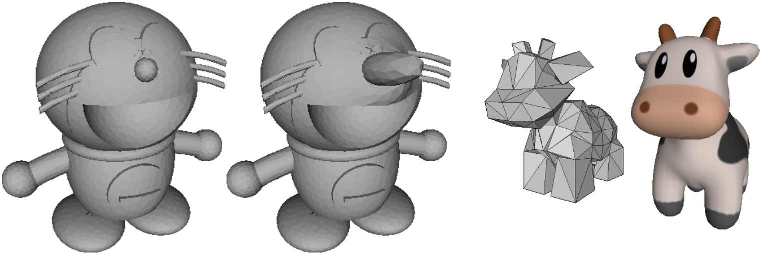
Geometric Modeling II: Mesh Processing



Content

Introduction

Pipeline

Data-Structures

Evaluation and Interrogation

Processing

Libraries and software

Conclusion

Parametric Representation

Represent a surface as a continuous function from a bounded domain $U \subset R^2$ to $S \subset R^3$.

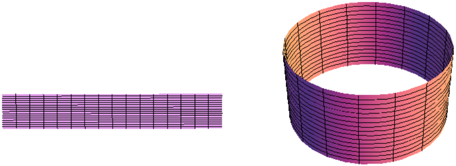


Figure: The rectangle $[0, 2\pi] \times [0, 1]$ mapped to a cylinder.

Parametric Representation

- Practically, it is rare to find a single function parametrizing the whole surface
- Instead, a surface is generally represented as a collection of functions from simple 2D domains to 3D
- Typical models usually consist of a huge collection of such surface patches

Parametric Representation

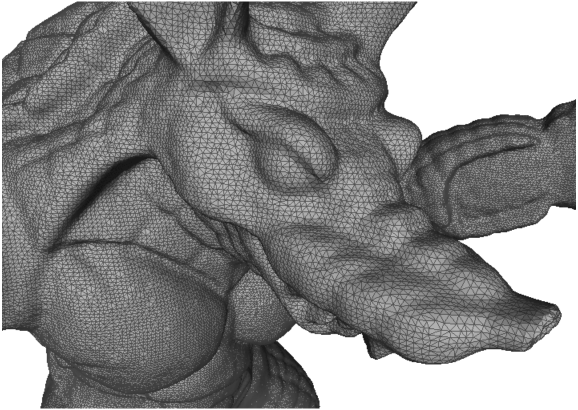
Precision: accuracy improves with refinement and depends on:

- The order of the polynomial used for the mapping: p; in the case of a polygon mesh $p = 1$ is fixed (linear functions are used);
- The length of the segment: h; the smaller / the more segments, the more precise the approximation

The approximation error behaves as $O(h^{p+1})$

Triangle Mesh

- Connectivity: how vertices are connected to form edges, faces
- Geometry: the positions of the vertices in space

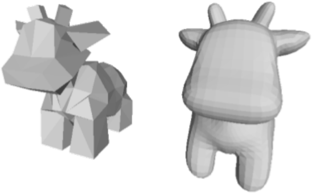


Triangle Mesh

Main advantages:

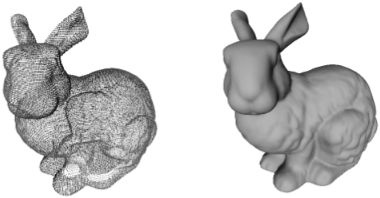
- Easy to enumerate points on the surface
- Easy to compute neighbors to a given vertex, edge, face
- Can represent object with arbitrary topology
- Flexibility for piecewise smooth surface and adaptive refinement
- Approximation error: $O(h^2)$ where h is edge segment
- Efficient rendering (rasterizer)

Mesh construction

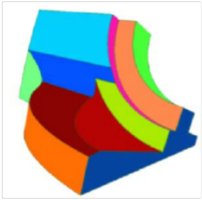


From subdivision surface
(Catmull-Clark)

From parametric surface
(Bezier)

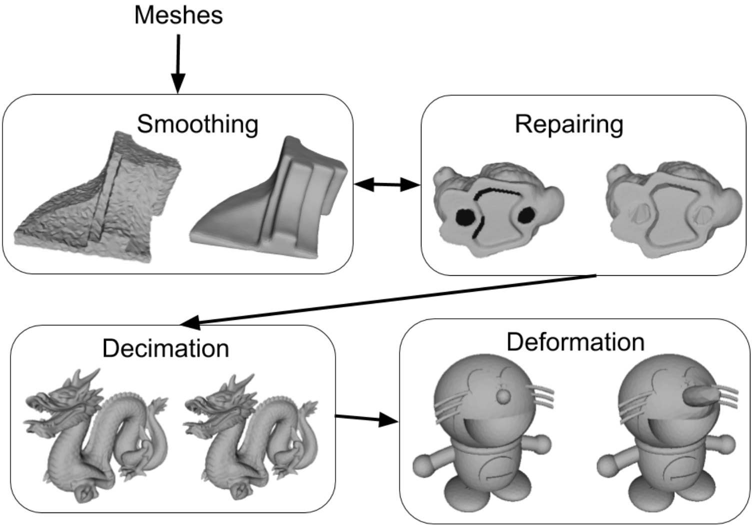


Scanned, registration,
reconstruction



From a CAD package

Mesh processing pipeline



Triangle Soup

- List of faces
- Each face represented independently by the coordinates of its vertices

```
Number of triangles F
x11 y11 z11 x12 y12 z12 x13 y13 z13
...
xf1 yf1 zf1 xf2 yf2 zf2 xf3 yf3 zf3
```

Example: STL file format used for rapid prototyping (3D printing)

Triangle Soup Data-Structure

```
struct _Triangle {
  struct _Vector3 _v0;
  struct _Vector3 _v1;
  struct _Vector3 _v2;
};

struct _TriMesh {
  int _number_triangles;
  struct _Triangle* _triangles;
};
```

Triangle Soup

Advantages:

- Simple and general
- Sufficient for visualization (no smooth shading), 3D printing

Inconvenients:

- Redundant information
- No connectivity information stored

Indexed Triangle Set

- List of vertices and faces
- Each vertex stores its coordinates (geometry)
- Each face stores indices to its vertices (topology)

Example: most of the mesh file formats: OFF, PLY, OBJ etc

Indexed Triangle Set

- Vertex List:

```
Number of vertices v
x11 y11 z11
...
xv1 yv1 zv1
```

- Triangle List:

```
Number of faces f
v11 v12 v13
...
vf1 vf2 vf3
```

Indexed Triangle List Data-Structure

```
struct _Triple {
  int _v0, _v1, _v2;
};

struct _TriMesh{
  int _number_vertices;
  int _number_triangles;

  struct _Vector3* _vertices;
  struct _Triple* _triangles;
};
```

Indexed Triangle Set

Advantages:

- Simple
- Reduce storage (because vertex information is not repeated)

Inconvenients:

- No explicit connectivity information
- No per-edge information (but can store attributes per vertex or face)

Half-Edge

- List of vertices, half-edges, faces;
- Each vertex stores its coordinates and an half-edge;
- Each face stores the neighbor half-edge;
- Each half-edge points to a starting vertex, the adjacent face, and the opposite, next (and sometimes also previous) half-edges.

Notes: an half-edge is oriented;

Half-Edge

List of vertices:

```
Number of vertices v
x11 y11 z11; e1
...
xv1 yv1 zv1; ev
```

List of half-edges:

```
Number of edges e
v1; f1; o1 n1 p1
...
ve; fe; oe ne pe
```

List of faces:

```
Number of faces f
e1
...
ef
```

Half-Edge Data-Structure

```
struct _Vertex {
  struct _Vector3 _coordinates;
  struct _HalfEdge* _he;
};

struct _HalfEdge {
  struct _Vertex* _start_vertex;
  struct _Face* _face;
  struct _HalfEdge *_opposite, *_next;
};
```

Half-Edge Data-Structure

```
struct _Face {
  struct _HalfEdge* _he;
};

struct _TriMesh {
  int _number_vertices;
  int _number_half_edges;
  int _number_faces;

  struct _Vertex* _vertices;
  struct _HalfEdge* _half_edges;
  struct _Face* _faces;
};
```

Half-Edge

Advantages:

- Reduced storage
- Fixed-size data-structure
- Connectivity information is easy to retrieve (for example: vertices neighbor to a given vertex, edges or faces incident to a given vertex, . . .)

Example: One Ring Traversal

One ring traversal: Given a vertex in the mesh, find all its neighboring vertices.

- Start at vertex
- Follow the outgoing half-edge
- Visit the neighbor
- Follow the opposite half-edge
- Follow the next half-edge
- ...

Example: One Ring Traversal

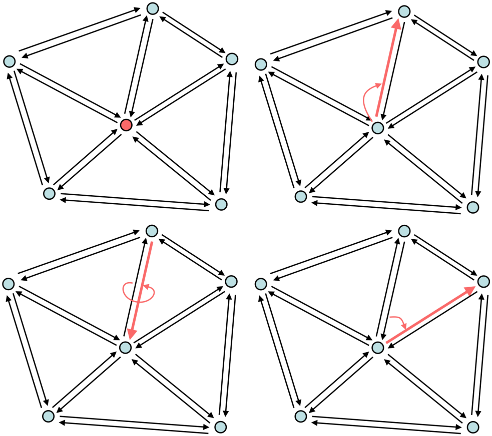


Figure: Some of the steps during the one-ring traversal of a given vertex

Example: One Ring Traversal

In pseudo-code:

```
HalfEdge* curr_he = curr_v->_he;
HalfEdge* start_he = curr_he;
do {
  // get a pointer to the neighbor vertex v
  Vertex* v = curr_he->_next->_start_vertex;

  // do something with the neighbor v
  // ...

  // go to the next following the half-edges
  curr_he = curr_he->_opposite->_next;
} while (curr_he != start_he);
```

One Ring Traversal: Half-Edge vs Indexed Triangle List

- Time complexity for one ring traversal using half-edge: $O(1)$,
- Using an indexed triangle list: $O(F)$ (F is the number of faces (triangles) in the mesh),
- For an indexed triangle list, possible to pre-compute adjacency tables (cost: $O(F)$), queries are then $O(1)$. But: if the connectivity changes, all the tables should be recomputed,
- Conclusion: If looking for neighbors (or incident edges or faces) to a vertex (or face or edge) is an operation that will be executed often and if the mesh connectivity can change, then it is worthwhile to use a more sophisticated data-structure like the half-edge structure.

Position evaluation

Coordinates of a point within a triangle are obtained from its barycentric coordinates:

$$(\alpha, \beta, \gamma) \to \alpha P_1 + \beta P_2 + \gamma P_3$$

where $0 \leq \alpha$, $0 \leq \beta$, $0 \leq \gamma$, $\alpha + \beta + \gamma = 1$ and P_1, P_2 and P_3 are the vertex coordinates of the triangle.

Face normal evaluation

Given the coordinates of the three vertices of a triangle: P_1, P_2 and P_3, the normal to the triangle (face normal) can be computed as:

$$n = \frac{(P_2 - P_1) \times (P_3 - P_1)}{||(P_2 - P_1) \times (P_3 - P_1)||}$$

where $\times$ is the cross product (vector product between two vectors) and $||.||$ is the Euclidean norm of a vector:

$$(x_1, y_1, z_1) \times (x_2, y_2, z_2) = (y_1 z_2 - y_2 z_1, z_1 x_2 - z_2 x_1, x_1 y_2 - y_1 x_2)$$

and

$$||(x, y, z)|| = \sqrt{x^2 + y^2 + z^2}$$

Dividing by the norm is required to have a unit normal. A unit normal is required e.g. for applying an illumination model (like Blinn-Phong).

Face normal evaluation: Example

A triangle has vertices $((0,0,0),(1,0,0),(1,1,0))$. A normal vector to the triangle has coordinates: $(0,0,1)$. It has already unit norm.

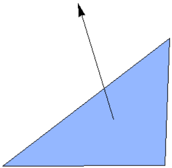


Figure: The triangle specified by its vertices: $((0,0,0),(1,0,0),(1,1,0))$ and its outer unit normal.

Vertex normal evaluation

At a vertex the situation is a bit different, because the underlying surface is in general smooth, while its approximation by a triangle mesh is C^1 discontinuous. There are different ways to solve the problem but in general they involve computing a weighted average of the normals of the neighbouring faces:

$$n(v) = \frac{\sum_{f \in Neigh(v)} w(v,f)n(f)}{||\sum_{f \in Neigh(v)} w(v,f)n(f)||}$$

where v is the vertex at which the normal $n(v)$ is to be computed, $Neigh(v)$ is the set of faces adjacent to v, $n(f)$ is the normal at a given face f and $w(v,f)$ is a weight (that may depend on v and f)

Vertex normal evaluation

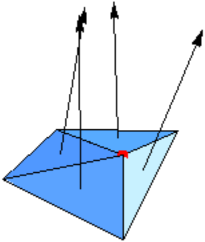


Figure: A vertex (red), its adjacent faces and their normal

Vertex normal evaluation: Weight

- Constant weight: $w(v, f) = 1$;
- Area weighted: $w(v, f) = Area(f)$ where $Area(f)$ computes the area of the face (triangle) f;
- Nelson Max's weight: $w(v, f) = \frac{sin\alpha_i}{||vv_i||||vv_{i+1}||}$, where v_i and v_{i+1} are the two other vertices in the face f and α_i is the angle between vv_i and vv_{i+1};
- Pseudo normal weight: $w(v, f) = \alpha_i$ where α_i, v_i and v_{i+1} have the same signification as above.

Shading

Normals are needed when applying an illumination model (such as the Phong model).

- Flat shading: one normal is computed per face, and the illumination model is applied once (e.g. at the barycenter); the same intensity is used everywhere inside the triangle;
- Gouraud shading: one normal is computed per vertex, the illumination model is applied at each vertex, and intensity inside the triangle is computed by interpolation of the intensity at the vertices.

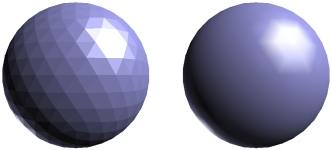


Figure: Left: flat shading; Right: Gouraud shading

Flat shading in OpenGL

In OpenGL, flat shading is obtained by calling *glShadeModel*(*GL_FLAT*) and specifying 1 normal per face:

```
glShadeModel(GL_FLAT);
// ...
glBegin(GL_TRIANGLES);
// the normal vector should be unit
glNormal3f(nx, ny, nz);

glVertex3f(v0x, v0y, v0z);
glVertex3f(v1x, v1y, v1z);
glVertex3f(v2x, v2y, v2z);
glEnd();
```

Gouraud shading in OpenGL

In OpenGL, Gouraud (or smooth) shading is obtained by calling *glShadeModel*(*GL_SMOOTH*) and specifying 1 normal per vertex:

```
glShadeModel(GL_SMOOTH);
// ...
glBegin(GL_TRIANGLES);
glNormal3f(n0x, n0y, n0z);
glVertex3f(v0x, v0y, v0z);

glNormal3f(n1x, n1y, n1z);
glVertex3f(v1x, v1y, v1z);

glNormal3f(n2x, n2y, n2z);
glVertex3f(v2x, v2y, v2z);
glEnd();
```

Rendering of triangle mesh - indexed triangle list

Let us assume that we want to render a triangle mesh stored as an indexed list of vertices and triangles.

```
TriMesh tri_mesh;

void display(void) {
  int n = tri_mesh._number_triangles;
  int i;
  for (i = 0; i < n; ++i) {
    // ...
  }
}
```

Rendering of triangle mesh

```
void display(void) {
  // ...
  for (i = 0; i < n; ++i) {
    int v0, v1, v2;
    Vector3 p0, p1, p2;
    Vector3 n0, n1, n2;

    v0 = tri_mesh._triangles[i]._v0;
    v1 = tri_mesh._triangles[i]._v1;
    v2 = tri_mesh._triangles[i]._v2;

    n0 = tri_mesh._vertex_normals[v0];
    n1 = tri_mesh._vertex_normals[v1];
    n2 = tri_mesh._vertex_normals[v2];
    // ...
 }
}
```

Rendering of triangle mesh

```
void display(void) {
  for (i = 0; i < n; ++i) {
    // ...
    p0 = tri_mesh._vertices[v0];
    p1 = tri_mesh._vertices[v1];
    p2 = tri_mesh._vertices[v2];

    glBegin{GL_TRIANGLES);
    glNormal3f(n0._x, n0._y, n0._z);
    glVertex3f(p0._x, p0._y, p0._z);
    glNormal3f(n1._x, n1._y, n1._z);
    glVertex3f(p1._x, p1._y, p1._z);
    glNormal3f(n2._x, n2._y, n2._z);
    glVertex3f(p2._x, p2._y, p2._z);
    glEnd();
  }
}
```

Rendering of triangle mesh - Half-Edge

```
void display(void) {
  int i;
  for (i = 0; i < num_faces; ++i) {
    Face f = tri_mesh.faces[i];
    Halfedge* he = f._he;
    glBegin(GL_POLYGON);
    do {
      Vector3 n = he->_vertex->_normal;
      Vector3 v = he->_vertex->_coordinates;
      glNormal3f(n._x,n._y,n._z);
      glVertex3f(v._x,v._y,v._z);
      he = he->_next;
    } while (he != f._he);
    glEnd();
  }
}
```

Mesh processing

Over the years, a number of operations have been proposed:

- Smoothing,
- Decimation/Simplification,
- Repairing,
- Deformation,
- Parameterization,
- Remeshing,
- Shape correspondence and matching,
- ...

Smoothing

Smoothing: remove noise from model. Typically needed for meshes acquired via scanning.

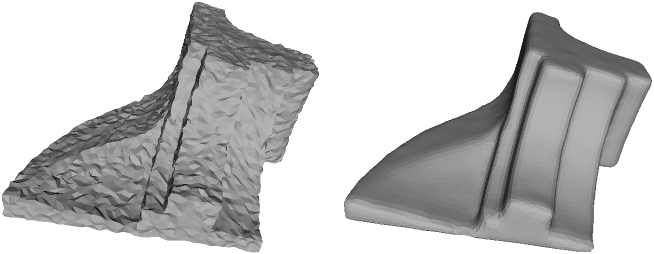


Figure: Left: noisy triangle mesh; Right: smoothed triangle mesh

Laplacian smoothing

Algorithm:

Repeat for a given number of iterations:

  For each vertex v_i:

    Compute $L(v_i)$

    Update $v_i \leftarrow v_i + \lambda L(v_i)$

where $0 < \lambda < 1$.

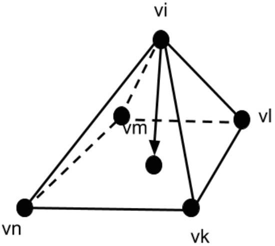
Now we need to provide an expression for this operator L.

Laplacian smoothing: umbrella operator

Use:

$$L(v_i) = \frac{1}{|N_i|}\left(\sum_{v_j \in N_i} v_j\right) - v_i$$

where N_i is the set of vertices adjacent to v_i, and $|N_i|$ is the cardinality (number of elements) of this set.
Problem: shrinkage, modification of flat meshes.

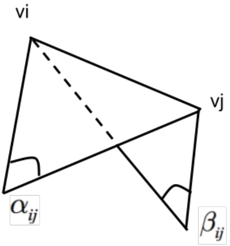


Laplace operator discretization: Cotangent weights

Use $L(v_i) = \Delta_S v_i$ and approximate the Laplace-Beltrami operator (Δ_S):

$$L(v_i) = \frac{1}{\sum_{j \in N_i} w_{ij}} \left(\sum_{j \in N_i} w_{ij} v_j \right) - v_i$$

where $w_{ij} = \frac{1}{2}(\cot \alpha_{ij} + \cot \beta_{ij})$



Decimation

Decimation: decrease the number of vertices / triangles while keeping the appearance of the original object

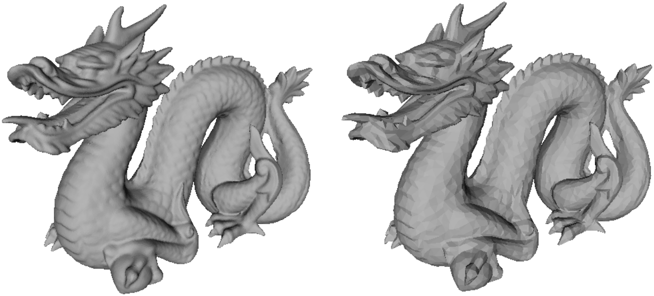


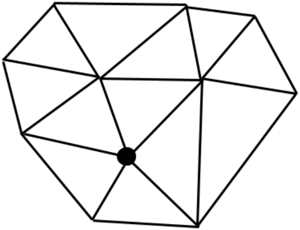
Figure: Left: input triangle mesh for a dragon (Number of vertices: 99892; Number of triangles: 200000); Right: decimated object (Number of vertices: 6247; Number of triangles: 12500)

Decimation: Greedy optimization

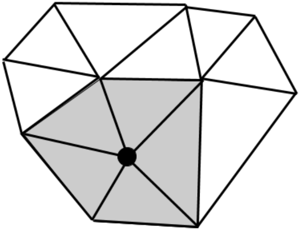
Algorithm:
For each vertex:
  Evaluate vertex quality
  Enqueue in priority queue
Repeat until satisfied criterion:
  Dequeue vertex from priority queue
  Identify neighboring triangles
  Remove triangles
  Triangulate hole
  Update vertex quality

For evaluating the quality of a vertex, one can use the squared distance to the best plane fitted from the vertex and its neighboring vertices.

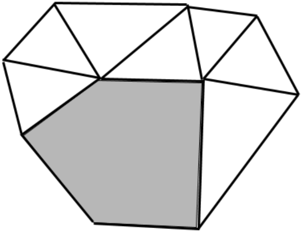
Decimation operator: vertex removal



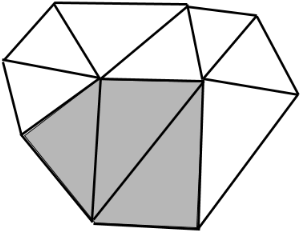
Select a vertex



Identify neighboring triangles



Remove the triangles



Triangulate the hole

Libraries for mesh manipulation

- OpenMesh: `http://www.openmesh.org`. Generic and efficient polygon mesh data-structure (half-edge);
- OpenFlipper: `http://www.openflipper.org`. Programming framework for processing, modeling and rendering of geometric data (based on OpenMesh);
- CGAL: `http://www.cgal.org`. Robust library for computational geometry (half-edge);
- libigl: `http://libigl.github.io/libigl/`. A library for geometry processing (indexed triangle set).

Software for mesh processing

- MeshLab: `http://meshlab.sourceforge.net/`. Implement most of the state of the art algorithms for mesh processing;
- Graphite: `http://alice.loria.fr/index.php?option=com_content&view=article&id=22`;
- Blender: `https://www.blender.org`; 3D creation suite; include functionalities for geometric modeling.

Conclusion

Triangle meshes are a good compromise for representing surface:

- Approximation: $O(h^2)$ error
- Efficient algorithms for rendering (rasterizer)
- Efficient techniques for evaluation and interrogation (position of point on the surface, normal, curvatures, etc)
- Many operations available (smoothing, decimation, etc)