15 Efficient meshes

Follows chapter 16 in RTR 4e
Basics of efficiency for meshes

Use triangle or quad meshes

- general polygon meshes lead to too much complexity
- quad meshes are great for some applications but more constrained

Use shared-vertex triangle meshes for GPU applications

- major memory/bandwidth savings over separate triangles
- if you get separate triangles, merge them in a pre-process

Store most data at vertices

- there are ~half as many vertices as faces
- vertex data may be interpolated across faces
- in typical GPU mesh representation, vertices must be duplicated to create discontinuities
More sophistication in mesh storage

**Optimizing vertex order**
- strips and fans as classic examples (when per-frame bandwidth was the concern)
- modern systems don’t use these but optimize for hit rate in vertex cache

**Reducing the number of triangles**
- ultimately this is needed to save more time and space
- many *levels of detail* are useful
  - simpler meshes for faraway objects
  - simpler meshes for lower-resolution screens
  - simpler meshes for lower-performance hardware or networks
Triangle strips

• **Take advantage of the mesh property**
  - each triangle is usually adjacent to the previous
  - let every vertex create a triangle by reusing the second and third vertices of the previous triangle
  - every sequence of three vertices produces a triangle (but not in the same order)
  - e.g., 0, 1, 2, 3, 4, 5, 6, 7, … leads to
    - \((0 \ 1 \ 2), (2 \ 1 \ 3), (2 \ 3 \ 4), (4 \ 3 \ 5), (4 \ 5 \ 6), (6 \ 5 \ 7), \ldots\)
  - for long strips, this requires about one index per triangle
Triangle strips

| verts[0] | $x_0, y_0, z_0$ |
| verts[1] | $x_1, y_1, z_1$ |
|          | $x_2, y_2, z_2$ |
|          | $x_3, y_3, z_3$ |
|          | ...            |

| tStrip[0] | 4, 0, 1, 2, 5, 8 |
| tStrip[1] | 6, 9, 0, 3, 2, 10, 7 |
|           | ...            |
Triangle strips

verts[0]
  x₀, y₀, z₀
verts[1]
  x₁, y₁, z₁
  x₂, y₂, z₂
  x₃, y₃, z₃
  ...

tStrip[0]  4, 0, 1, 2, 5, 8
  tStrip[1]  6, 9, 0, 3, 2, 10, 7
  :
Triangle strips

- **array of vertex positions**
  - `float[nV][3]`: 12 bytes per vertex
    - (3 coordinates x 4 bytes) per vertex

- **array of index lists**
  - `int[nS][variable]`: 2 + \( n \) indices per strip
  - on average, \((1 + \epsilon)\) indices per triangle (assuming long strips)
    - 2 triangles per vertex (on average)
    - about 4 bytes per triangle (on average)

- **total is 20 bytes per vertex (limiting best case)**
  - factor of 3.6 over separate triangles; 1.8 over indexed mesh
Triangle fans

• Same idea as triangle strips, but keep oldest rather than newest
  – every sequence of three vertices produces a triangle
  – e.g., 0, 1, 2, 3, 4, 5, … leads to
    (0 1 2), (0 2 3), (0 3 4), (0 4 5), …
  – for long fans, this requires about one index per triangle

• Memory considerations exactly the same as triangle strip
Triangle strips gain efficiency by caching the most recent two vertices.

- We are essentially using a FIFO cache policy with a size of 2.
- Cache miss rate approaches 1 miss / triangle.
Optimizing for larger caches

With indexed meshes, saving indices is less important

- we store lots of data at vertices; ~6 indices is the least of our worries
- just putting meshes in triangle-strip order gives you the same vertex caching behavior (“transparent” vertex caching)

GPU pipelines are built with post-transform vertex caches

- cache the results of the vertex processing stage
- cache hits can save substantial computation
- (for parallelism newer systems process primitives in batches, but the effect is similar)

As with other applications of caches, now order of data access matters
$r$ is the average cache miss rate
[Sander et al. 2007 “Fast Triangle Reordering for Vertex Locality and Reduced Overdraw”]
Mesh simplification

Many ways to simplify meshes

- remove chunks, retriangulate hole
- quantize vertices to centers of voxels

Particularly simple and effective is edge collapses, or edge contractions:
Quadric Error Metric

Edge-collapse simplification produces a sequence of meshes

- each mesh has one fewer face
- each is derived from the previous by a single edge collapse

Key question: where to put the vertex after the collapse?

- at first vertex? at second? at midpoint?
- can choose location as the solution to an optimization
Where to put the new vertex?

It depends on the mesh geometry:

- one way to formalize: the new vertex should be close to the planes of the triangles around it before the edge collapse.
A particularly convenient error metric: sum of squared distances to planes

- each plane has an equation, can be represented as a 4-vector \((a, b, c, d)\)
  with \((a, b, c)\) components normalized

- distance of a vertex \(v\) from the plane \(p\) is then the inner product \(p^Tv\)

- squared distance from plane is in the form \(v^TMv\) for a 4x4 \(M\) (a quadric)

\[
\Delta(v) = \sum_{p \in \text{planes}(v)} (v^Tp)(p^Tv) \\
= \sum_{p \in \text{planes}(v)} v^T(p^Tp)v \\
= v^T \left( \sum_{p \in \text{planes}(v)} K_p \right) v
\]

- and better yet, the sum-squared distance from several planes is still in the form \(v^TQv\)
QEM simplification

With the error in the form of a quadric per vertex:

- the matrix is easy to compute from the surrounding triangles
- the error is easy to optimize. Given $Q_1$ and $Q_2$ belonging to a pair of vertices $v_1$ and $v_2$, we simply sum the errors of the two vertices:

$$\Delta(v) = \Delta_1(v) + \Delta_2(v)$$
$$= v^T Q_1 v + v^T Q_2 v$$
$$= v^T (Q_1 + Q_2) v$$

- minimizing this error is a 4x4 linear system—very fast
- algorithm
  - 0. compute $Q$s for all vertices, compute errors for all potential edge collapses.
  - 1. use priority queue to find smallest-error edge. Collapse it; update the neighboring $Q$s.
  - 2. repeat until mesh is small enough!
69k faces

1k faces

surfaces of constant cost for reducing to 999 faces

[Garland & Heckbert 1997 “Surface Simplification Using Quadric Error Metrics”]
Continuous level-of-detail: Progressive Meshes

Key observation: edge collapse is invertible

- just need to store (offsets to) the locations of the two new vertices

Thus a sequence of edge collapses, reversed, is a representation for a mesh

\[
\begin{align*}
\hat{M} = M^n \quad &\xrightarrow{\text{ecol}_n^{-1}} \ldots \quad &\xrightarrow{\text{ecol}_1} \quad &M^1 \quad &\xrightarrow{\text{ecol}_0} \quad &M^0. \\
M^0 \quad &\xrightarrow{\text{vsplit}_0} \quad &M^1 \quad &\xrightarrow{\text{vsplit}_1} \ldots \quad &\xrightarrow{\text{vsplit}_{n-1}} \quad &(M^n = \hat{M})
\end{align*}
\]

[Hoppe 1996 “Progressive Meshes”]
Progressive Meshes

**Store full representation, load various levels of detail**
- just load or transmit a prefix of the list of edge splits
- can change level of detail smoothly depending on size/distance/salience/etc.

**Can interpolate (“geomorphs”)**
- sudden edge splits/collapses are jarring
- interpolate new vertices from merged position to new positions
- leads to truly continuous LoD

**Extra details (of QEM and PM)**
- boundaries, creases—want to preserve them
- merging of small pieces—otherwise can’t simplify enough
- maintenance of additional attributes—throw them in the metric too
LOD  0.000  #Faces  122

[Hoppe 1996 “Progressive Meshes”]
[Hoppe 1996 “Progressive Meshes”]