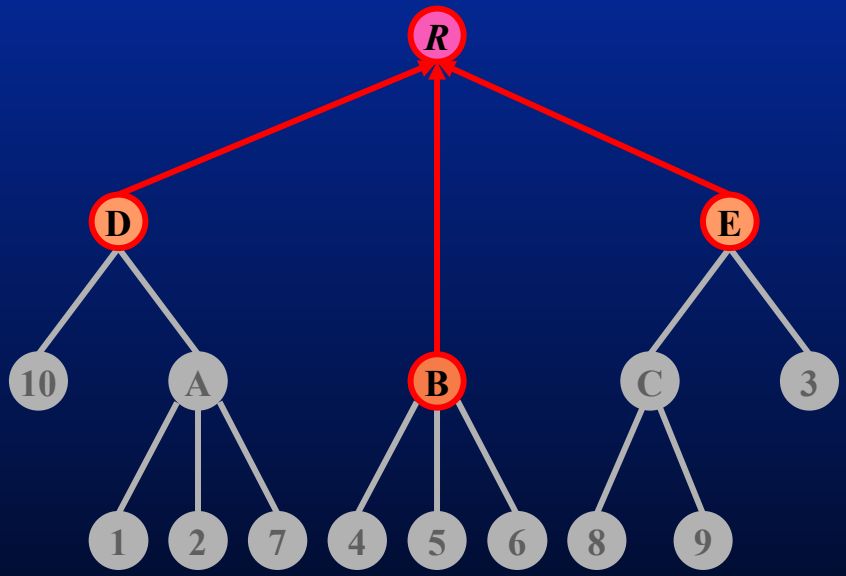


Triangles in active list

Vertex tree

Vertex Tree Example

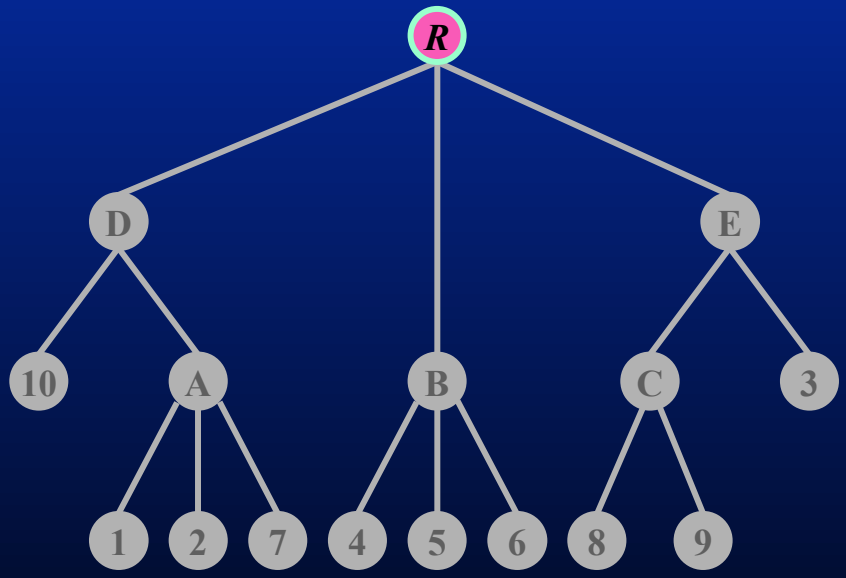




Triangles in active list

Vertex tree

Vertex Tree Example

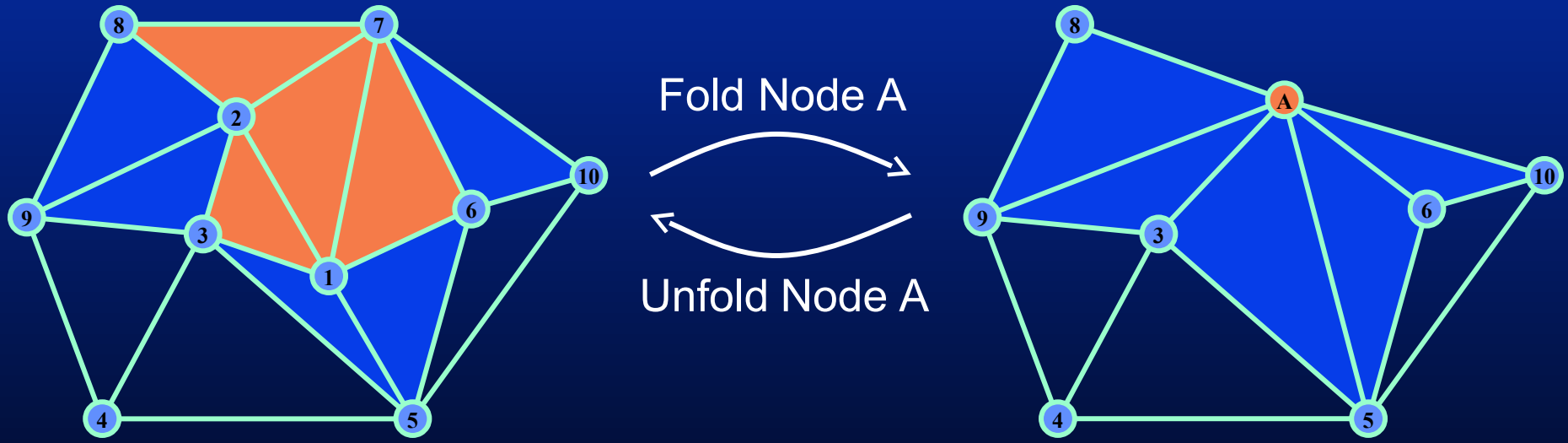


Triangles in active list

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

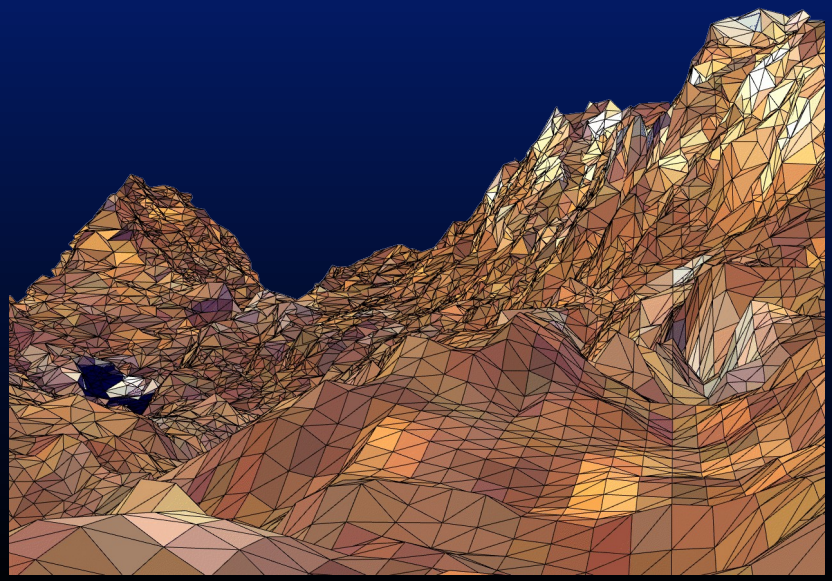
- The *key observation*:
  - Each node's subtris can be computed offline to be accessed quickly at run time
  - Each node's livetris can be maintained at run time, or lazily evaluated upon rendering

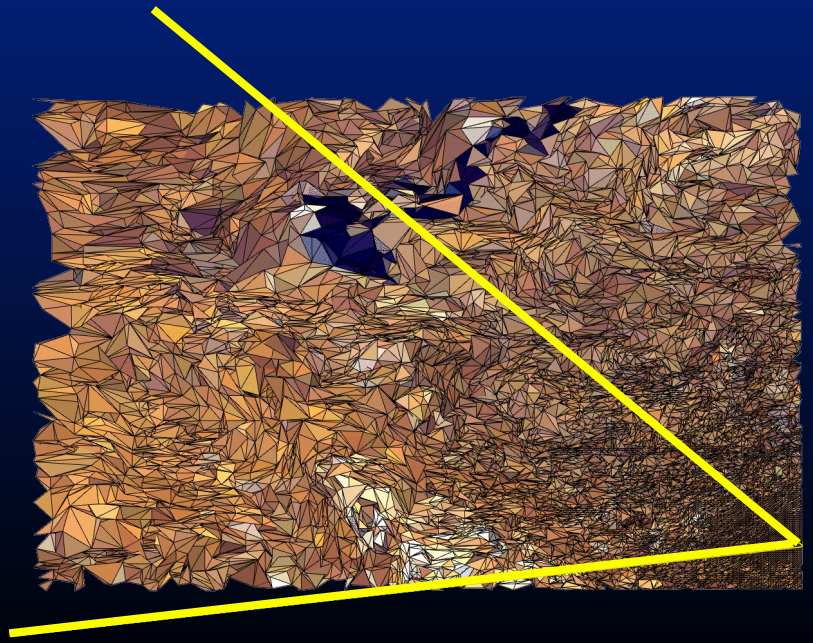
View-Dependent Simplification

- 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)

Screenspace Error Threshold

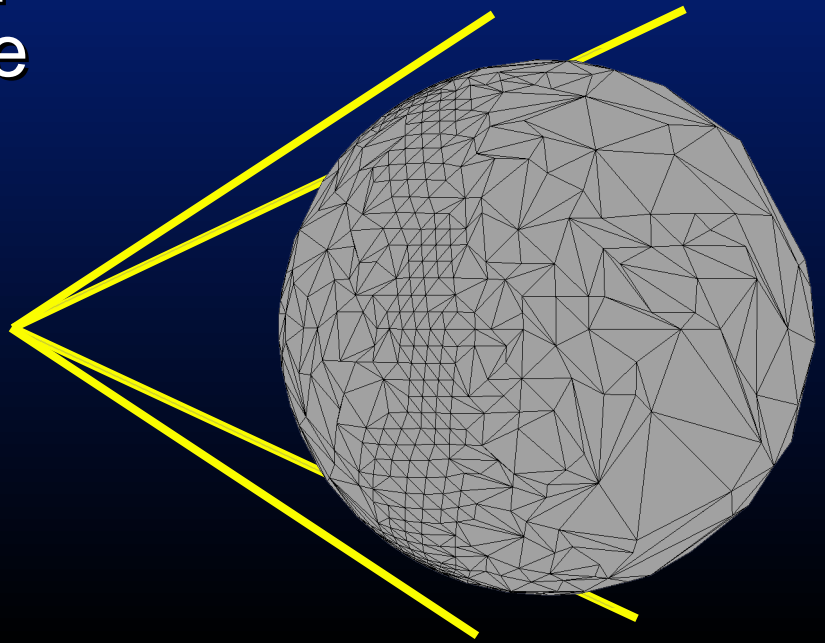
- Nodes chosen by projected area
  - User sets screenspace size threshold
  - Nodes which grow larger than threshold are unfolded





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

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

View-Dependent Criteria:
Other Possibilities

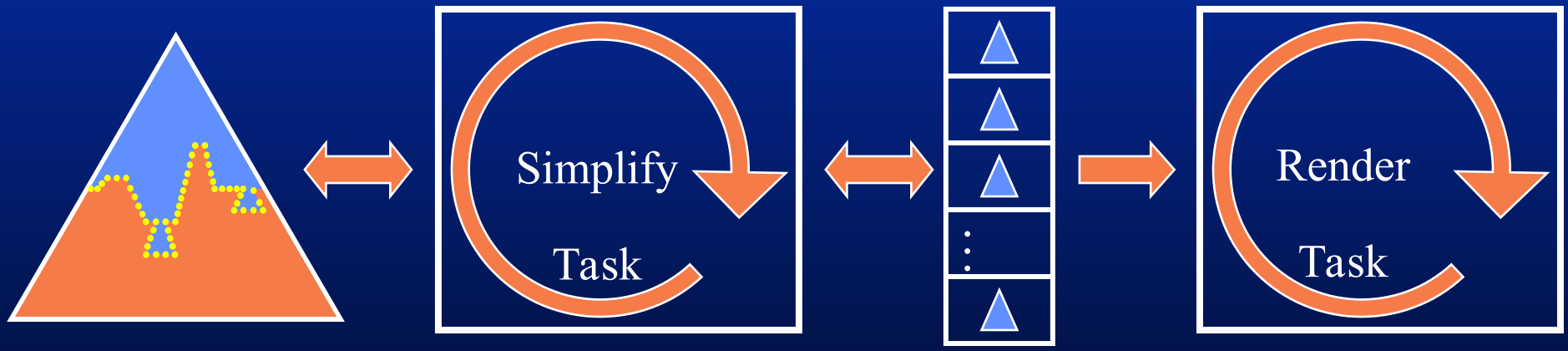
- *Sophisticated surface deviation metrics:*
  See Jon's talk!
- *Sophisticated perceptual criteria*:
  See Martin's talk!
- *Sophisticated temporal criteria*:
  See Ben's talk!

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

Asynchronous Simplification

- Algorithm partitions into two tasks:



Vertex Tree

Active Triangle List

- Run them in parallel

Asynchronous Simplification

- If *S* = time to simplify, *R* = time to render:
  - Single process = (*S* + *R*)
  - Pipelined = **max**(*S*, *R*)
  - Asynchronous = *R*
- The goal: efficient utilization of GPU/CPU
  - e.g., NV Vertex Array Range (VAR) rendering

Temporal Coherence

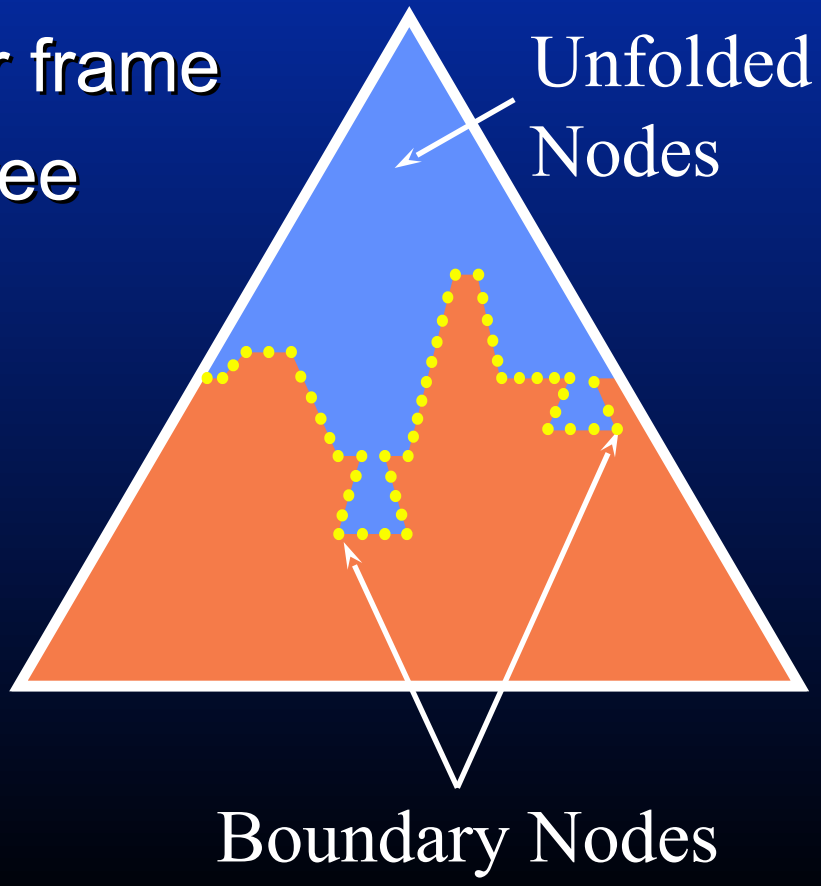
- Exploit the fact that frame-to-frame changes are small
- Three examples:
  - Active triangle list
  - Vertex tree
  - Budget-based simplification

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

Exploiting Temporal Coherence

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



Temporal Coherence: Triangle Budget Simplification

- Exploiting temporal coherence in budget-based simplification
  - Introduced by ROAM [Duchaineau 97]
  - Start with tree from last frame, recalculate error for relevant nodes
  - Sort into two priority queues
    - One for potential unfolds, sorted on max error
    - One for potential folds, sorted on min error

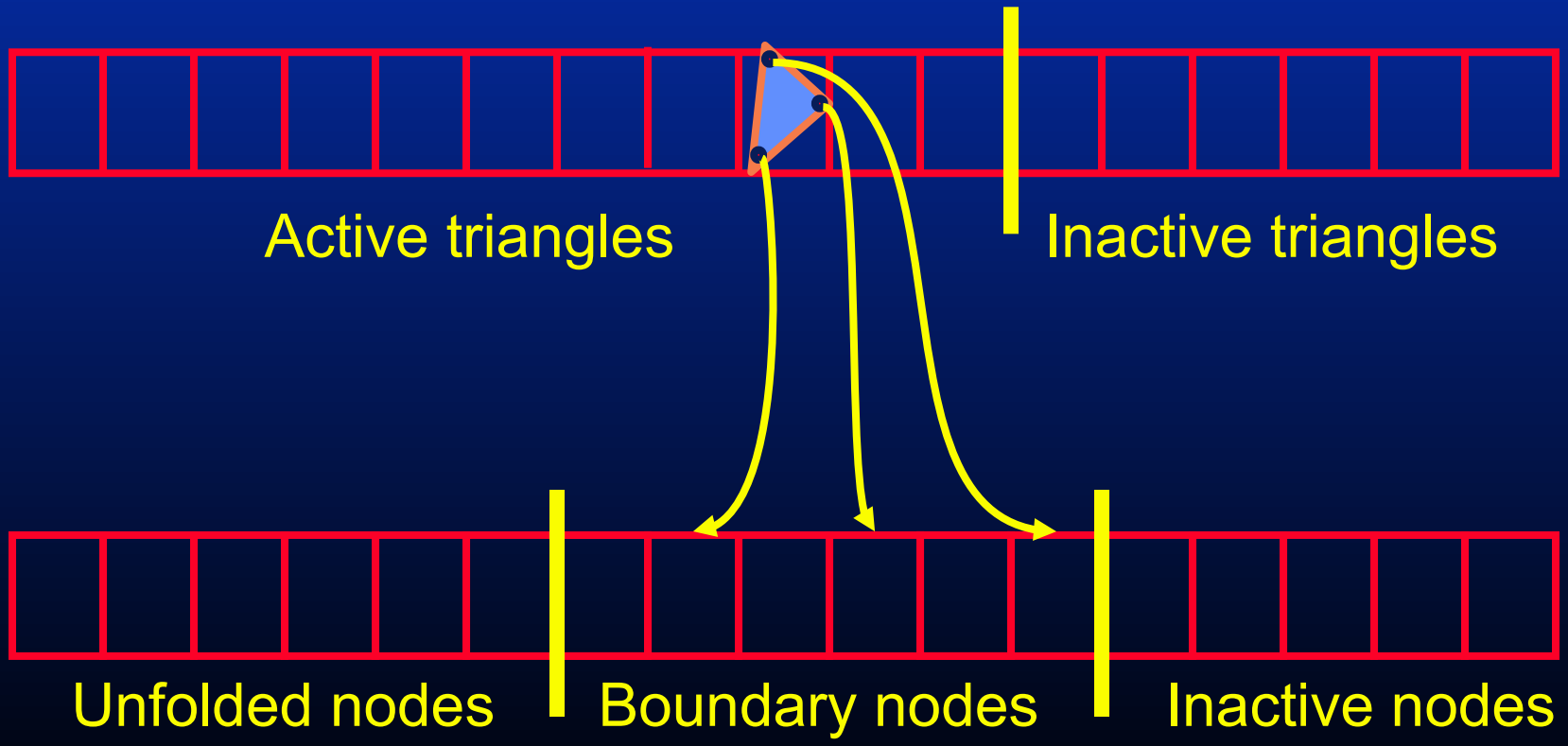
Temporal Coherence: Triangle Budget Simplification

- Then simplify:
  - While budget is met, unfold max node
    - This is the node whose folding has created the most error in the model
  - While budget is exceeded, fold min node
    - This is the node that introduces the least error when folded
  - Insert parents and children into queues

  Repeat until $error_{max} < error_{min}$

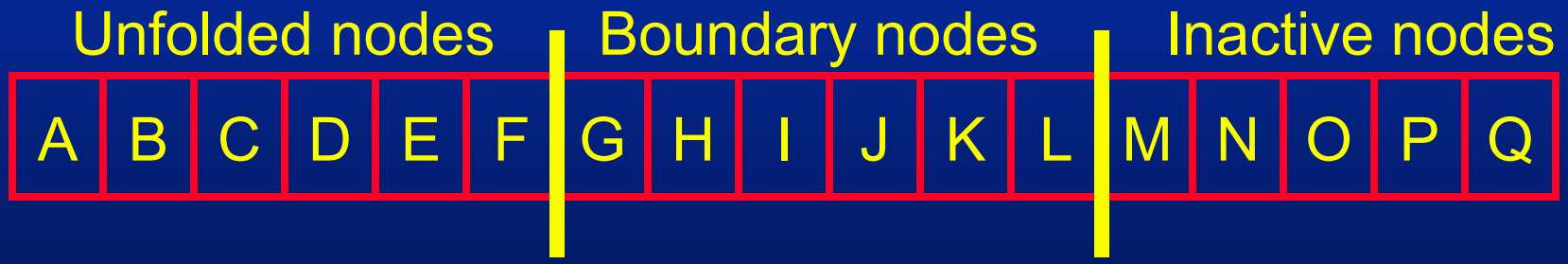
Optimizing For Rendering

- Idea: maintain geometry in coherent arrays



Optimizing For Rendering

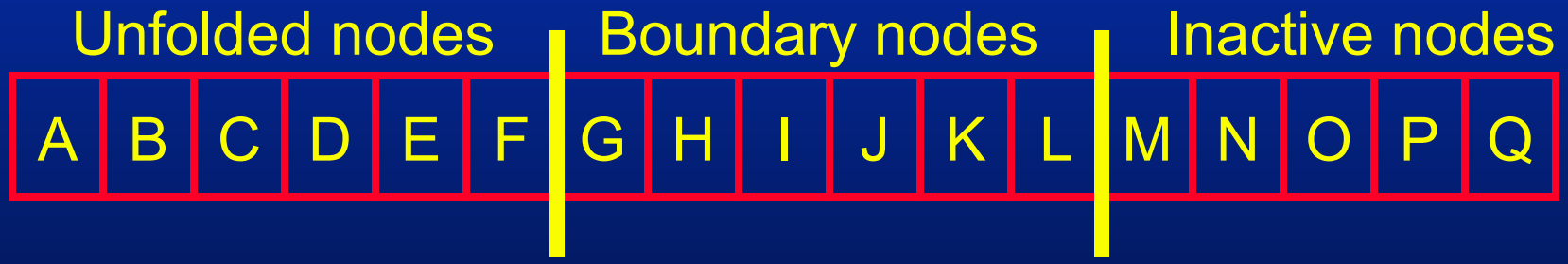
- Idea: use swaps to maintain coherence



Fold node D:

Optimizing For Rendering

- Idea: use swaps to maintain coherence

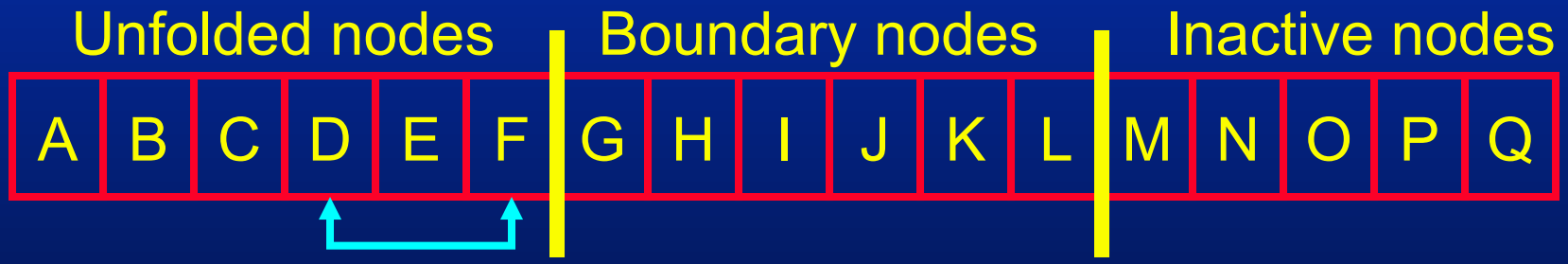


Fold node D:

Swap D with F

Optimizing For Rendering

- Idea: use swaps to maintain coherence

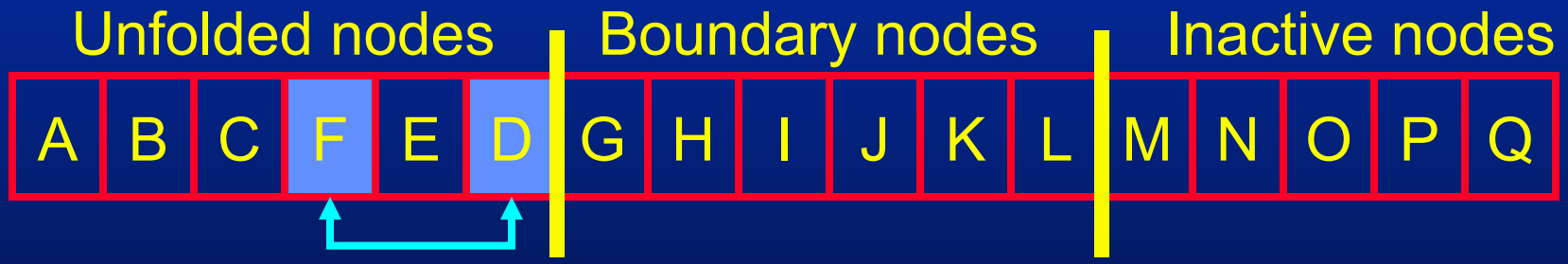


Fold node D:

Swap D with F

Optimizing For Rendering

- Idea: use swaps to maintain coherence

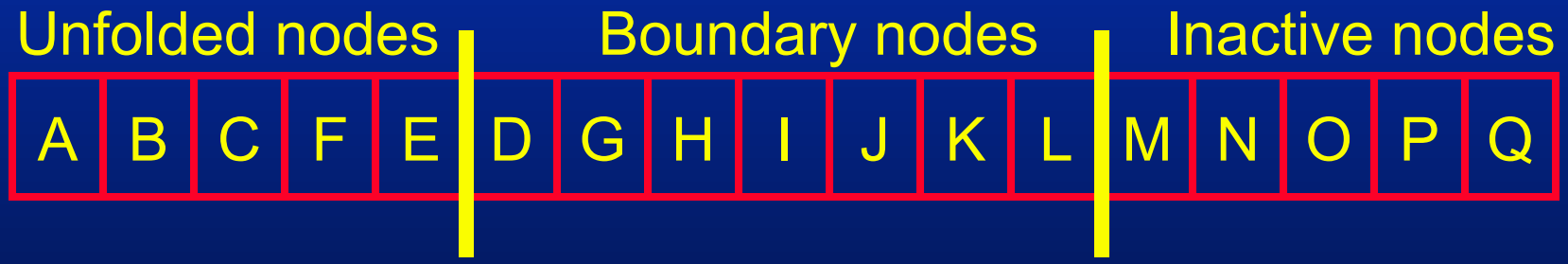


Fold node D:

Swap D with F

Optimizing For Rendering

- Idea: use swaps to maintain coherence

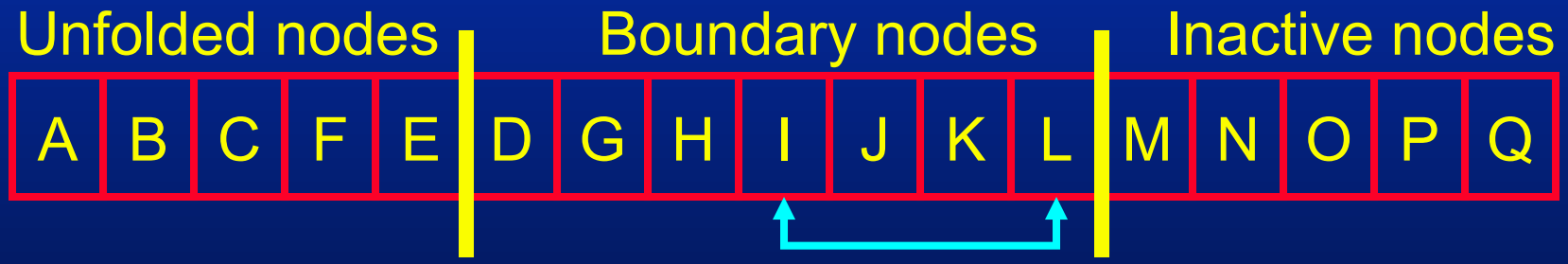


Fold node D:

Move Unfolded/Boundary Marker

Optimizing For Rendering

- Idea: use swaps to maintain coherence

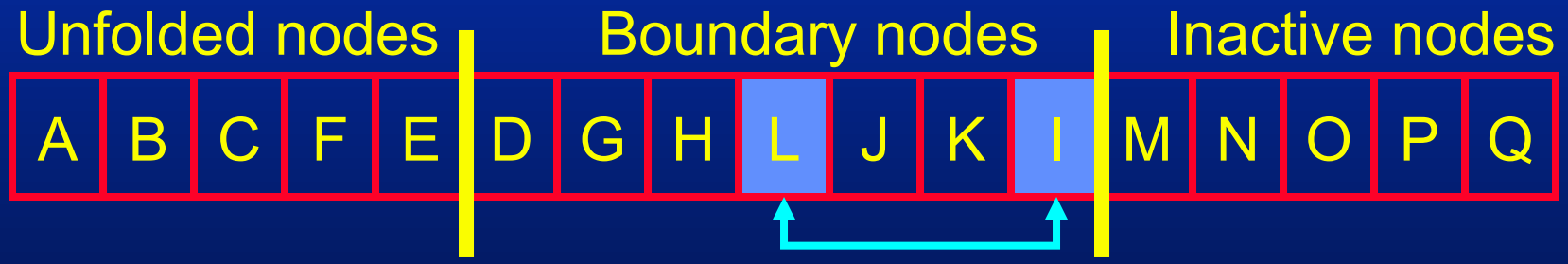


Fold node D:

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

Optimizing For Rendering

- Idea: use swaps to maintain coherence

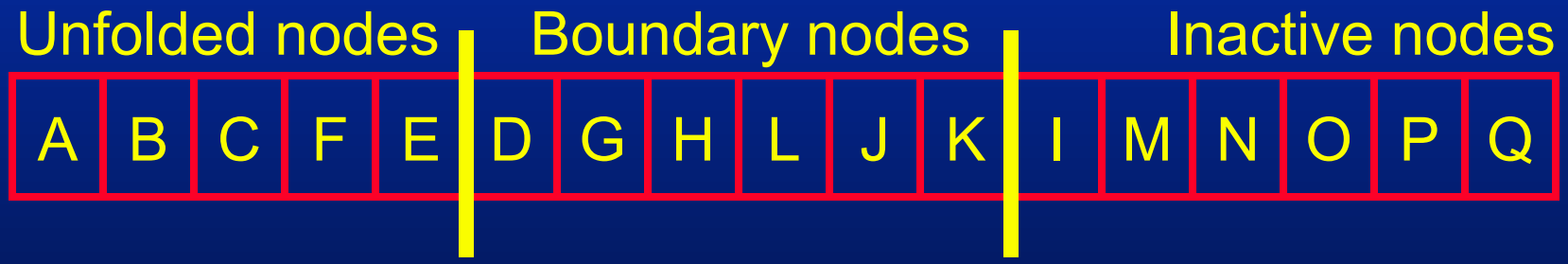


Fold node D:

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

Optimizing For Rendering

- Idea: use swaps to maintain coherence

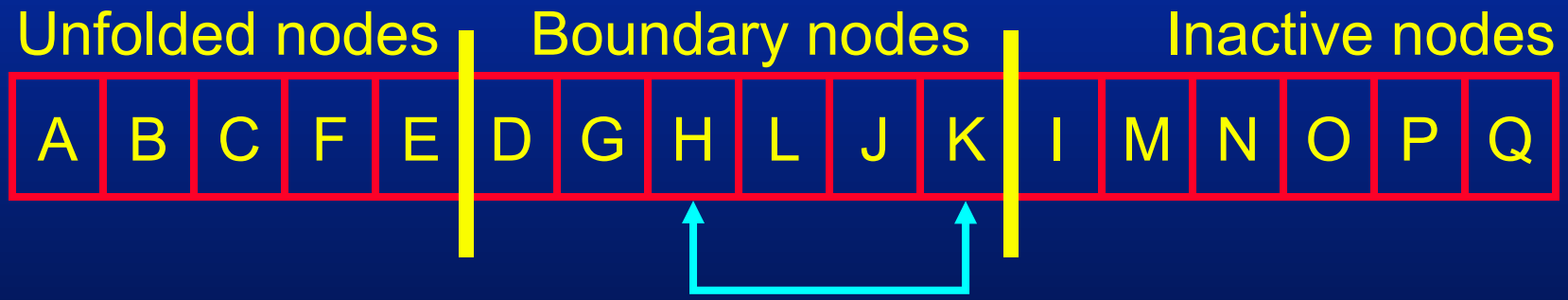


Fold node D:

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

Optimizing For Rendering

- Idea: use swaps to maintain coherence

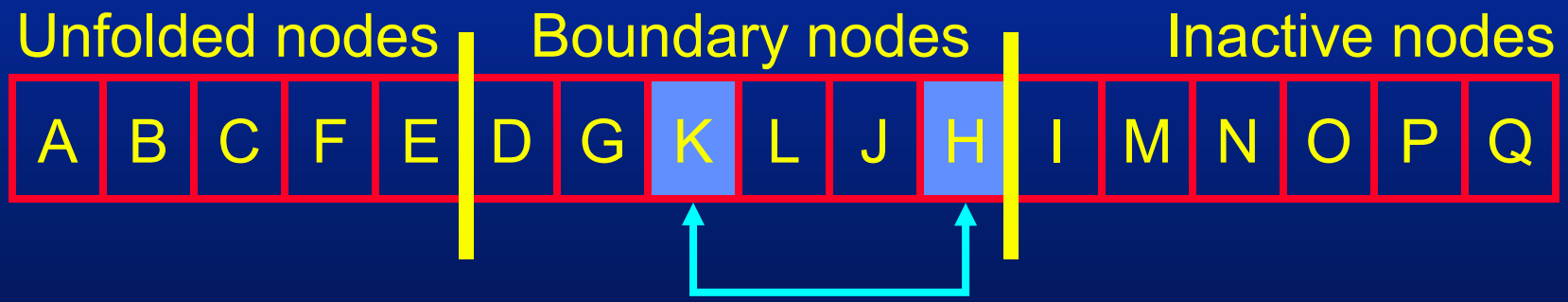


Fold node D:

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

Optimizing For Rendering

- Idea: use swaps to maintain coherence

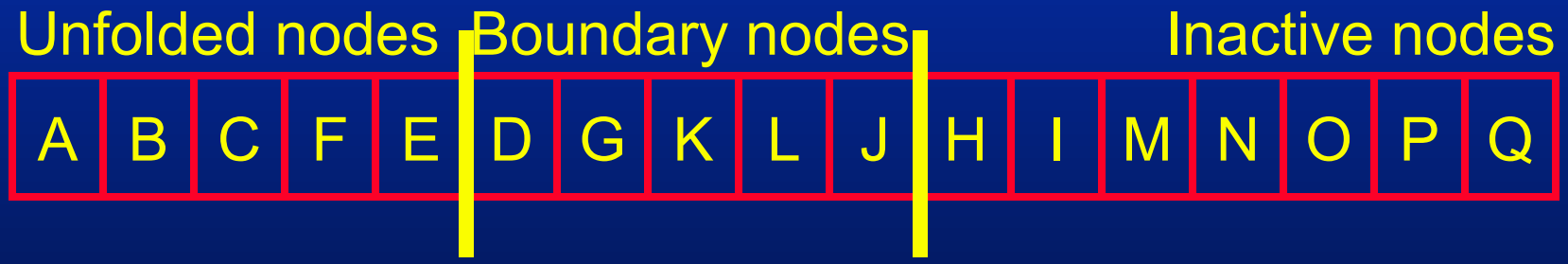


Fold node D:

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

Optimizing For Rendering

- Idea: use swaps to maintain coherence

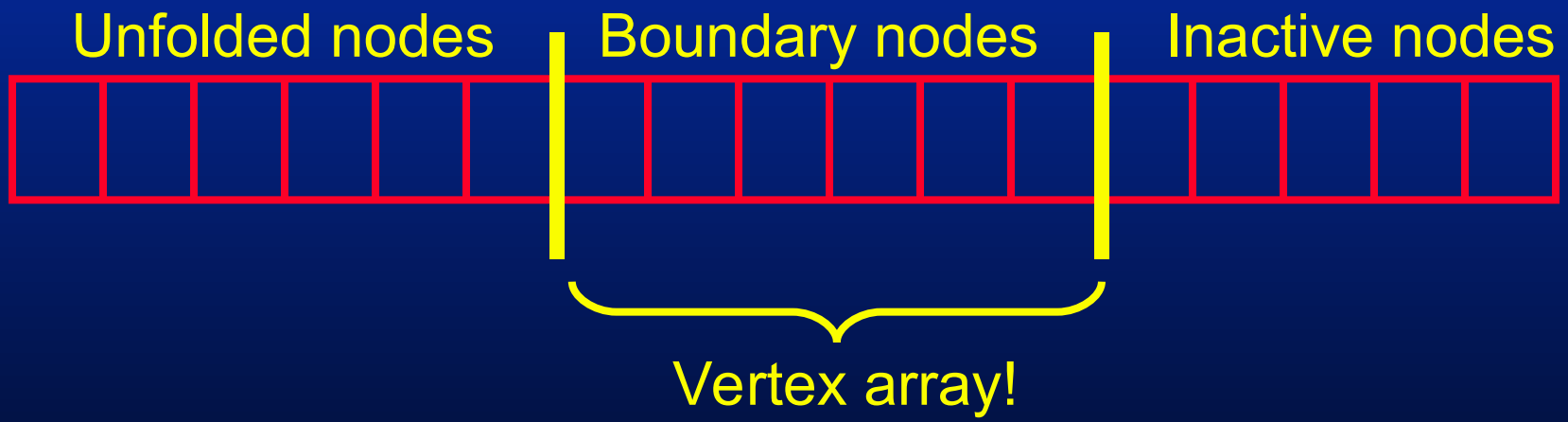


Fold node D:

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

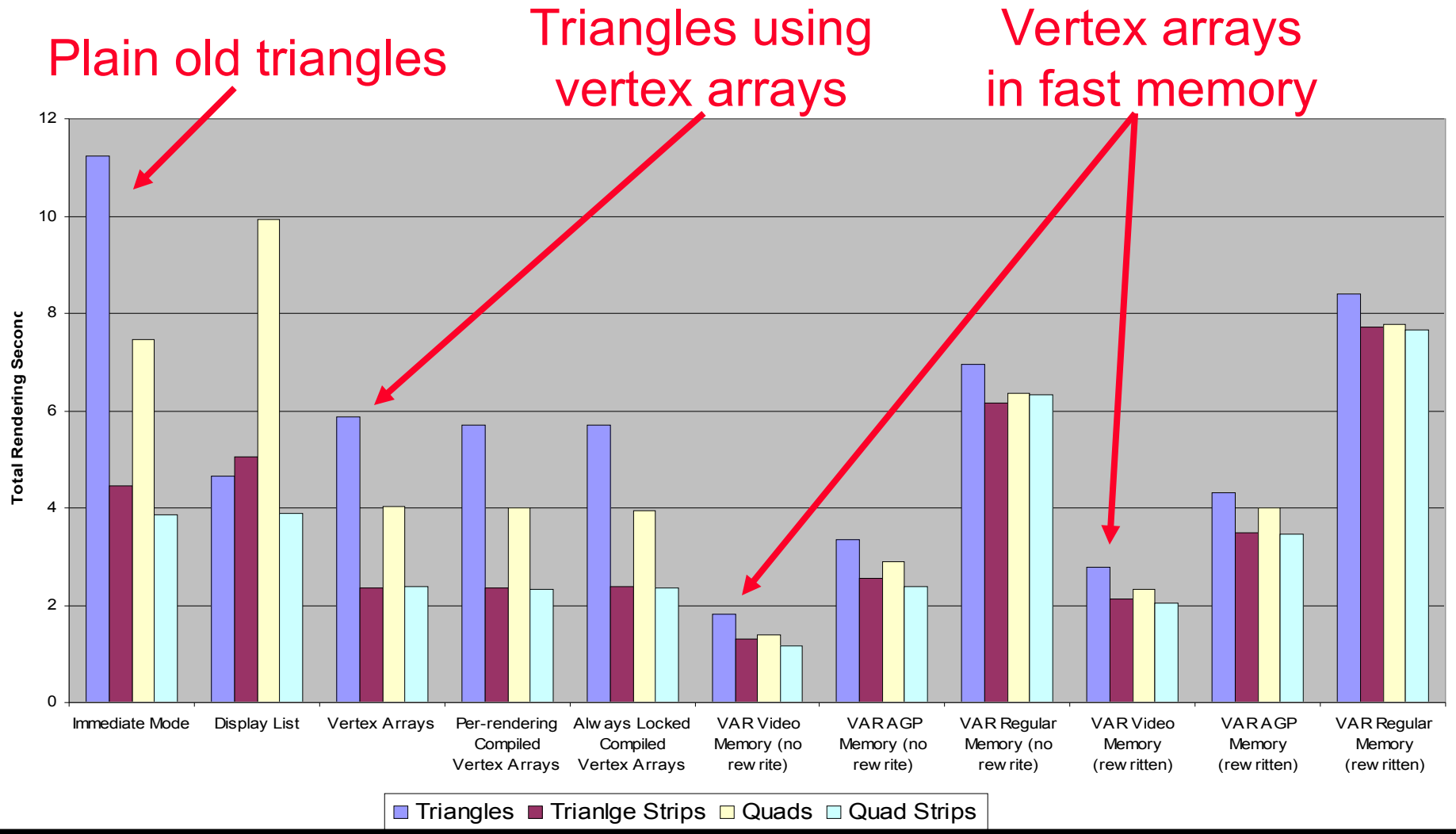
Optimizing For Rendering: Vertex Arrays

- Biggest win: vertex arrays



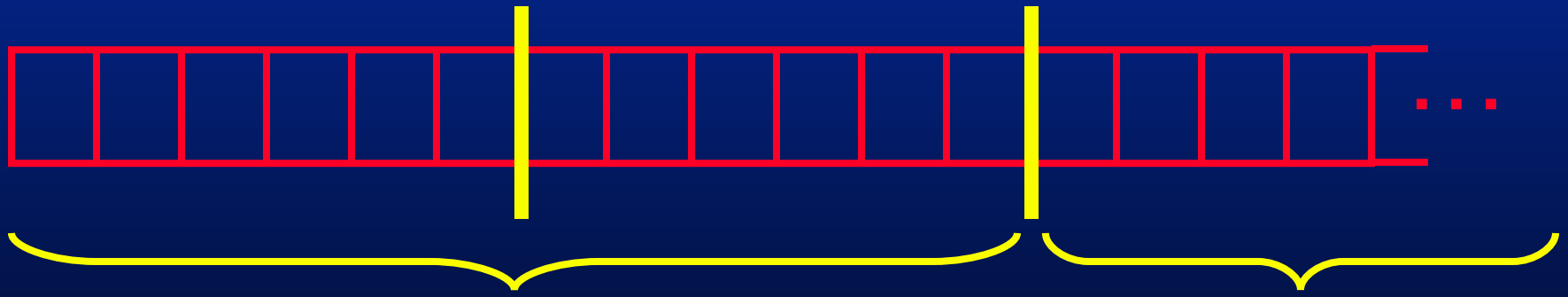
  - Actually, keep separate parallel arrays for rendering data (coords, colors, etc)

Optimizing For Rendering:
Vertex Arrays on GeForce2



Out-of-core Rendering

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



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

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

Summary:
VDS Pros

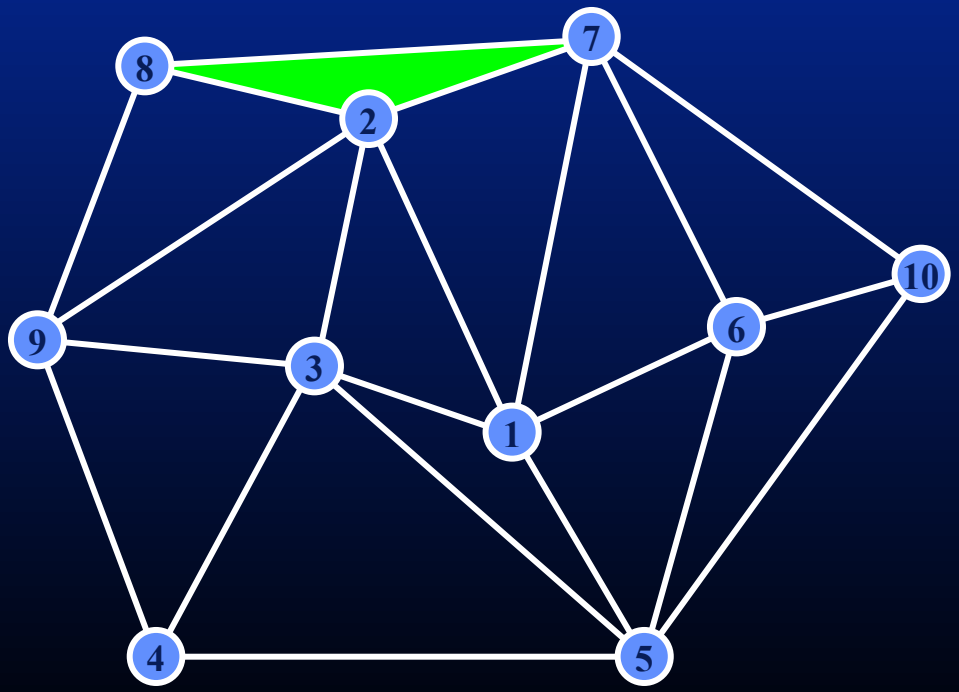
- Rendering can be implemented efficiently using vertex arrays
- Supports rendering of models much larger than main memory

Summary:
VDS Cons

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

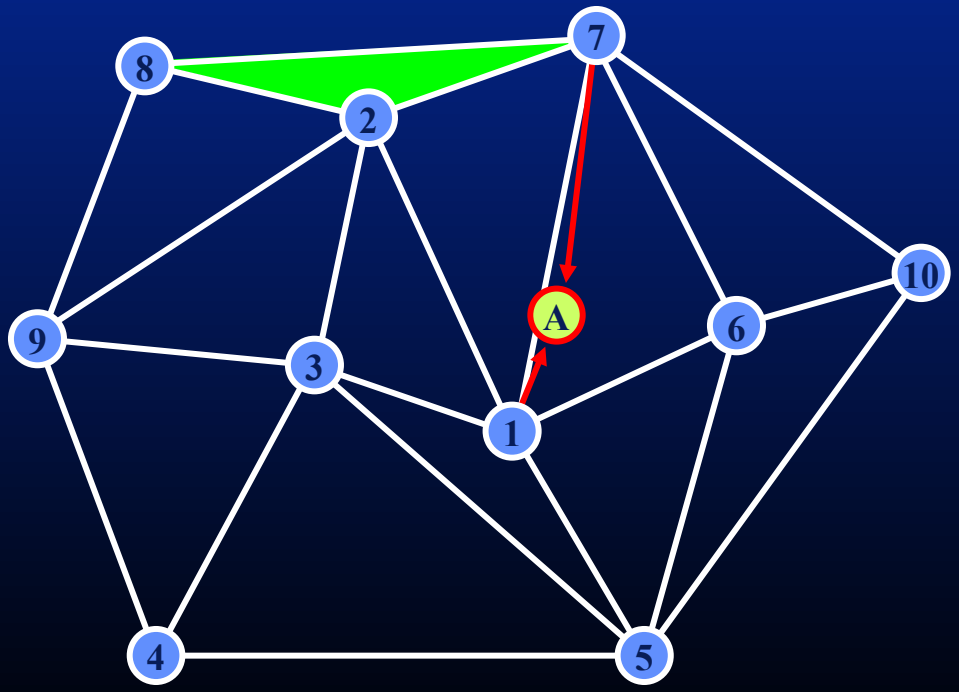
Summary:
VDS Cons

- Be aware of *mesh foldovers:*



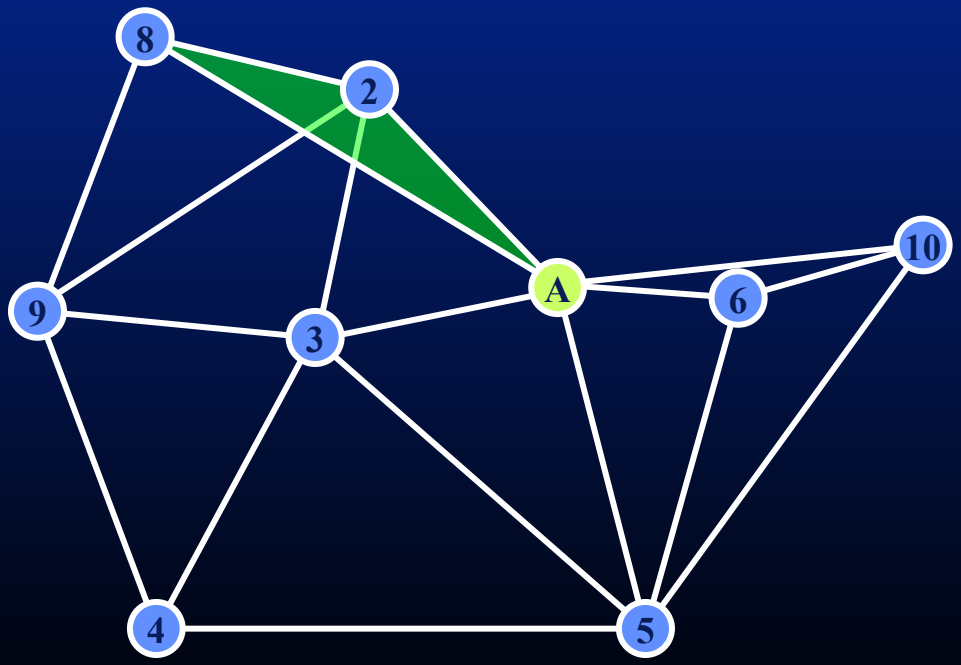
Summary:
VDS Cons

- Be aware of *mesh foldovers:*



Summary:
VDS Cons

- Be aware of *mesh foldovers:*



Summary:
VDS Cons

- Be aware of *mesh foldovers:*
  - These can be very distracting artifacts
  - Amitabh will talk about how to prevent them

View-Dependent Versus Discrete LOD

- View-dependent LOD is superior to traditional discrete 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

View-Dependent Versus Discrete LOD

- Discrete 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
    - Optimize vertex cache utilization and such

View-Dependent Versus Discrete LOD

- Applications that may want to use:
  - Discrete LOD
    - Video games (but much more on this later…)
    - Simulators
    - Many walkthrough-style demos
  - Dynamic and view-dependent LOD
    - CAD design review tools
    - Medical & scientific visualization toolkits
    - Terrain flyovers (much more later…)

Continuous LOD:
The Sweet Spot?

- Continuous LOD may be the right compromise on modern PC hardware
  - Benefits of fine granularity without the cost of view-dependent evaluation
  - Can be implemented efficiently with regard to
    - Memory
    - CPU
    - GPU

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*

VDSlib:
Ongoing Work

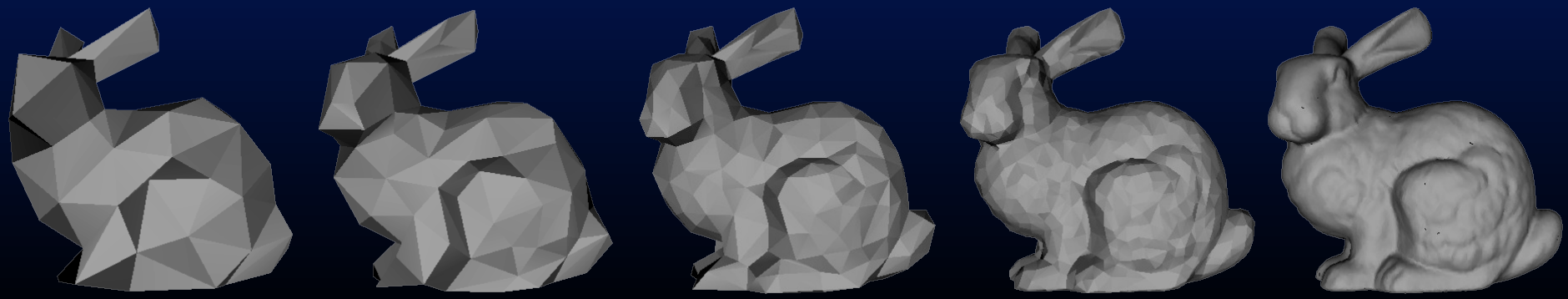
- Ongoing research projects using VDSlib:
  - Out-of-core LOD for interactive rendering of *truly* massive models
  - Perceptually-guided view-dependent LOD, including gaze-directed techniques
  - Non-photorealistic rendering using VDSlib as a framework

SAN ANTONIO

SIGGRAPH

2002

The End



Appendix:
Related Work

- Hoppe: Progressive Meshes (SIGGRAPH 96, SIGGRAPH 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

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

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
  - Follow-up papers, e.g. extending QEM to surface attributes
  - Public domain code for generating LODs with QEM

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

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.

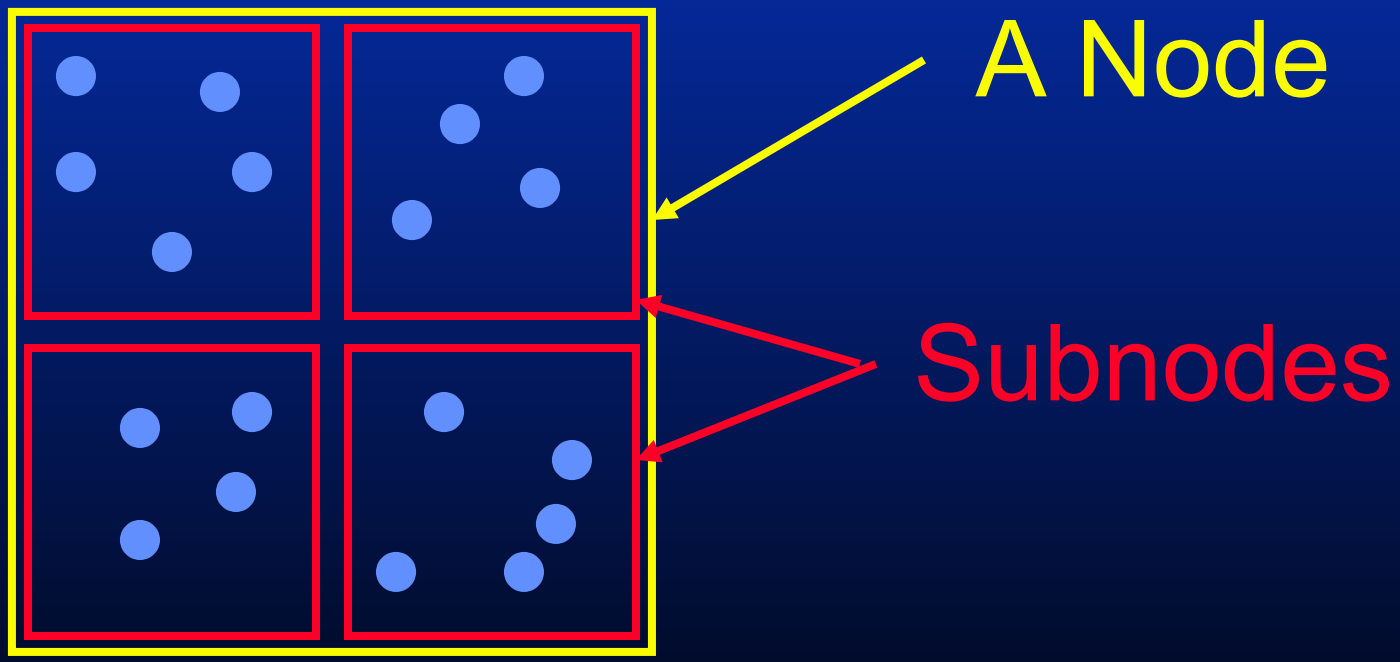
Appendix:
Attached Papers

- David Luebke, Jonathan Cohen,Nathaniel Williams, Mike Kelley, Brenden Schubert . *Perceptually Guided Simplification of Lit, Textured Meshes.* University of Virginia Tech Report CS-2002-07.
  - Describes ongoing work applying perceptual metrics (see Martin's talk) to view-dependent polygonal simplification.

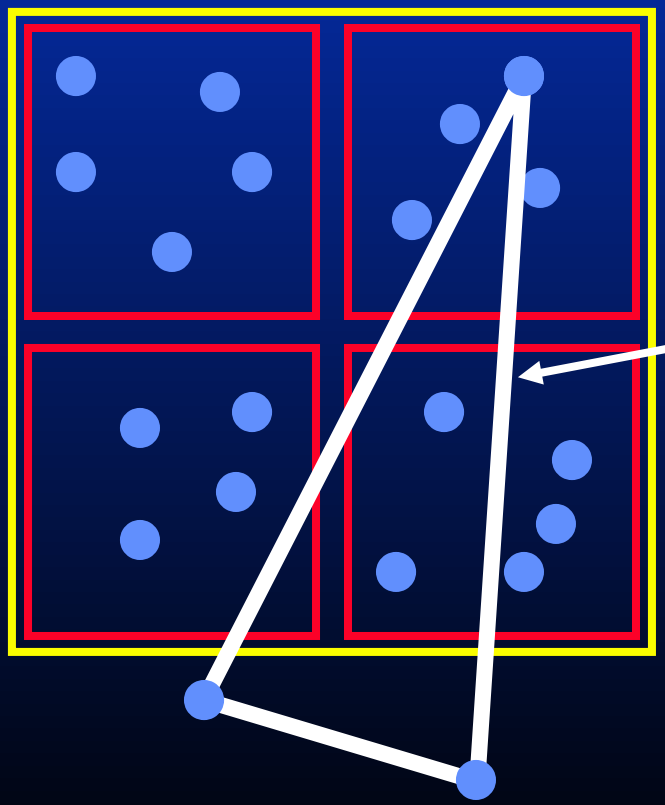
The Vertex Tree:
Livetris and Subtris

- 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

The Vertex Tree:
Livetris and Subtris

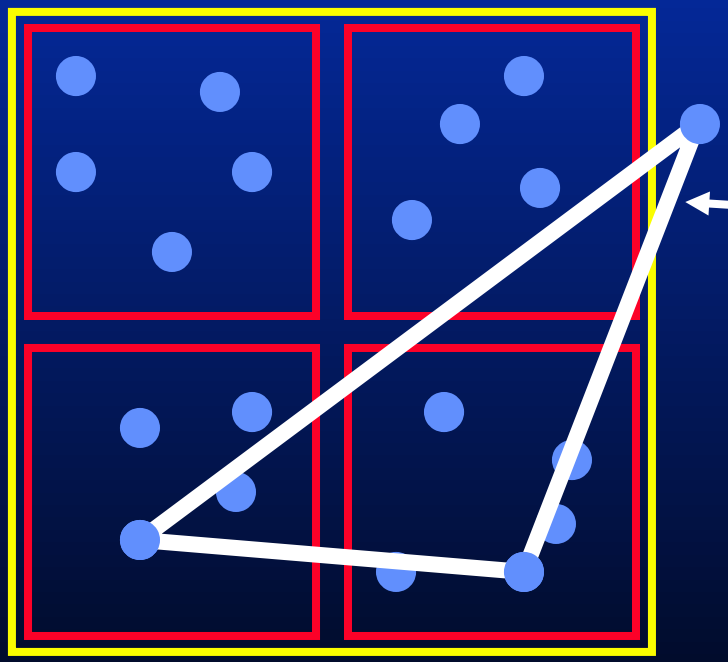


The Vertex Tree:
Livetris and Subtris



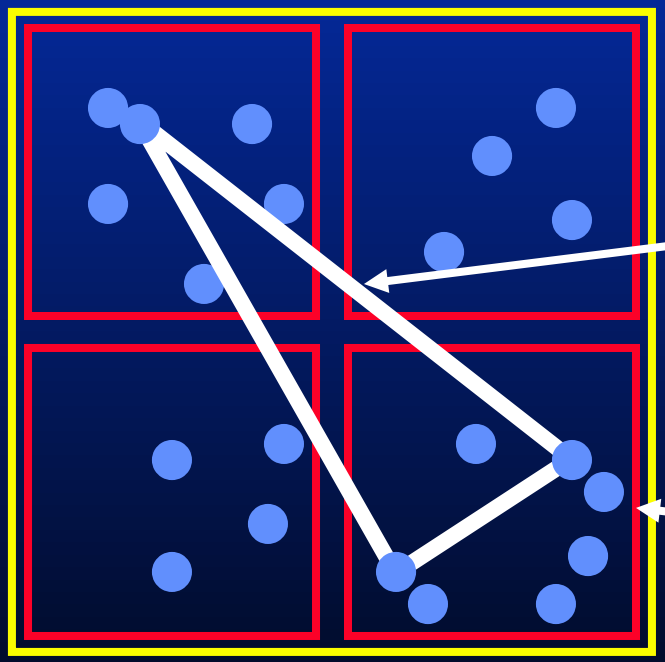
This is a **livetri** of the node

The Vertex Tree:
Livetris and Subtris



This is a **subtri** of the node

The Vertex Tree:
Livetris and Subtris



This is neither.

It's a subtri
of this subnode