Mesh Filtering, Remeshing, and Simplification
Today

• Consider *(triangle)* meshes

• Basic operations using notions that you’ve already seen

• Three important applications :
  
  – Filtering (changing the vertex positions to denoise a mesh)
  
  – Remeshing (resample in a « smart » way)
  
  – Simplification (downsample in a « smart » way)

• We’ll see the last one (upsample in a « smart » way) next.
Triangle meshes: Reminder

- An array of vertex positions
  \[xyzxyzxyzxyzxyz\ldots\]
  size \(3 \times \text{nb of vertices}\)

- An array of triangle indices
  \[ijkijkijkijk\ldots\]
  size \(3 \times \text{nb of triangles}\)
  (ex in the picture: \(1,2,3,1,3,4,\ldots\))
Normals: Reminder

- Compute per-triangle normals
- Average normals of neighboring triangles to obtain vertex normals

\[ n_f = \frac{(p_1 - p_0) \wedge (p_2 - p_0)}{\| (p_1 - p_0) \wedge (p_2 - p_0) \|} \]
Laplacian operator : Reminder

Takes scalars defined on vertices, computes the Laplacian at each vertex

\[ \nabla^2 f(v_i) = \frac{1}{|v_i|} \sum_{e_{ij}} \frac{\cot(\alpha_{ij}) + \cot(\beta_{ij})}{2} (f(v_j) - f(v_i)) \]

Note : Once again, you may see the version without \(1/|v_i|\) here and there. Once again, the version without is the integrated operator (integrated over the area around vertex \(v_i\)), and the version with is the point-wise operator.
Curvature: Reminder

**Gaussian curvature:**

\[ K_i = \left( 2 \pi - \sum_{t \in T_1(i)} \gamma_i(t) \right) / A_i \]

Angle deficit around vertex \( i \)

Area around vertex \( i \)

**Mean curvature:**

\[ H_i = \frac{-1}{4 A_i} \sum_{i \in V_1(i)} \left( \cot(\alpha_{ij}) + \cot(\beta_{ij}) \right) (v_j - v_i) \cdot N_i \]

Half-edge data structure

```c
struct Halfedge {
    HalfedgeRef next_halfedge;
    HalfedgeRef opposite_halfedge;
    FaceRef face;
    VertexRef to_vertex;
};

struct Face {
    HalfedgeRef halfedge;
};

struct Vertex {
    HalfedgeRef outgoing_halfedge;
};

void enumerate_1_ring(Vertex *center) {
    HalfedgeRef h = outgoing_halfedge(center);
    HalfedgeRef hstop = h;
    do {
        VertexRef v = to_vertex(h);
        // do something with v
        h = next_halfedge(opposite_halfedge(h));
    } while (h != hstop);
}
```
Mesh Filtering
Laplacian filtering

\[ \forall i, v_i := v_i - \delta_i \]
Laplacian filtering

original

Uniform weights

\[ w_{ij} = 1 \]

Cotangent weights

\[ w_{ij} = \cot(\alpha_{ij}) + \cot(\beta_{ij}) \]

\[ v_i' = \sum_{j \in V_1(i)} w_{ij} v_j / \sum_{j \in V_1(i)} w_{ij} \]
Bilateral filtering
Bilateral filtering

(a) input mesh

(b) smoothed mesh

\[
v' = \frac{1}{w(v)} \sum_{t_i \in \eta} P_{t_i}(v) a_{t_i} G_{\sigma_s}(\|v - c_i\|) G_{\sigma_r}(\|v - P_{t_i}(v)\|)
\]
Bilateral filtering

- **original**
- **noise removal**
  - narrow spatial, narrow influence
  - \( \sigma_f = 2, \sigma_g = 0.2 \)
- **smooth small features**
  - narrow spatial, wide influence
  - \( \sigma_f = 2, \sigma_g = 4 \)
- **smooth large features**
  - wide spatial, wide influence
  - \( \sigma_f = 4, \sigma_g = 4 \)
Laplacian vs Bilateral

- Laplacian: one-ring vertex neighborhood → fast
- Bilateral: large neighborhoods → slow
- Laplacian: derived from « sound maths »
  - Mean curvature flow
- Bilateral: preservation of rich geometric frequencies
  - Ad-hoc inspiration from image filtering
Remeshing
Good meshes

Equal edge lengths
Equilateral triangles
Valence close to 6
Good meshes

Equal edge lengths
Equilateral triangles
Valence close to 6
Uniform vs. adaptive sampling
Good meshes

Equal edge lengths
Equilateral triangles
Valence close to 6
Uniform vs. adaptive sampling
Feature preservation
Good meshes

Equal edge lengths
Equilateral triangles
Valence close to 6
Uniform vs. adaptive sampling
Feature preservation
Alignment to curvature lines
Isotropic vs. anisotropic
Good meshes

Equal edge lengths
Equilateral triangles
Valence close to 6
Uniform vs. adaptive sampling
Feature preservation
Alignment to curvature lines
Isotropic vs. anisotropic

Triangles vs. quads
Element distribution
Element orientation
Element orientation
Applications

Well-shaped elements

for processing & simulation (numerical stability & efficiency)
A « simple » remeshing algorithm (Botsch 2004)

Avoid global parameterization
   Numerically very sensitive
   Topological restrictions

Avoid local parameterizations
   Expensive computations

Use local operators & projections
   Resampling of 100k triangles in < 5s
Local operations

- Edge Collapse
- Edge Split
- Edge Flip
- Vertex Shift
Specify target edge length $L$
Compute edge length range $[L_{\text{min}}, L_{\text{max}}]$

Iterate:
1. **Split** edges longer than $L_{\text{max}}$
2. **Collapse** edges shorter than $L_{\text{min}}$
3. **Flip** edges to get closer to valence 6
4. **Vertex shift** by tangential relaxation
5. **Project** vertices onto reference mesh
Split long edges / Collapse short edges

\[ L_{\text{max}} \]

\[ \frac{1}{2} L_{\text{max}} \quad \frac{1}{2} L_{\text{max}} \]

\[ |L_{\text{max}} - L| = \left| \frac{1}{2} L_{\text{max}} - L \right| \]

\[ \Rightarrow L_{\text{max}} = \frac{4}{3} L \]

\[ L_{\text{min}} \quad L_{\text{min}} \quad L_{\text{min}} \]

\[ \frac{3}{2} L_{\text{min}} \quad \frac{3}{2} L_{\text{min}} \]

\[ |L_{\text{min}} - L| = \left| \frac{3}{2} L_{\text{min}} - L \right| \]

\[ \Rightarrow L_{\text{min}} = \frac{4}{5} L \]
Flip to optimize for valence

Improve valences
  Avg. valence is 6 (Euler)
  Reduce variation

Optimal valence is
  6 for interior vertices
  4 for boundary vertices
Flip to optimize for valence

Improve valences
   Avg. valence is 6 (Euler)
   Reduce variation

Optimal valence is
   6 for interior vertices
   4 for boundary vertices

Minimize valence excess
\[ \sum_{i=1}^{4} \left( \text{valence}(v_i) - \text{opt\_valence}(v_i) \right)^2 \]
Vertex relocation

Local “spring” relaxation

Uniform Laplacian smoothing

Bary-center of one-ring neighbors

\[ c_i = \frac{1}{\text{valence}(v_i)} \sum_{j \in N(v_i)} p_j \]
Vertex reprojection

- Project vertices onto original reference mesh
- Assign position & interpolated normal
Results

Original

\((\frac{1}{2}, 2)\)

\((\frac{4}{5}, \frac{4}{3})\)
Feature preservation

Define features
  Sharp edges
  Material boundaries

Adjust local operators
  Don’t move corners
  Collapse only along features
  Don’t flip feature edges
  Project to feature curves
Adaptive remeshing

Precompute max. curvature on reference mesh
Target edge length locally determined by curvature
Adjust split / collapse criteria
Mesh Simplification
Adaptive tessellation

~150k triangles

~80k triangles
Level of Detail (LoD) rendering
Size/Quality compromise
Required for limited capacity hardwares
Grid-based Simplification

- Setup a uniform grid
- Average all vertices falling into a single bin
- Keep only non-degenerate triangles (whose corners fall into three different bins)
Grid-based Simplification

Representant point: using average position
Grid-based Simplification

Representant point: using median position
Grid-based Simplification

Representant point : using QEF minimizer
Priors on QEF

Squared distance between $x$ and plane $P_i$: 
\[ d^2(x, P_i) = \left( n_i^T x - d_i \right)^2 \]

Using homogeneous coordinates: 
\[ \tilde{x} = (x, 1) \quad \tilde{n}_i = (n_i, -d_i) \]

\[ d^2(x, P_i) = (\tilde{n}_i^T \tilde{x})^2 = \tilde{x}^T \tilde{n}_i \tilde{n}_i^T \tilde{x} =: \tilde{x}^T Q_i \tilde{x} \]

Symmetric 4x4 matrix (quadric)

The sum of squared distances to an arbitrary number of planes is still a quadric!

\[ E(x) = \sum_{t_i \in R} \tilde{x}^T Q_i \tilde{x} = \tilde{x}^T \left( \sum_{t_i \in R} Q_i \right) \tilde{x} := \tilde{x}^T Q \tilde{x} \]
Quadric Error Function (QEF) optimization

\[ E(x) = \sum_{t_i \in R} \bar{x}^T Q_i \bar{x} = \bar{x}^T \left( \sum_{t_i \in R} Q_i \right) \bar{x} := \bar{x}^T Q \bar{x} \]

- Easy to construct
- Easy to minimize:

\[ Q = \begin{bmatrix} A & -b \\ -b^T & c \end{bmatrix} \]

\[ \begin{array}{c}
Ax = b \\
\end{array} \]

- Solved using, e.g., SVD
Comparison

mean  median  QEF
Grid-based Simplification

- Extremely simple to implement (+)
- Very costly for large grids (-)
- Uniform level of detail (-)
Adaptivity

- Use an octree instead of a uniform grid
- Not much more complicated
- Allocate only cells around the input vertices (no memory wasted)
Edge-decimation algorithms

- Uses simple concepts already seen (edge collapse, QEF)
- Iterative
- Continuous level of detail
- Simple to implement
- The basis for lots of algorithms
Edge-decimation algorithms

- Pick an edge \((v_i, v_j)\), and collapse it to a new vertex \(v\). 2 adjacent triangles are removed.
- The cost of a collapse is given by its QEF (a QEF is associated with each initial input vertex, and the QEF of a new vertex is the sum of the QEFs of the collapsed vertices). The new vertex position is the minimizer of the QEF

\[ p_i^T Q_i p_i = 0, \quad i = \{1, 2\} \]

\[ Q_3 = Q_1 + Q_2 \]

\[ \text{solve } v_3^T Q_3 v_3 = \min \]

\[ < \varepsilon \quad \rightarrow \quad \text{ok} \]
Edge-decimation algorithms
Edge-decimation algorithms
Edge-decimation algorithms
Edge-decimation algorithms
Edge-decimation algorithms
Edge-decimation algorithms
Edge-decimation algorithms
Edge-decimation algorithms
Edge-decimation algorithms
Edge-decimation algorithms
Progressive meshes

- It is very easy to record all edge collapse operations (and reverse as well)
Progressive meshes

- Noting $M(k+1)$ the mesh obtained from decimating one edge of mesh $M(k)$, and $M_0$ the input mesh,
- All meshes $\{M(k)\}_k$ can be efficiently stored as $M_0$ and a set of $N$ small collapse/split operations (for $N$ vertices in $M_0$)
- The hierarchy defines a « progressive mesh » : 

![Diagram showing progressive meshes](image)
Extension to animated meshes

- Simply sum the QEFs over all the input frames, and decimate jointly an edge in all the frames.