Mesh Representation and Decimation

COS 526: Advanced Computer Graphics

PRINCETON
UNIVERSITY

Slide credits: Tom Funkhouser, Ravi Ramamoorthi, Keenan Crane, Hugues Hoppe

Motivation

- We want to do operations on meshes
  - Rendering
  - Simplification
  - Computational geometry
  - Smoothing
  - Analysis
- Range from “graph-like” to “signal-processing-like”
- Best representations (mesh data structures)?

Desirable Characteristics for Mesh Data Structures

- Generality – from most general to least
  - Polygon soup
  - Only triangles
  - 2-manifold → ≤ 2 triangles per edge
  - Orientable → consistent CW / CCW winding
  - Closed → no boundary
- Compact storage

Desirable Characteristics for Mesh Data Structures

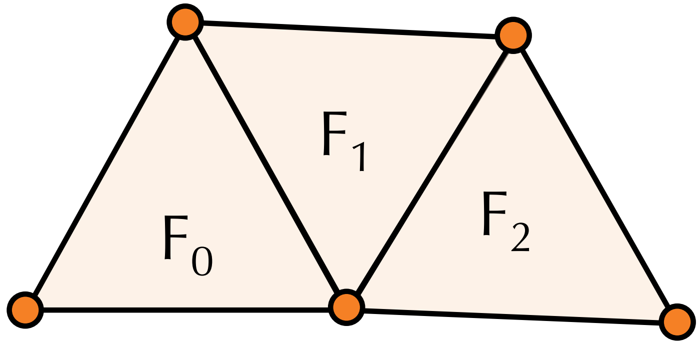
- Efficient support for operations:
  - Given face, find its vertices
  - Given vertex, find faces touching it
  - Given face, find neighboring faces
  - Given vertex, find neighboring edges or vertices
  - Given edge, find vertices and faces it touches

Mesh Data Structures

- Independent faces
- Indexed face set
- Adjacency lists
- Winged-edge
- Half-edge

Independent Faces

- Faces list vertex coordinates
  - Redundant vertices
  - No topology information

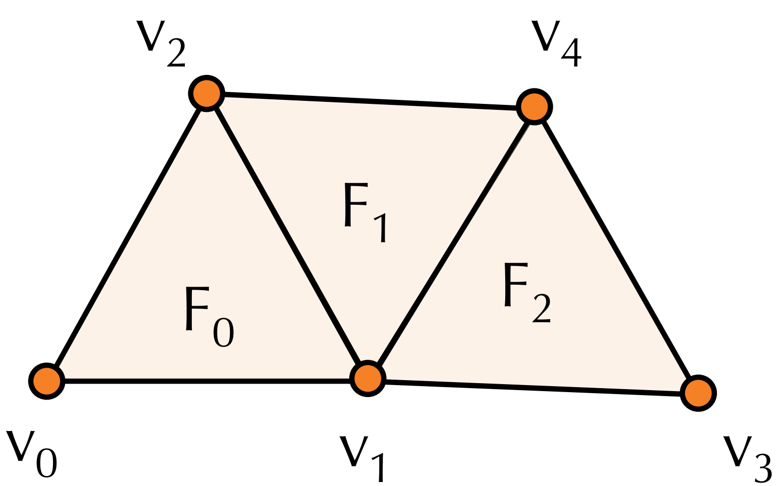


Face Table

F_0: (x_0,y_0,z_0), (x_1,y_1,z_1), (x_2,y_2,z_2)
F_1: (x_3,y_3,z_3), (x_4,y_4,z_4), (x_5,y_5,z_5)
F_2: (x_6,y_6,z_6), (x_7,y_7,z_7), (x_8,y_8,z_8)

Indexed Face Set

- Faces list vertex references – “shared vertices”

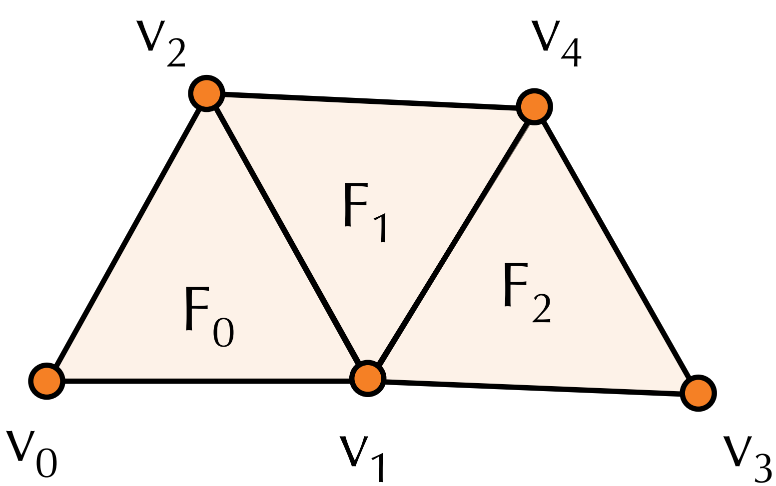


Vertex Table	Face Table
v_0: (x_0,y_0,z_0)	F_0: 0, 1, 2
v_1: (x_1,y_1,z_1)	F_1: 1, 4, 2
v_2: (x_2,y_2,z_2)	F_2: 1, 3, 4
v_3: (x_3,y_3,z_3)	
v_4: (x_4,y_4,z_4)	

Note CCW ordering

Indexed Face Set

- Storage efficiency?
- Which operations supported in O(1) time?



Vertex Table
v_0: (x_0,y_0,z_0)
v_1: (x_1,y_1,z_1)
v_2: (x_2,y_2,z_2)
v_3: (x_3,y_3,z_3)
v_4: (x_4,y_4,z_4)

Face Table
F_0: 0, 1, 2
F_1: 1, 4, 2
F_2: 1, 3, 4

Note CCW ordering

Efficient Algorithm Design

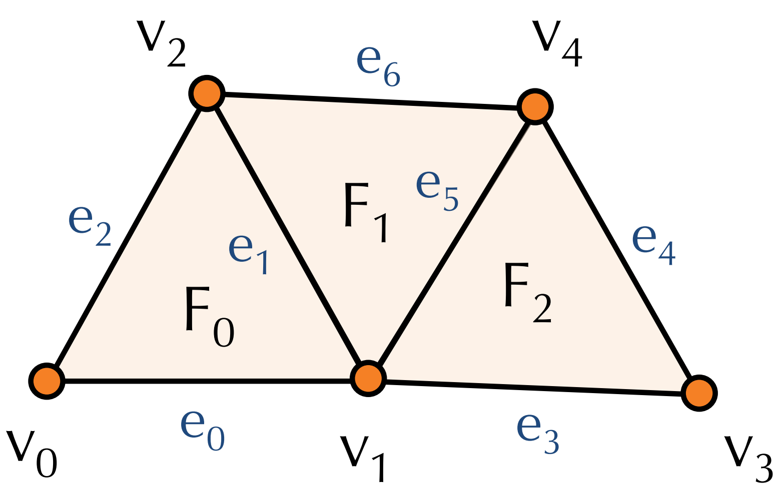
- Can *sometimes* design algorithms to compensate for operations not supported by data structures
- Example: per-vertex normals
  - Average normal of faces touching each vertex
  - With indexed face set, vertex $\rightarrow$ face is $O(n)$
  - Naive algorithm for all vertices: $O(n^2)$
  - Can you think of an $O(n)$ algorithm?

Efficient Algorithm Design

- Can *sometimes* design algorithms to compensate for operations not supported by data structures
- Example: per-vertex normals
  - Average normal of faces touching each vertex
  - With indexed face set, vertex $\rightarrow$ face is $O(n)$
  - Naive algorithm for all vertices: $O(n^2)$
  - Can you think of an $O(n)$ algorithm?
- For other operations, useful to have vertex $\rightarrow$ face (and/or other) adjacency lookup be $O(1)$

Full Adjacency Lists

- Store all vertex, face, and edge adjacencies



<u>Edge Adjacency Table</u>

e_0: v_0, v_1; F_0,$\varnothing$; $\varnothing$,e_2,e_1,$\varnothing$
e_1: v_1,v_2; F_0,F_1; e_5,e_0,e_2,e_6
⋮

<u>Face Adjacency Table</u>

F_0: v_0,v_1,v_2; F_1,$\varnothing$,$\varnothing$; e_1,e_2,e_0
F_1: v_1,v_4,v_2; $\varnothing$,F_0,F_2; e_6,e_1,e_5
F_2: v_1,v_3,v_4; $\varnothing$,F_1,$\varnothing$; e_4,e_5,e_3

<u>Vertex Adjacency Table</u>

v_0: v_1,v_2; F_0; e_0,e_2
v_1: v_3,v_4,v_2,v_0; F_2,F_1,F_0; e_3,e_5,e_1,e_0
⋮

Full Adjacency: Issues

- "Lookup" operations are efficient
- Storage is expensive
- Updating data structures is *very* expensive
- For most applications, *partial* adjacencies are sufficient

Partial Adjacency Lists

- Store some adjacencies, use to derive others
- Many possibilities…

<u>Edge Adjacency Table</u>

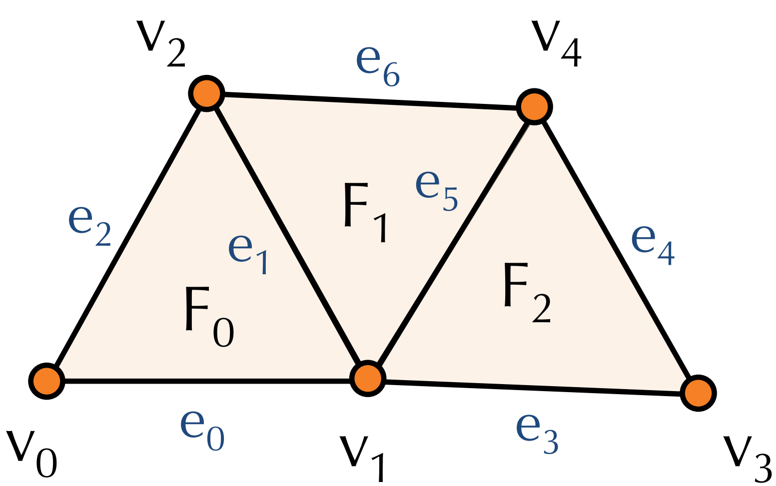
e_0: v_0, v_1; F_0,$\varnothing$; $\varnothing$,e_2,e_1,$\varnothing$
e_1: v_1,v_2; F_0,F_1; e_5,e_0,e_2,e_6
⋮

<u>Face Adjacency Table</u>

F_0: v_0,v_1,v_2; F_1,$\varnothing$,$\varnothing$; e_1,e_2,e_0
F_1: v_1,v_4,v_2; $\varnothing$,F_0,F_2; e_6,e_1,e_5
F_2: v_1,v_3,v_4; $\varnothing$,F_1,$\varnothing$; e_4,e_5,e_3

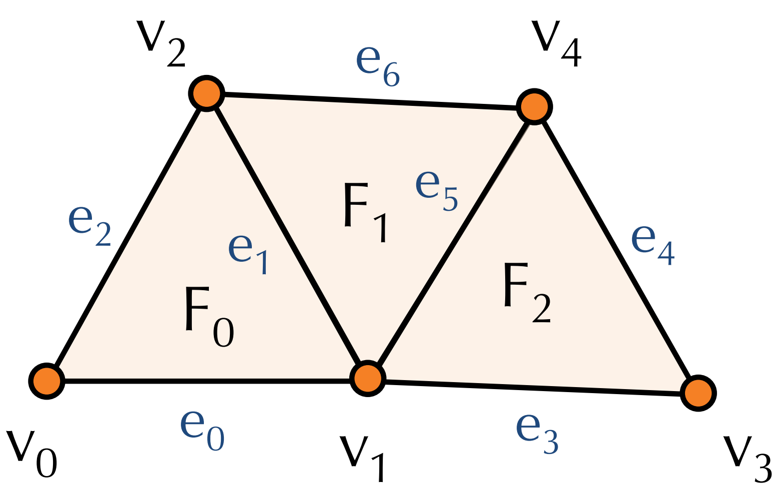
<u>Vertex Adjacency Table</u>

v_0: v_1,v_2; F_0; e_0,e_2
v_1: v_3,v_4,v_2,v_0; F_2,F_1,F_0; e_3,e_5,e_1,e_0
⋮



Partial Adjacency Lists

- Some combinations only make sense for closed manifolds



Edge Adjacency Table

e_0: v_0, v_1; F_0,$\varnothing$; $\varnothing$,e_2,e_1,$\varnothing$
e_1: v_1,v_2; F_0,F_1; e_5,e_0,e_2,e_6
⋮

Face Adjacency Table

F_0: v_0,v_1,v_2; F_1,$\varnothing$,$\varnothing$; e_1,e_2,e_0
F_1: v_1,v_4,v_2; $\varnothing$,F_0,F_2; e_6,e_1,e_5
F_2: v_1,v_3,v_4; $\varnothing$,F_1,$\varnothing$; e_4,e_5,e_3

Vertex Adjacency Table

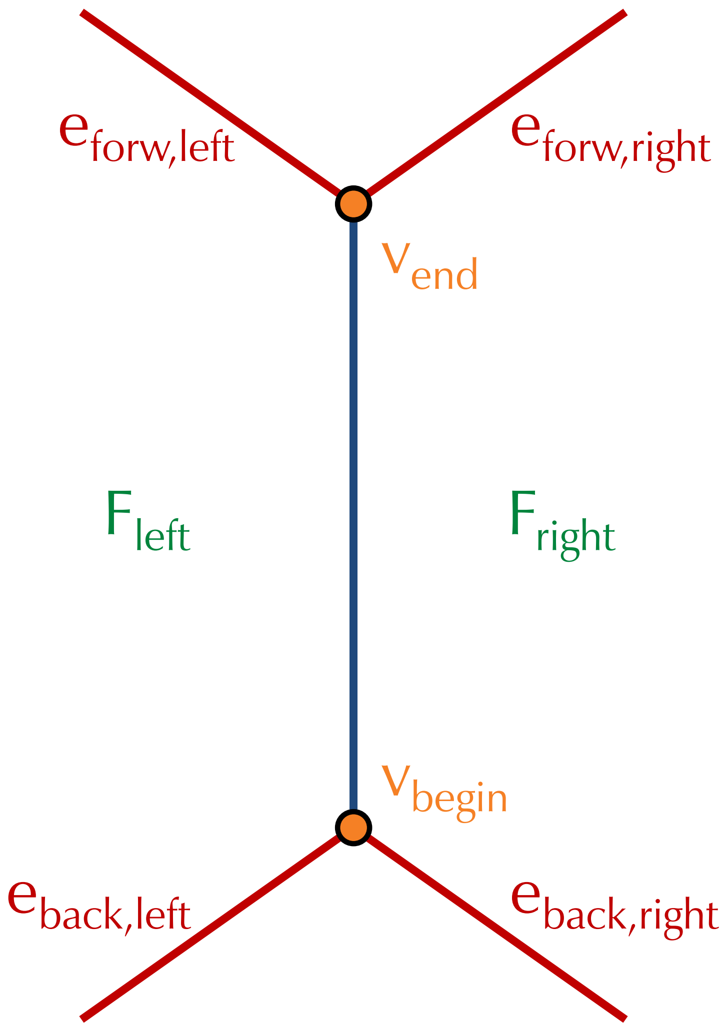
v_0: v_1,v_2; F_0; e_0,e_2
v_1: v_3,v_4,v_2,v_0; F_2,F_1,F_0; e_3,e_5,e_1,e_0
⋮

Winged, Half Edge Representations

- Most information associated with edges
  - Vertices, faces point to one edge each
- Compact Storage
- Many operations efficient
- Allow one to walk around mesh
- General for arbitrary polygons (not just triangles)
- But, relative to partial adjacency tables, updating can be more complex

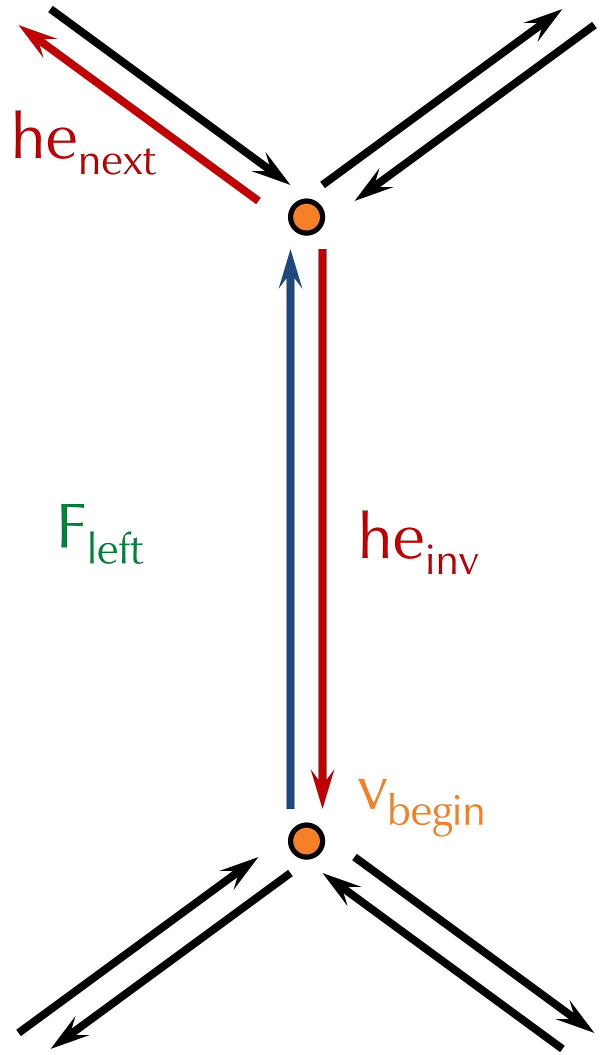
Winged Edge

- Each edge stores 2 vertices, 2 faces, 4 edges – fixed size
- Enough information to completely "walk around" faces or vertices
- Think how to implement
  - Walking around vertex
  - Finding neighborhood of face



Half Edge

- Instead of single edge, 2 directed "half edges"
- Each stores 1 vertex, 1 face, 2 half-edges
- Makes some operations more efficient



HalfEdge Data Structure (example)

```
class HalfEdge {              // Only one example, some critical functions
  public:
    HalfEdgeIter next;        // points to the next halfedge around the current face
    HalfEdgeIter flip;        // points to the other halfedge associated with this edge
    VertexIter vertex;        // points to the vertex at the "tail" of this halfedge
    EdgeIter edge;            // points to the edge associated with this halfedge
    FaceIter face;            // points to the face containing this halfedge
    bool onBoundary;          // true if this halfedge is contained in a boundary
                              // loop; false otherwise
} ;
```

From Keenan Crane's Geometry Processing code https://github.com/dgpdec/course

HalfEdge Walk Around Faces

```
int Vertex :: valence( void ) const {     // returns the number of incident faces
    int n = 0;
    HalfEdgeCIter h = he;                 // Start loop with half-edge for that vertex
    do {
      n++;                                // Increment Valence.  Other operations similar:
                                          // For area,  A += h -> face -> area() ;
      h = h->flip->next;                  // Next Face.  Why does this work?
    } while ( h != he );                  // Stop when loop is complete.
    return n;
}
```

From Keenan Crane's Geometry Processing code https://github.com/dgpdec/course

Mesh Decimation

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

Division, Viewpoint, Cohen

Mesh Decimation

- Reduce number of polygons
  - Less storage
  - Faster rendering
  - Simpler manipulation
- Desirable properties
  - Generality
  - Efficiency
  - Produces "good" approximation

Michelangelo's St. Matthew
Original model: ~400M polygons

Mesh Simplification Considerations

- Type of input mesh?
- Modifies topology?
- Continuous LOD?
- Speed vs. quality?

Mesh Decimation

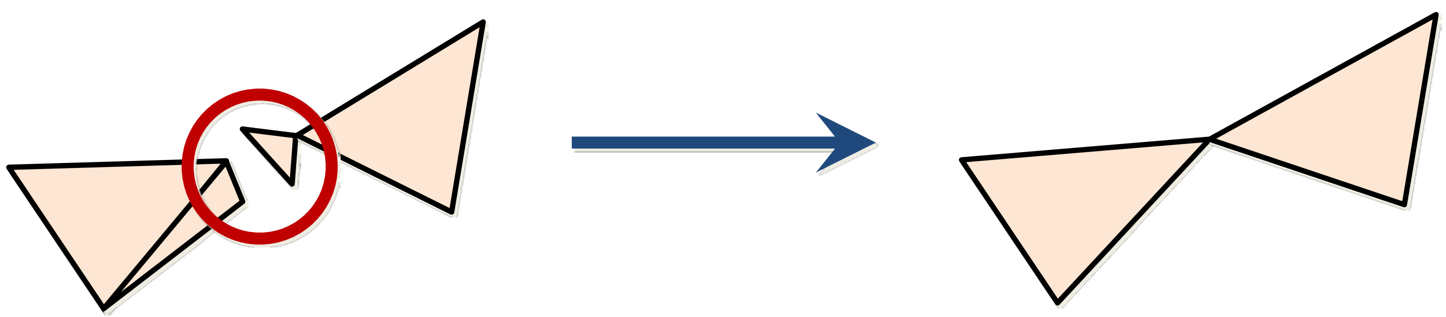
- Typical: greedy algorithm
  - Measure error of possible "simple" operations
  - Place operations in queue according to error
  - Perform operations in queue successively
  - After each operation, re-evaluate error metrics

Primitive Operations

- Simplify a bit at a time by removing a few faces
  - Repeated to simplify whole mesh
- Types of operations
  - Vertex cluster
  - Vertex remove
  - Edge collapse
  - Pair contraction

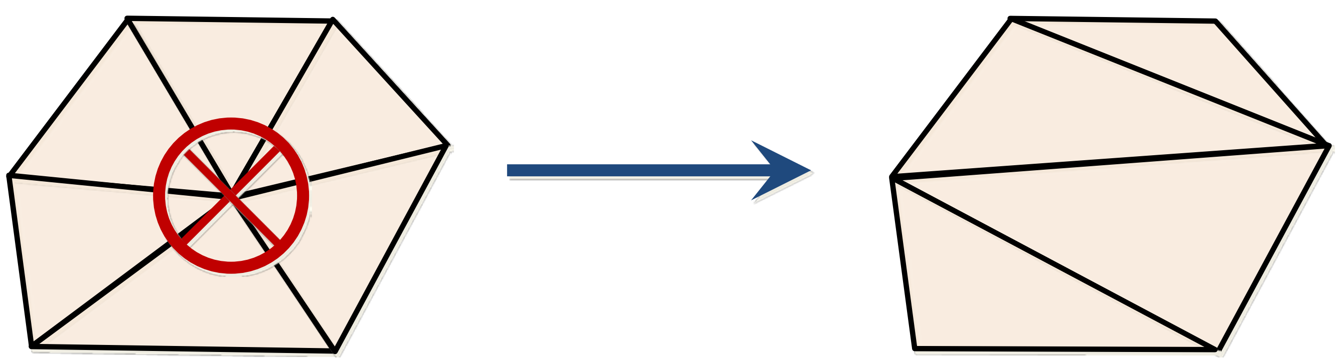
Vertex Cluster

- Method
  - Merge vertices based on proximity
  - Triangles with repeated vertices can collapse to edges or points
- Properties
  - General and robust
  - Can be unattractive if results in topology change



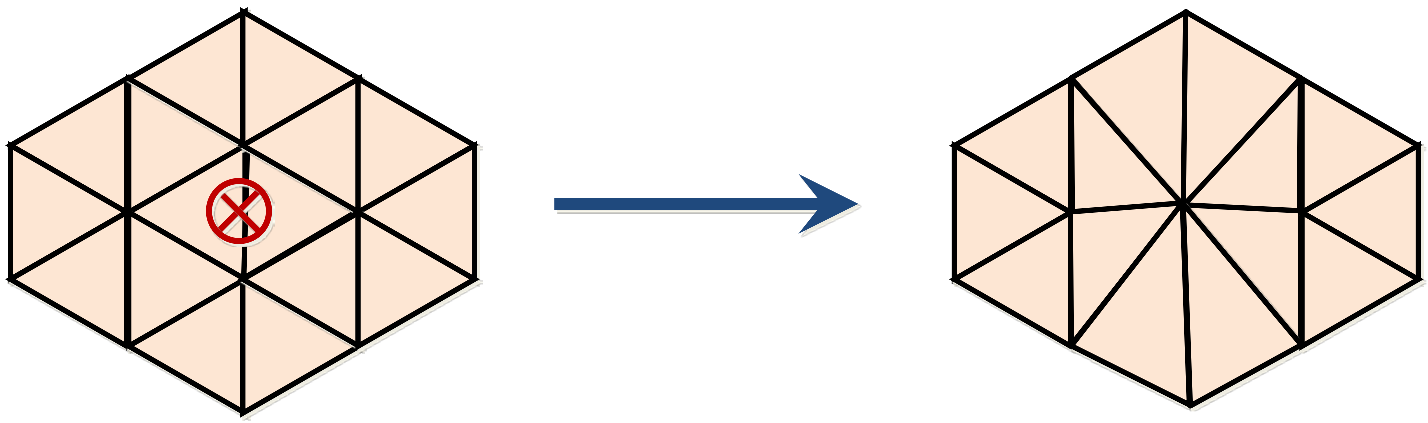
Vertex Remove

- Method
  - Remove vertex and adjacent faces
  - Fill hole with new triangles (reduction of 2)
- Properties
  - Requires manifold surface, preserves topology
  - Typically more attractive
  - Filling hole well not always easy



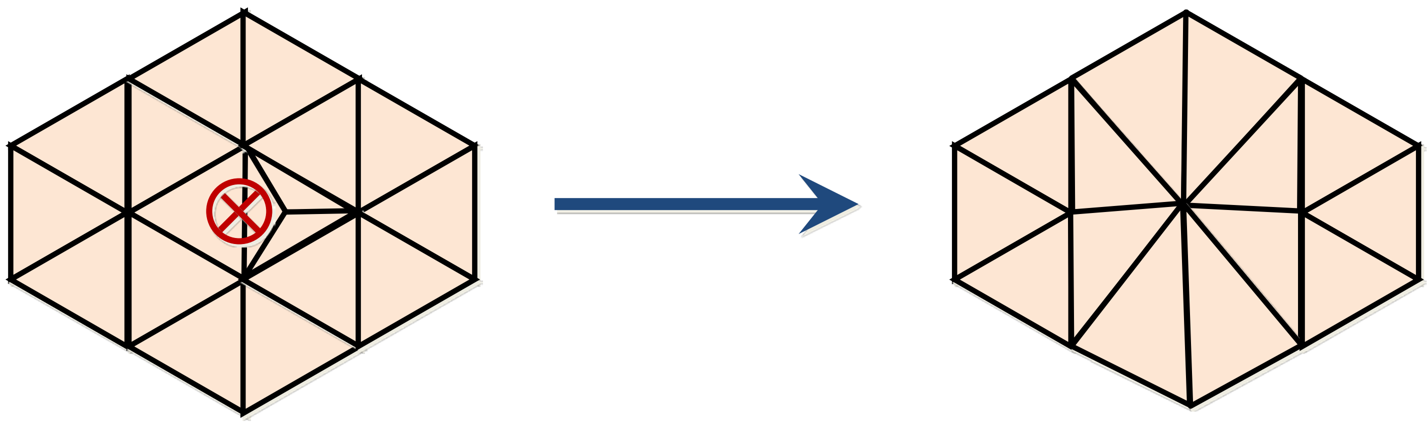
Edge Collapse

- Method
  - Merge two edge vertices to one
  - Delete degenerate triangles



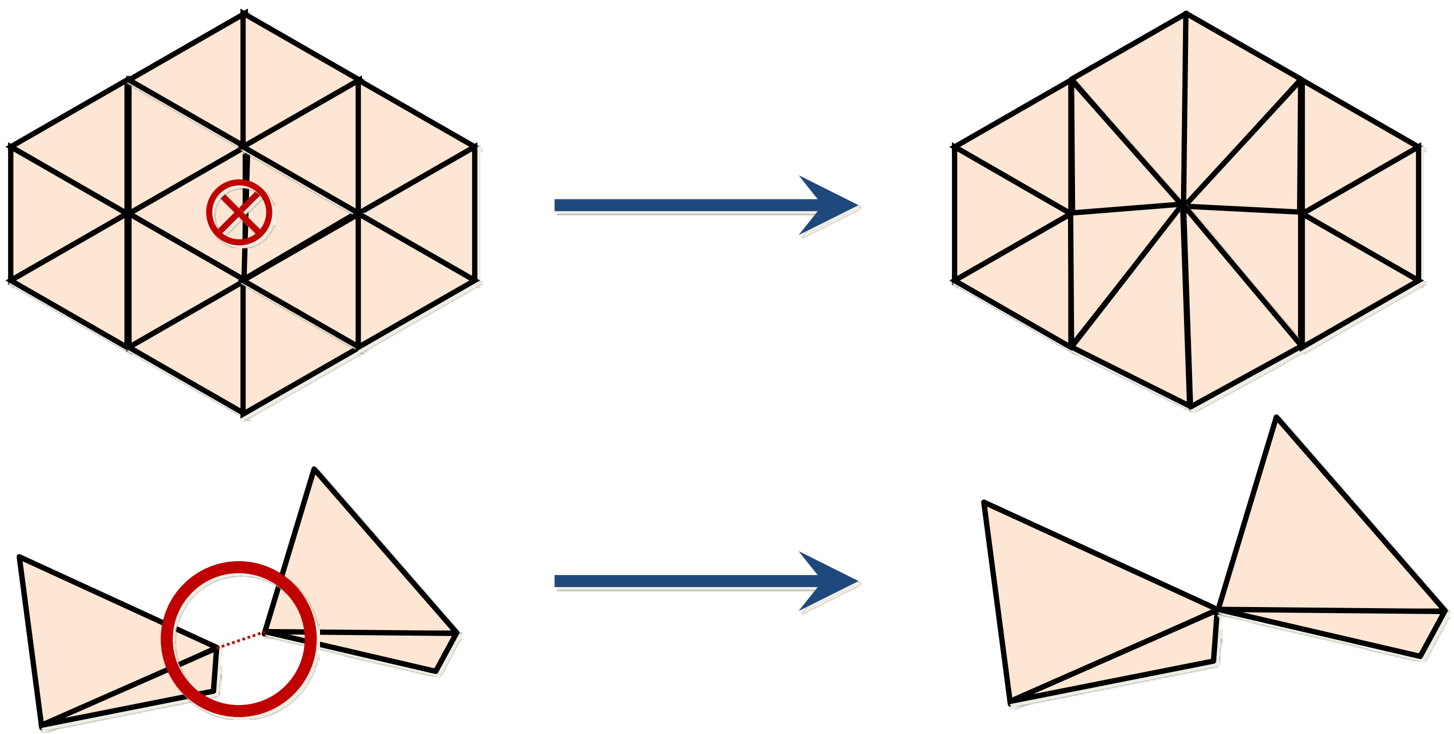
Edge Collapse

- Method
  - Merge two edge vertices to one
  - Delete degenerate triangles (warning: can be nontrivial!)
- Properties
  - Special case of vertex cluster
  - Allows smooth transition
  - Can change topology



Pair Contraction

- Generalization of edge collapse + vertex cluster: also allow nearby but disjoint regions to merge



Operation Considerations

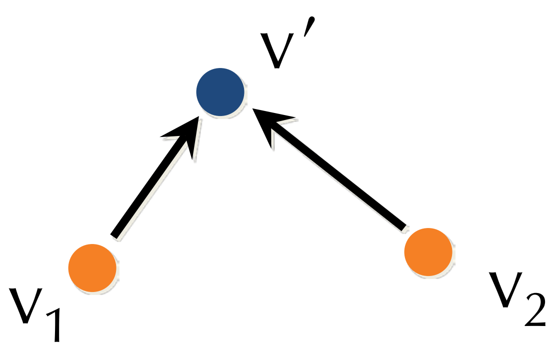
- Topology considerations
  - Attention to topology promotes better appearance
  - Allowing non-manifolds increases robustness and ability to simplify
- Operation considerations
  - Collapse-type operations allow smooth transitions
  - Vertex remove affects smaller portion of mesh than edge collapse

Geometric Error Metrics

- Motivation
  - Promote accurate 3D shape preservation
  - Preserve screen-space silhouettes and pixel coverage
- Types
  - Vertex-Vertex Distance
  - Surface-Surface Distance
  - Vertex-Surface Distance
  - Vertex-Plane Distance

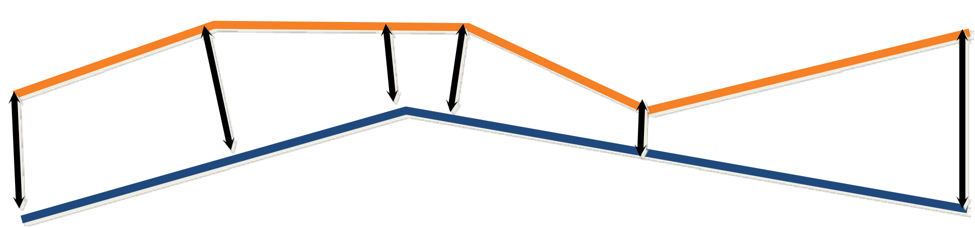
Vertex-Vertex Distance

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



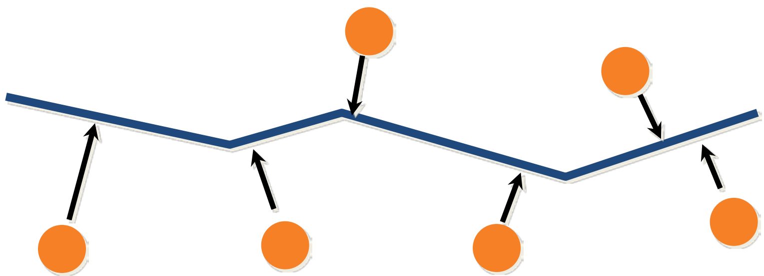
Surface-Surface Distance

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



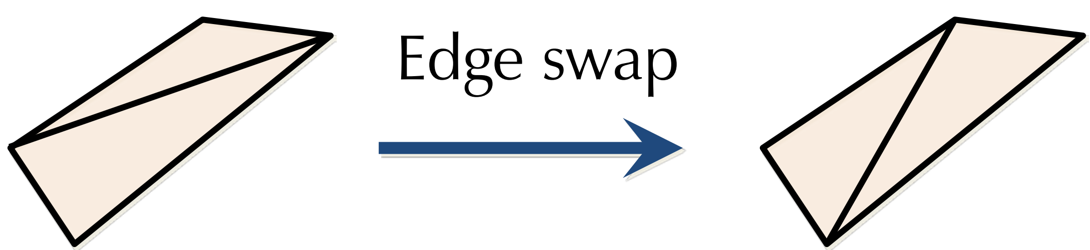
Veretx-Surface Distance

- For each original vertex, find closest point on simplified surface
- Compute sum of squared distances
- Faster approximation to surface-surface distance
  - But not the same: error is zero only at vertices and preserved edges



Geometric Error Observations

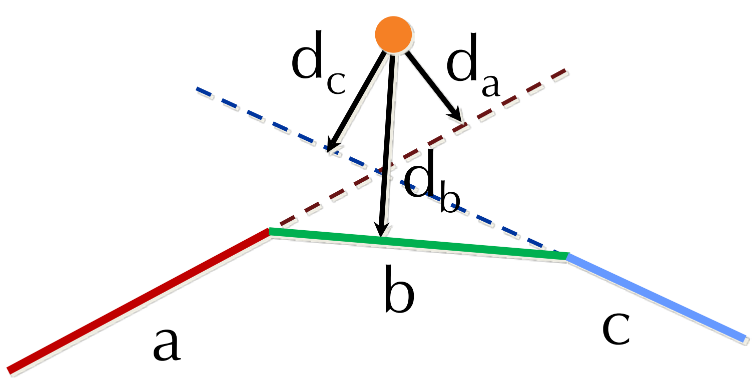
- Vertex-vertex and vertex-surface distance
  - Fast
  - Low error in practice, but not guaranteed by metric
- Surface-surface distance
  - Required for guaranteed error bounds



vertex-vertex $\neq$ surface-surface

Vertex-Plane Distance

- Store set of planes with each vertex
  - Error based on distance from vertex to planes
  - When vertices are merged, merge plane sets
- Error Quadrics
  - Store quadric form instead of explicit plane sets



Quadric Error Metrics

- Sum of squared distances from vertex to planes:

$$\Delta_{\mathbf{v}} = \sum_{\mathbf{p}} Dist(\mathbf{v},\mathbf{p})^2$$

$$\mathbf{v} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, \quad \mathbf{p} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

$$Dist(\mathbf{v},\mathbf{p}) = ax + by + cz + d = \mathbf{p}^{\mathrm{T}}\mathbf{v}$$

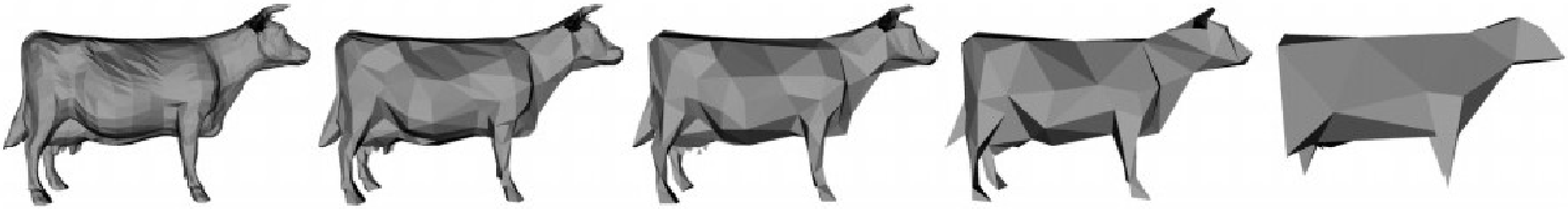
Quadric Error Metrics

$$\begin{aligned}\Delta &= \sum_{\mathbf{p}} (\mathbf{p}^{\mathrm{T}}\mathbf{v})^2 \\ &= \sum_{\mathbf{p}} \mathbf{v}^{\mathrm{T}}\mathbf{p}\mathbf{p}^{\mathrm{T}}\mathbf{v} \\ &= \mathbf{v}^{\mathrm{T}}\left(\sum_{\mathbf{p}} \mathbf{p}\mathbf{p}^{\mathrm{T}}\right)\mathbf{v} \\ &= \mathbf{v}^{\mathrm{T}}\mathbf{Q}\mathbf{v}\end{aligned}$$

- Common mathematical trick: quadratic form = symmetric matrix Q multiplied twice by a vector

Quadric Error Metrics

- Garland & Heckbert, SIGGRAPH 97
- Greedy decimation algorithm
- Pair collapse (allow edge + non-edge collapses)
- Quadric error metrics:
  - Evaluate potential collapses
  - Determine optimal new vertex locations



Using Quadrics

- Approximate error of edge collapses
  - Each vertex v has associated quadric Q
  - Error of collapsing v_1 and v_2 to v' is $v'^T Q_1 v' + v'^T Q_2 v'$
  - Quadric for new vertex v' is $Q' = Q_1 + Q_2$

Using Quadrics

- Find optimal location v′ after collapse:

$$\mathbf{Q}' = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ q_{14} & q_{24} & q_{34} & q_{44} \end{bmatrix}$$

$$\min_{\mathbf{v}'} \mathbf{v}'^{\mathrm{T}} \mathbf{Q}' \mathbf{v}' : \quad \frac{\partial}{\partial x} = \frac{\partial}{\partial y} = \frac{\partial}{\partial z} = 0$$

Using Quadrics

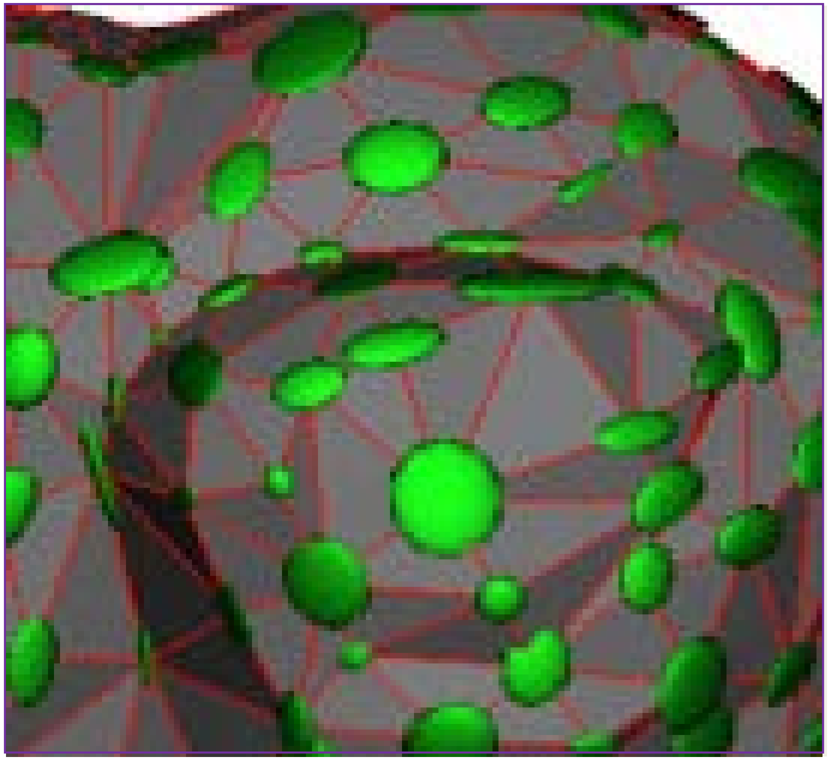
- Find optimal location v′ after collapse:

$$\begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{v}' = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

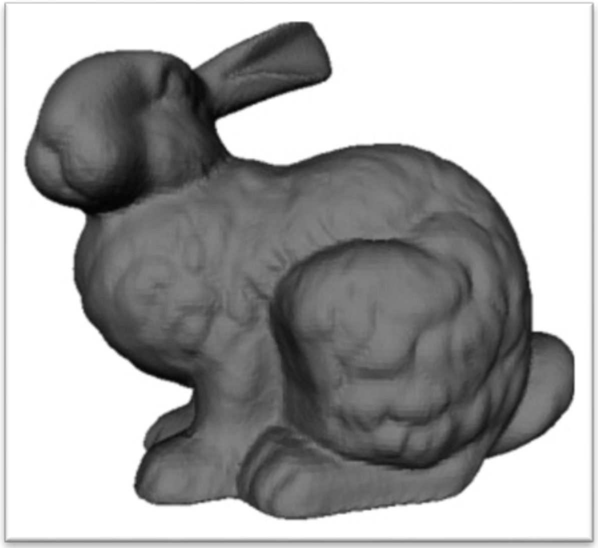
$$\mathbf{v}' = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Quadric Visualization

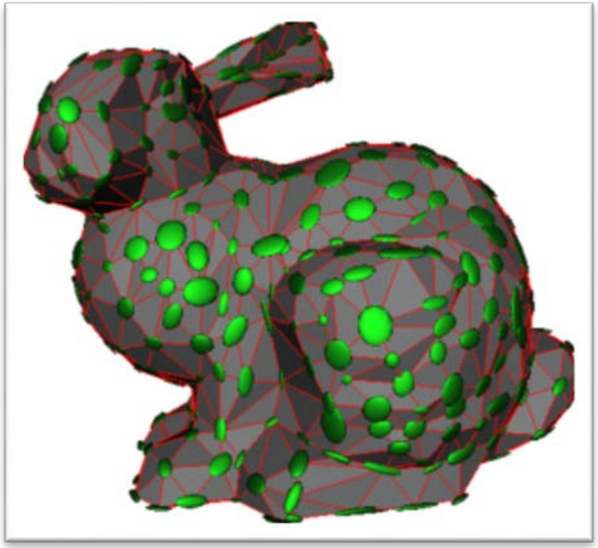
- Ellipsoids: iso-error surfaces
- Smaller ellipsoid = greater error for a given motion
- Lower error for motion parallel to surface
- Lower error in flat regions than at corners
- Elongated in "cylindrical" regions



Results



Original



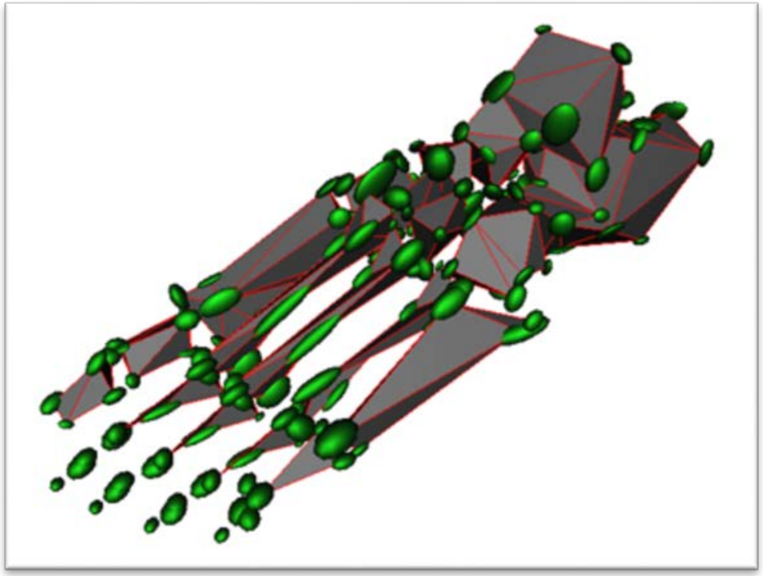
Quadrics

1k tris

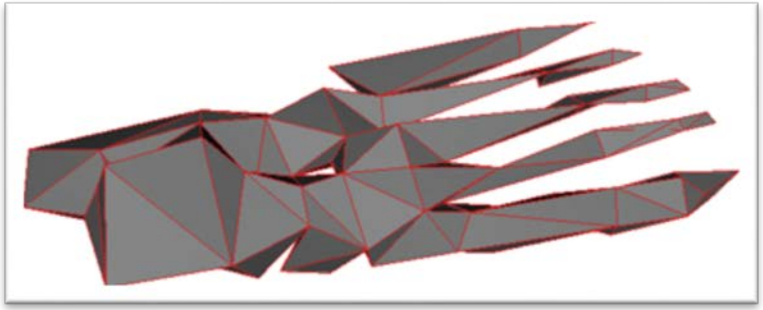
100 tris

Results

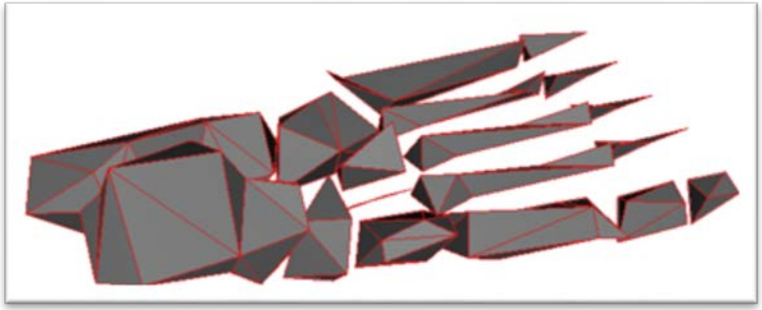
Original



Quadrics



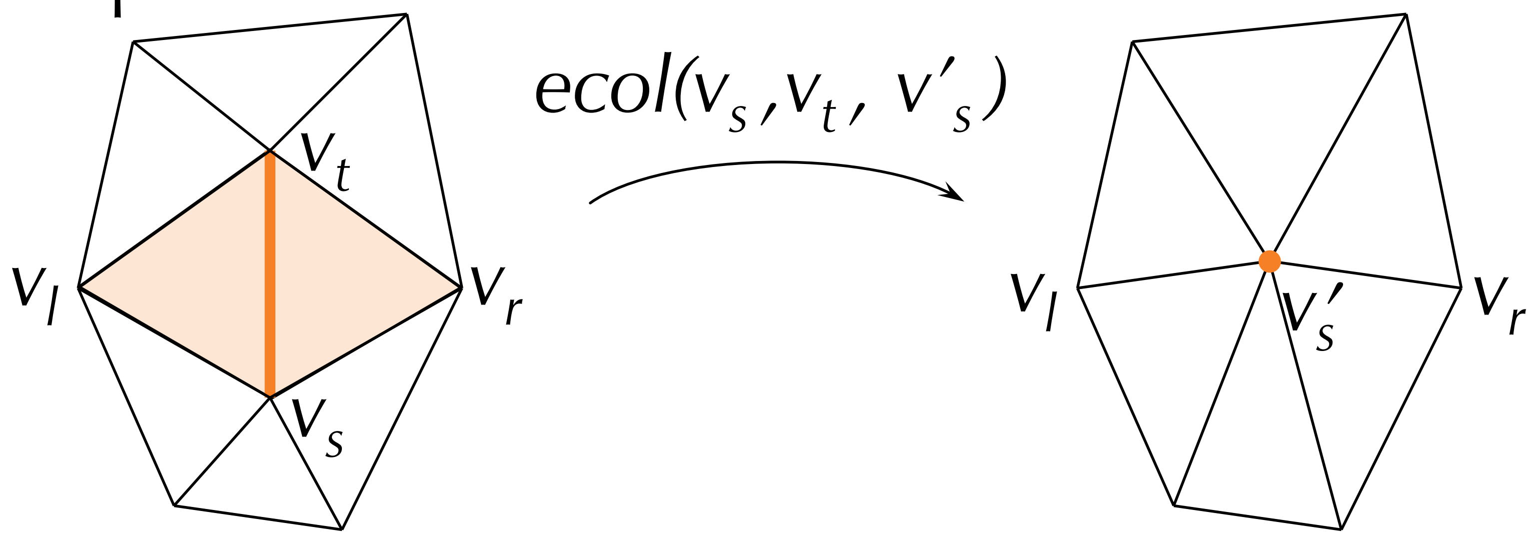
250 tris

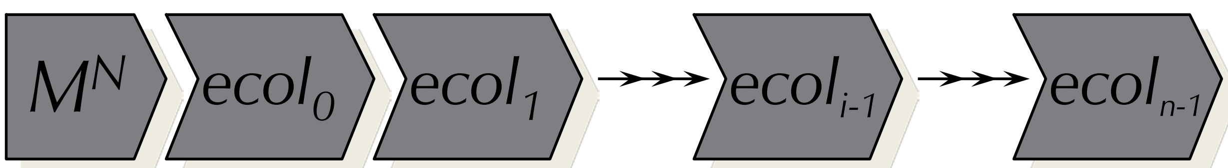


250 tris, edge collapses only

Progressive Mesh

- Encode continuous detail as sequence of edge collapses

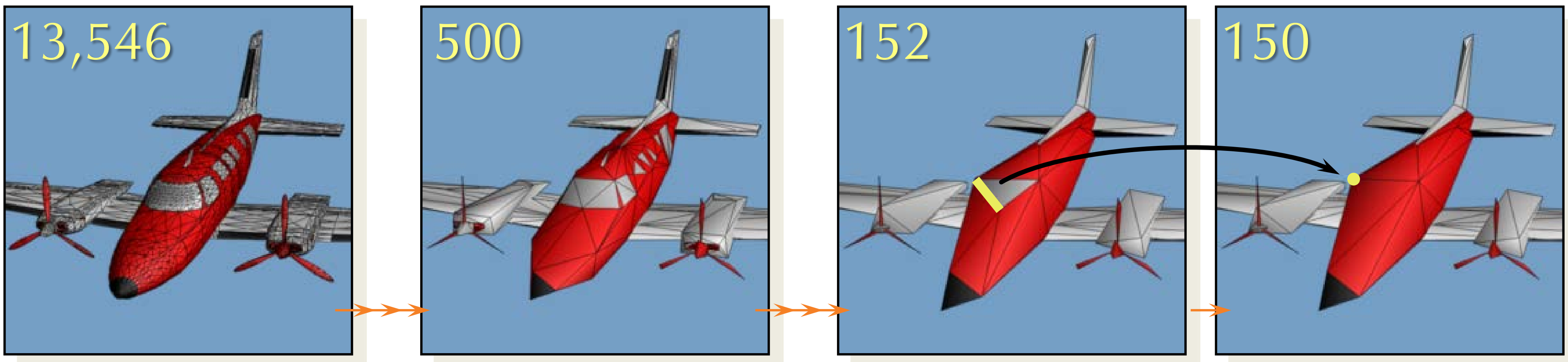




Hoppe

Progressive Mesh

- Simplification process



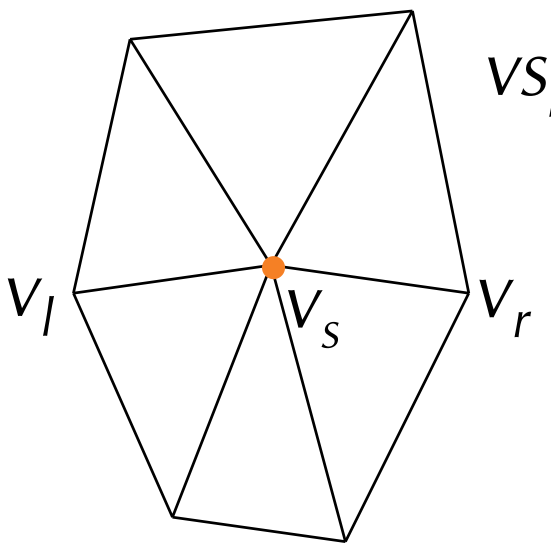
$$\hat{M} = M^{n} \xrightarrow[ecol_{n-1}]{} \twoheadrightarrow\!\!\to M^{175} \twoheadrightarrow\!\!\to \xrightarrow[ecol_{i}]{} M^{1} \xrightarrow[ecol_{0}]{} M^{0}$$

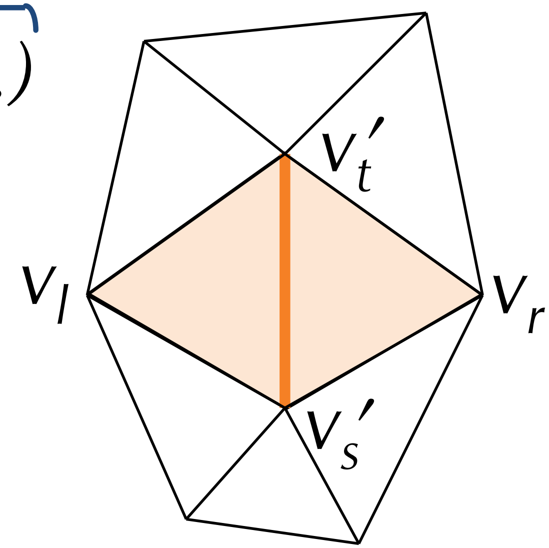
Hoppe

Progressive Mesh

- Inversion is possible with vertex split transformation

$$vspl(v_s, v_l, v_r, \overbrace{v_s', v_t', \ldots}^{attributes})$$

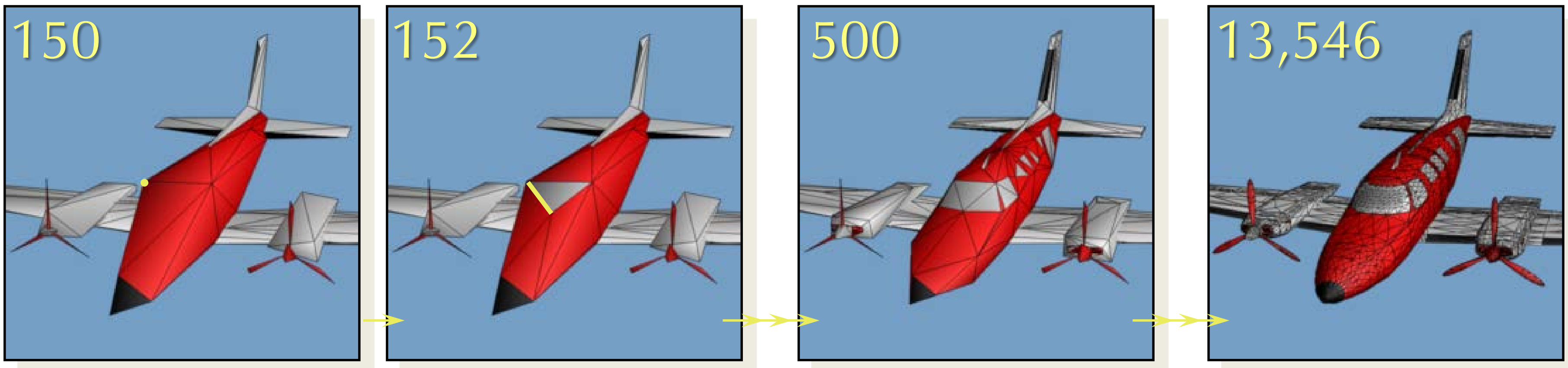




Hoppe

Progressive Mesh

- Reconstruction process

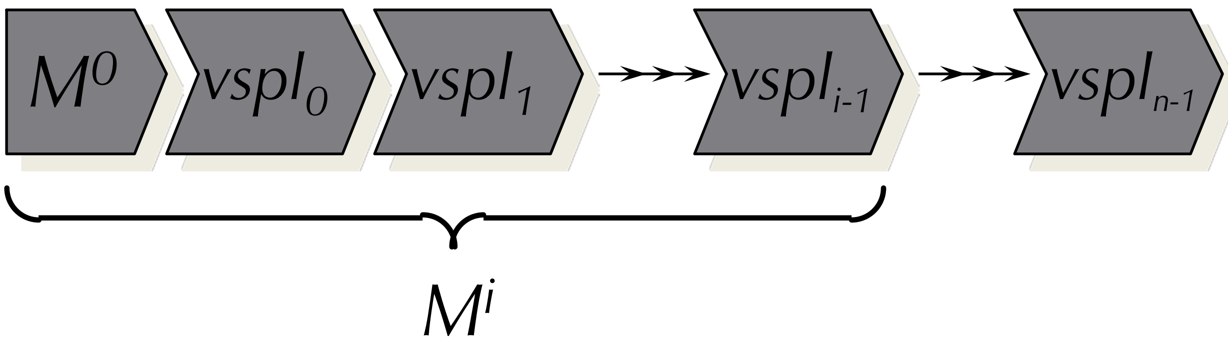


$$M^0 \xrightarrow{vspl_0} M^1 \longrightarrow \ldots \xrightarrow{vspl_i} \ldots M^{175} \longrightarrow \xrightarrow{vspl_{n-1}} M^n = \hat{M}$$

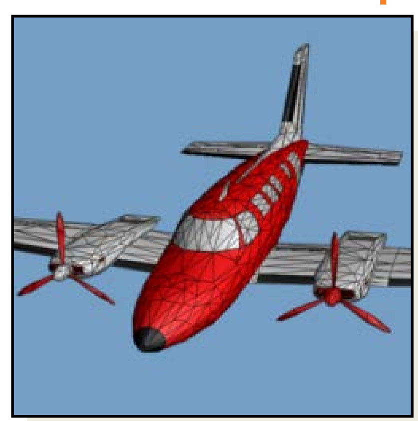
progressive mesh (PM) representation

Progressive Mesh

- From PM, extract M_i of any desired complexity (this is multiresolution)



3,478 faces? No problem



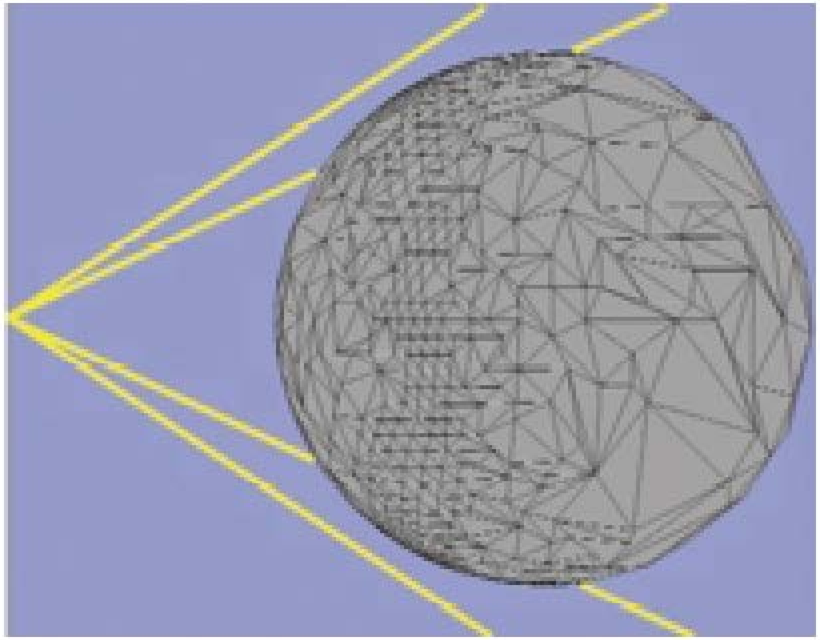
Hoppe

Progressive Mesh

Hoppe

View-Dependent Simplification

- Simplify dynamically according to viewpoint
  - Visibility
  - Silhouettes
  - Lighting



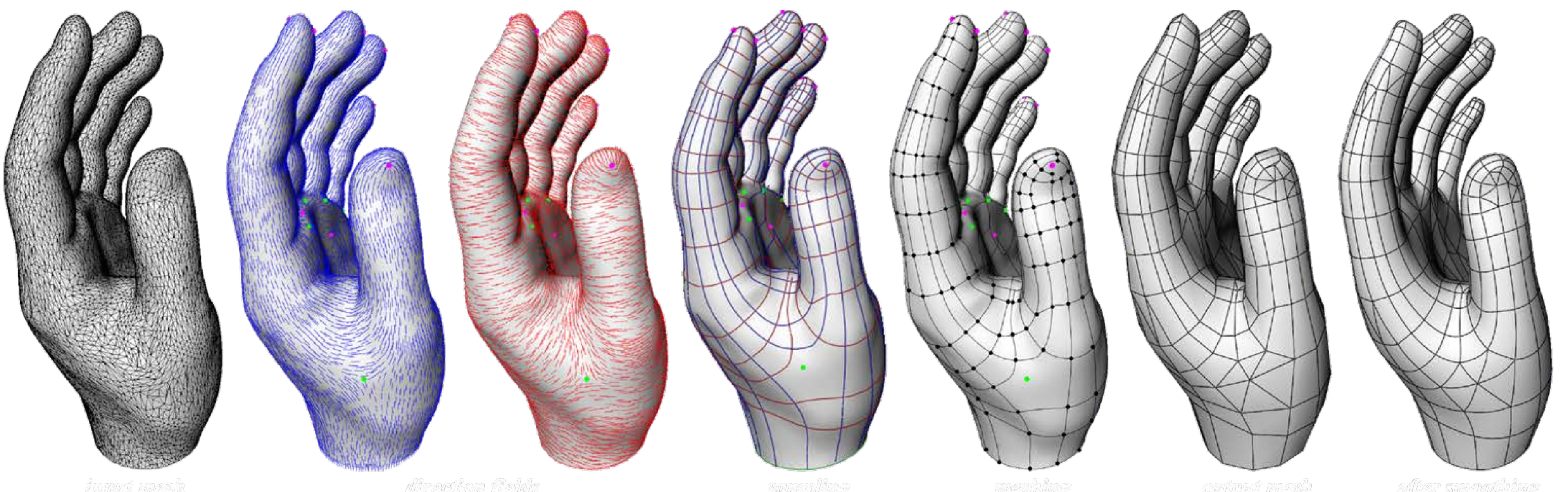
Hoppe

Remeshing

- Alternative to decimation
- Placing polygons to approximate shape vs. greedily removing polygons from a complex one
  - “Bottom up” vs. top-down
  - Usually better approximation at a low polygon count
  - Can place polygons in more “intuitive” places

Anisotropic Polygonal Remeshing

- Draw lines of curvature, place samples, connect
  [Alliez et al., SIGGRAPH 03]



Variational Shape Approximation

- Grow close-to-planar patches, polygonize
  [Cohen-Steiner et al., SIGGRAPH 04]