Progressive Mesh

Reddy Sambavaram
Insomniac Games
LOD Schemes

- Artist made LODs (time consuming, old but effective way)
- ViewDependentMesh (usually used for very large complicated meshes. CAD apps. Probably the future???)
- ViewIndependentMesh (started off as a retro fit to get view dependent ideas on to view independent meshes for the current gen games)

Indexed based lod (we don’t touch vertex data but re-hash index data for lods)
Factors affecting the choice of LOD Scheme

- Per Class memory (mesh representation data shared by all instances of the mesh)
- Instance memory cost
- Per-frame processing cost
- Complexity of the scheme (artist/set-up time needed)
- Vertex cache coherency
End of Resistance 1

- Ties and ufrag had no lod
- Few mobys had discrete artist made lod (remember mn0, mn1 maya nodes...)
- We did have shader lod
  - (where shader defaulted to cheaper version of itself [disabling detail etc..] at distance).
Index Lod Basics

Tri indices: 0,1,5, 0,5,3, 3,5,4, 4,5,1, 4,1,2, 4,2,3

LOD Tri indices: 0,1,5, 0,5,3, 5,1,2, 5,2,3

• Collapse edge 4-5 by replacing index 4 with 5 and remove the triangles associated with that edge.

• If all the tris in above mesh were planar and it was flat shaded, we wouldn’t perceive the difference among the two meshes.

• Essentially at a given distance we want to collapse a certain set of edges (and remove the tris with it) such that the reduced mesh is indistinguishable from the hi-res one.
Advantages of the Index Lod

• No spu time needed
• No per instance memory unlike other lod schemes
• Just make the index offset point to different location per instance (vertex data un-touched)
• Minimal user time (mostly automatic) needed to set up the lod
Dis-advantages of Index Iod

• Instead of collapsing one vertex on to another one, if instead we collapsed both vertices on to a new optimal one then we can get the distance at which we collapse much closer.

• Edge collapse is a discrete jump rather than a smooth transition

But the benefits of minimal processing and lower memory requirements out-weighed the dis-advantages for us.
Iterative Edge Contraction

• Every edge is assigned a cost (an estimate of visual quality degradation if we collapse it)
• Maintain edges in a heap keyed on cost
• At each iteration we extract the lowest cost edge and collapse it
• This greedy way of choosing the edges is a good compromise between a very fast low-quality methods and very slow high quality methods.
• It provides a good estimate of local shape and error
Quadric Error (PART 1)

- Each triangle of the original mesh satisfies the plane eq

\[ n^T v + d = 0 \]

- For a given edge collapse a->b, compute all the geometric error by computing sum of the errors by substituting “a” by “b” for all tris which involve “a”.

Quadric Error

• Squared distance of vertex to the plane

\[ Quadric : Q(v) = D^2 = (n^T v + d)^2 = v^T(n^T n)v + 2dn^T v + d^2 = v^T Av + 2b^T v + c \]

• For a given edge collapse a->b, quadric error is defined as sum of all the quadrics defined by the faces involving “a” i.e.,

\[ QuadricError : E(a- \rightarrow b) = \sum_{\text{TrisWith"a"}} \text{TriArea} \otimes Q(b) \]

• For 3 dimensions we can easily visualize. Now extend it to n dimensions. Each vertex has a geometric position and m scalar attributes (base map uvs, light map uvs, color, normals, skinning info...).

\[ v = [p, s] \in \mathbb{R}^{3+m} or \mathbb{R}^n \]
Quadric Error

Orthonormal vectors $e_1$ and $e_2$ define a local frame, with origin $p$, for the 2-plane defined by the triangle $(p, q, r)$.

\[
e_1 = \frac{q - p}{\|q - p\|}
\]

\[
e_2 = \frac{r - p - (e_1 \cdot (r - p))e_1}{\|r - p - (e_1 \cdot (r - p))e_1\|}
\]
Quadric Error

\[ \| \mathbf{u} \|^2 = (\mathbf{u} \cdot \mathbf{e}_1)^2 + \cdots + (\mathbf{u} \cdot \mathbf{e}_n)^2, \]

Perpendicular distance = \[
(u \cdot e_3)^2 + \cdots + (u \cdot e_n)^2 = \| u \|^2 - (u \cdot e_1)^2 - (u \cdot e_2)^2.
\]

\[
D^2 = \| \mathbf{u} \|^2 - (\mathbf{u}^T \mathbf{e}_1)^2 - (\mathbf{u}^T \mathbf{e}_2)^2
= \mathbf{u}^T \mathbf{u} - (\mathbf{u}^T \mathbf{e}_1)(\mathbf{e}_1^T \mathbf{u}) - (\mathbf{u}^T \mathbf{e}_2)(\mathbf{e}_2^T \mathbf{u}).
\]

\[
D^2 = v^T v - 2p^T v + p \cdot p \\
- v^T (e_1 e_1^T)v + 2(p \cdot e_1)e_1^T v - (p \cdot e_1)^2 \\
- v^T (e_2 e_2^T)v + 2(p \cdot e_2)e_2^T v - (p \cdot e_2)^2,
\]

\[ D^2 = v^T A v + 2b^T v + c \] where:

\[
A = I - e_1 e_1^T - e_2 e_2^T \\
b = (p \cdot e_1)e_1 + (p \cdot e_2)e_2 - p \\
c = p \cdot p - (p \cdot e_1)^2 - (p \cdot e_2)^2.
\]
Quadric Error

- Matrix A of size \((3+m)x(3+m)\) is dense
- Storage for the co-efficients in error = \((4+m)(5+m)/2\) (quadratic in \(m\))
- To trade off geometric accuracy and attribute accuracy, user specifies for each attribute a relative importance weight.
  - we pre-multiply attributes by weight
  - i.e., scaling in some axes in \(3+m\) space
- For scale invariance, all attributes are resized to be unit length in each attribute field.
  - can be cost prohibitive memory and computational complexity wise as the mesh size increases)
Modified Quadric Error Metric

- Rather than projecting a given point onto the tri face in an abstract higher-dimensional space of size $3+m$, do projection in geometric space of size 3.
- Compute attribute error based on geometric correspondence (Huges Hoppe).

\[ |p - p'| = \text{geometric error} \]
\[ |s - s'| = \text{attribute error} \]

Correspondence between point $p$ with attribute $s$ and its projection onto the plane spanning face $(v_1, v_2, v_3)$.

- Error = Geometric error (projected distance) + attribute error.
- To compute attribute error we need to compute expected attribute value $s'$ at the projected point $p'$ on the tri.
Modified quadric error metric

\[ s_j'(p') = g_j^T p + d_j \]

- Gradient \( g \) and scalar \( d \) should linearly interpolate over the tri
- Hence we compute them by 4x4 linear system

\[
\begin{pmatrix}
  p_1^T & 1 \\
  p_2^T & 1 \\
  p_3^T & 1 \\
  n^T & 0
\end{pmatrix}
\begin{pmatrix}
  g_j \\
  d_j
\end{pmatrix}
= \begin{pmatrix}
  s_{1,j} \\
  s_{2,j} \\
  s_{3,j} \\
  0
\end{pmatrix}
\]

- Hence the geometric error is

\[
(\hat{s}_j(p) - s_j)^2 = (g_j^T p + d_j - s_j)^2
\]

- Storage cost comparison for the new error metric is 11+4m [compared to (4+m)x(5+m)/2]
- Our library uses above error metric (faster to compute and easier to store)
Handling Discontinuities

• So far, things would work fine if there are no geometric or attribute discontinuities. (which are bound to happen every where in our data)
• Basic algorithm above ignores boundaries and is clearly un-acceptable
Handling Discontinuities

- To preserve geometric dis-continuities, for each boundary edge we also take the distance from the plane perpendicular to the triangle along the edge.
- Therefore, cost of moving along the edge is lower than cost of moving away.
- Thus it allows interior vertex on the boundary to collapse into another vertex on the same boundary, but heavily weighs against moving it away from the boundary.
- Hence, the algorithm tries to preserve the feature edges (which are marked as boundaries using dis-continuities in attribute and geometric data by a pre-processing step before anything starts.)

Sample boundary constraint plane. Every edge along the boundary defines a single constraint plane.
Briefly “what is Indexed based Rendering”:
• All vertices are in one memory pool/list/vector
• Every tri is represented by indices of the vector of vertices which make it.
• All the indices of all tris in a mesh are in one memory pool
• GPU runs through the indices and for every 3 pull those corresponding vertices to render the triangle

Sliding Window algorithm:
• Each edge collapse (a patch) removes some tris and adds new tris which are less than what it removed
• Index ordering is dictated by the patches
• As distance increases we slide the start and end pointers (like lod0 & lod1 in the figure). Thus we progressively lod with increasing distance.
Sliding window algorithm
Sliding window algorithm

• At each frame based on distance from camera we estimated how many tris we need to draw
  – based on it which we moved the window on the index buffer
  – (we used a simple linear estimate function, adjustable by user)
• Granularity of a patch to a collapsed edge implied too many patches (un-acceptable index buffer bloat)
• So, we made our patches not one edge at a time.
  – It was compute based on error metric (for each delta of error we made a patch).
• Solved index buffer bloat and also aids in vertex coherency
  – un-explored for lack of time
  – But our current scheme essentially does this at few discrete steps).
Analysis of Sliding Window Scheme

Advantages:
• Fast progressive algorithm which implies we start lod’ing from a short distance
• Gradual small pops rather than discrete big pops (which means farther away we push distance to lod)

Dis-advantages:
• Index ordering is dictated by algorithm not exactly ideal ordering for the GPU
• Index memory bloat 3 to 4x

<table>
<thead>
<tr>
<th>Tie</th>
<th>Main ram</th>
<th>class’s vert data</th>
<th>Index data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>282/(200k) 40%</td>
<td>146.7k</td>
<td>117.74k/( 34k)</td>
</tr>
<tr>
<td>680</td>
<td>454/(334k) 36%</td>
<td>239.3k</td>
<td>174.8k/(55.04k)</td>
</tr>
</tbody>
</table>

(we could mitigate both the dis-advantages by increasing error delta for a patch and with in a patch index-order its tris for GPU. Essentially our current scheme)
Early Performance numbers

Sample early development [digging into my outlook] run-time stats at the same distance (compared to non-loding mesh)

<table>
<thead>
<tr>
<th>Tie</th>
<th>80% tri reduction</th>
<th>66%</th>
<th>47%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1267/0.149 ms(85%)</td>
<td>0.1068/.1425 (74%)</td>
<td>0.0727/0.135 (53%)</td>
</tr>
<tr>
<td>680</td>
<td>0.182/0.205 (88%)</td>
<td>0.169/0.196(86%)</td>
<td>0.101/0.187(54%)</td>
</tr>
</tbody>
</table>

When initially released into the wild, Ratchet never needed more than 60% of its tri count

(We even decided to silently cut the hi-lod patches and start the hi-res at 60% for game and keep the rest around only for cinematics)

“Btw, It's really cool to see how low the "far" scores can get when you combine fade-out lod and index lod. For that ratchet moby 376, I could easily get the far score down around 2.5 where it had been at 19.5 with no visible difference. “ – From an old mail from Al Hastings
• This is from self and data in i8 from last dec/jan (not the final version but the last version I cached as a back up)
Sample results
Sample results
Sample results
Sample Results
Sample Results
Sample Results
Sample Results
Sample Results
Sample Results
Current LOD Scheme used in R&C:TOD

• At the time tie’s had the concept of segments which also forced us to go in the direction of few discrete steps
  – as segment boundaries with in a fragment had to be preserved
  – our tool essentially did few discrete pops and optimized each patch for GPU index ordering.
• Current scheme essentially does what ties with segmenting did,
  – few finite steps across the whole asset (all fragments combined)
  – Fixes vertex cache and index bloat both of the sliding window deficiencies.
• Inherently few large patch error steps.
• Each patch index order is optimized for GPU
Current LOD Scheme used in R&C:TOD

• Num patches defaulted to 4 steps of 60%, 40% and 25% reductions
• User could tweak above settings and the distance at which it happens
• We hijacked [A Friend’s] tools to generate above steps as it considers asset as a whole
  – our tool was more developed for sliding window
  – our tool treated fragment boundaries as normal boundaries and operated independently inside it
• Index LOD is also aided by fade-out lod where beyond certain distance certain fragments have the ability to slowly fade out
Reference Papers

• “New Quadric Metric for Simplifying Meshes with Appearance Attribute” Hugues Hoppe
• Hugues Hoppe’s phd thesis
• “Efficient Minimization of New Quadric Metric for simplifying Meshes with Appearance Attributes” Hugues Hoppe & Steve Marschner
• “Simplifying Surfaces with Color and Texture using Quadric Error Metrics” Michael Garland & Paul S. Heckbert
• “Topology-driven Progressive Mesh Construction for Hardware-Accelerated Rendering” Pavlo turchyn & Sergey Korotov (Sliding Window finders)
• “Quadric-Based Simplification in Any Dimension” Michael Garland & Yuan Zhou
• “A Multiphase Approach to Efficient Surface Simplification” Michael Garland & Eric Shaffer
• “User-Guided Simplification” Youngihn Kho & Michael Garland