Level of Detail: A Brief Overview

Fall 2012

Daniel G. Aliaga
(slides based on those of David Luebke @ NVIDIA)

Department of Computer Science
Purdue University
Introduction

- Geometric simplification/Level of detail (LOD) management are important tools for maintaining interactivity during 3D rendering
  - Focuses on the fidelity / performance tradeoff
  - Not the only tool! Complementary with:
    - Parallel rendering
    - Occlusion culling/visibility culling
    - Out-of-core rendering
    - Image-based rendering [etc]
Level of Detail: The Basic Idea

• The problem:
  – Geometric datasets can be too complex to render at interactive rates

• 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, decimation, multiresolution modeling, …
Level of Detail: Traditional LOD In A Nutshell

- 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 LOD In A Nutshell

- Distant objects use coarser LODs:
Level of Detail: The Big Questions

- How to represent and generate simpler versions of a complex model?
Level of Detail: The Big Questions

- How to evaluate the fidelity of the simplified models?

- Courtesy Stanford 3D Scanning Repository

- 69,451 polys
- 2,502 polys
- 251 polys
- 76 polys
Level of Detail: The Big Questions

- When to use which LOD of an object?

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

Courtesy Stanford 3D Scanning Repository
Some Background

• History of LOD techniques
  – Early history: Clark (1976), flight simulators
  – Handmade LODs → automatic LODs
  – LOD run-time management: reactive → predictive (Funkhouser)

• LOD frameworks
  – Discrete (1976)
  – Continuous (1996)
  – View-dependent (1997)
  – GPU-based (200X)
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, we call this *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 triangles 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
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 discrete approach:
  – Discrete LOD: create individual levels of detail in a preprocess
  – Continuous LOD: create data structure from which a desired level of detail can be extracted \textit{at run time}. 
Continuous LOD: Advantages

• Better granularity $\rightarrow$ 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 $\rightarrow$ 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
Summary: LOD Frameworks

- **Discrete LOD**
  - Generate a handful of LODs for each object

- **Continuous LOD (CLOD)**
  - Generate data structure for each object from which a spectrum of detail can be extracted

- **View-dependent LOD**
  - Generate data structure from which an LOD specialized to the current view parameters can be generated on the fly.
  - One object may span multiple levels of detail

- **(Hierarchical LOD)**
  - Aggregate objects into assemblies with their own LODs
Choosing LODs: LOD Run-Time Management

- Fundamental LOD issue: where in the scene to allocate detail?
  - For discrete LOD this equates to choosing which LOD will represent each object
  - Run every frame on every object; keep it fast
Choosing LODs

• Describe a simple method for the system to choose LODs
  – Assign each LOD a range of distances
  – Calculate distance from viewer to object
  – Use corresponding LOD
Choosing LODs

• **What’s wrong with this simple approach?**
  – Visual “pop” when switching LODs can be disconcerting
  – Doesn’t maintain constant frame rate; lots of objects still means slow frame times
  – Requires someone to assign switching distances by hand
  – Correct switching distance may vary with field of view, resolution, etc.

• **What can we do about each of these?**
Choosing LODs
Maintaining constant frame rate

• One solution: scale LOD switching distances by a “bias”
  – Implement a feedback mechanism:
    • If last frame took too long, decrease bias
    • If last frame took too little time, increase bias
  – Dangers:
    • Oscillation caused by overly aggressive feedback
    • Sudden change in rendering load can still cause overly long frame times
Choosing LODs: Maintaining constant frame rate

- A better (but harder) solution: predictive LOD selection
- For each LOD estimate:
  - *Cost* (rendering time)
  - *Benefit* (importance to the image)
Choosing LODs:
Maintaining constant frame rate

• A better (but harder) solution: predictive LOD selection

• For each LOD estimate:
  – *Cost* (rendering time)
    • # of polygons
    • How large on screen
    • Vertex processing load (e.g., lighting)  OR
    • Fragment processing load (e.g., texturing)
  – *Benefit* (importance to the image)
Choosing LODs: Maintaining constant frame rate

• A better (but harder) solution: predictive LOD selection

• For each LOD estimate:
  
  – *Cost* (rendering time)
  
  – *Benefit* (importance to the image)
    • Size: larger objects contribute more to image
    • Accuracy: no of verts/polys, shading model, etc.
    • Priority: account for inherent importance
    • Eccentricity: peripheral objects harder to see
    • Velocity: fast-moving objects harder to see
    • Hysteresis: avoid flicker; use previous frame state
Choosing LODs:

- Given a fixed time budget, select LODs to maximize benefit within a cost constraint
  - Variation of the knapsack problem
  - *What do you think the complexity is?*
    - A: NP-Complete (like the 0-1 knapsack problem)
  - In practice, use a greedy algorithm
    - Sort objects by benefit/cost ratio, pick in sorted order until budget is exceeded
    - Guaranteed to achieve at least 50% optimal sol’n
    - Time: \(O(n \log n)\)
    - Can use incremental algorithm to exploit coherence
Level of Detail: Generating LODs
Generating LODs

• Simplification Operator

• Measuring Error
Generating LODs

- Simplification Operator
- Measuring Error
Generating LODs

• Simplification Operator:
  – Cell collapse
  – Vertex removal
  – Edge collapse
Generating LODs

• Simplification operator:
  – Cell collapse
  – Vertex removal
  – Edge collapse
    • Full edge collapse
      – + Better fidelity
    • Half edge collapse
      – + Less memory
      – - Quality
    • Vertex-pair merge a.k.a. “virtual edge collapse”
      – - Merges separate objects
Edge Collapse Algorithm

Collapse
Edge Collapse Benefits

- Edge collapse operation is simple
- Supports non-manifold topology:
Even better: *vertex-pair merging* merges two vertices that:

- Share an edge, or
- Are within some threshold distance $t$

Q: *What does vertex-pair merging enable over edge collapse?*
Generating LODs

- Simplification Operator
- Measuring Error
Measuring Error

• Most LOD algorithms measure error geometrically
  – What is the distance between the original and simplified surface?
  – What is the volume between the surfaces?

• Really this is just an approximation to the actual visual error, which includes:
  – Color, normal, & texture distortion
  – Importance of silhouettes, background illumination, semantic importance, etc.
Measuring Visual Error

- Measuring error
  - Image-based ideas
  - Lindstrom & Turk, SIGGRAPH 2000
  - Perceptually-based ideas
  - Luebke & Hallen, EGRW 2001
  - Williams, Luebke, Cohen, Kelley & Schubert, I3D 2003
Measuring Geometric Error

• Measuring error
  – Hausdorff distance
    • One-sided: \( h(A, B) = \max_{a \in A} \min_{b \in B} \|a - b\| \)
    • Two-sided: \( H(A, B) = \max (h(A, B), h(B, A)) \)
  – Common approximations:
    • Measure vertex-vertex distance, vertex-plane distance
    • METRO: Sample H(A,B) by sprinkling points on triangles
    • *Quadric Error Metrics*: a variation of vertex-plane distance that works well in practice
Measuring Perceptual Error

• Idea:
  – Measure local simplification operations against a perceptual model to predict whether the user can see the effect of simplification

• Model
  – *Use a* contrast sensitivity function
Perception 101: 
Contrast Sensitivity Function

- Contrast grating tests produce a contrast sensitivity function
  - Threshold contrast vs. spatial frequency
  - CSF predicts the minimum detectable static stimuli
Your Personal CSF

Campbell-Robson Chart by Izumi Ohzawa
Level of Detail:
View-Dependent Simplification
View-Dependent LOD: Algorithms

• Many good published algorithms:
  – Progressive Meshes
  – Hierarchical Dynamic Simplification
  – Multitriangulation
  – Others...
Overview: The VDS Algorithm

• Overview of the VDS algorithm:
  – A preprocess builds the *vertex hierarchy*, 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: 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 hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
Vertex Tree Example

Triangles in active list

Vertex hierarchy
The Vertex Tree

- At runtime, folds and unfolds create a cut or *boundary* across the vertex tree:

  - This part of the model is represented at high detail
  - This part in low detail
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
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
Asynchronous Simplification

- Algorithm partitions into two tasks:
  - Run them in parallel

- Vertex Tree
- Simplify Task
- Active Triangle List
- Render Task

- 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
Temporal Coherence

• Exploit the fact that frame-to-frame changes are small

• Three examples:
  – Active triangle list
  – Vertex tree
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*
LOD Book

• http://lodbook.com/source/
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](http://vdslib.virginia.edu)
GLOD

• An easy-to-use library for level of detail in OpenGL
  – LOD generation
  – LOD run-time management
  – View-dependent LOD (using VDSlib)

http://www.cs.jhu.edu/~graphics/GLOD