EECS 598-1
Lecture 3: Mesh class, manifold preservation, and out-of-core simplification

Igor Guskov

January 17, 2002

1 Orientable manifold edge-based mesh class

1.1 Directed edge

```
class DEdgeT {
public:
  // standard stuff
  DEdgeT(); // the default constructor produces a null directed edge
  DEdgeT(const DEdgeT& _de);
  DEdgeT& operator=(const DEdgeT& _de);

  DEdgeT(EdgeT* _e, int _dir);
  EdgeT* Edge() const;
  int Dir() const;

  // check if null or not
  operator bool() const;
  bool operator!() const;

  bool operator==(const DEdgeT& de) const;

  // navigation operators
  DEdgeT Flip() const; // equivalent to Sym(Fnext(*this))
  DEdgeT Enext() const; // Enext of a boundary edge is a null dedge
  DEdgeT Enext2() const; // Enext applied twice

  bool Boundary() const; // are we on the outside of the boundary?
```

```
  bool AdjacentToBoundary() const {
    return Boundary() || Flip().Boundary();
  }

  // origin and destination vertices of the directed edge
  VertexT* Org() const;
  VertexT* Dest() const;

  // these set values in the underlying edge structure
  void SetEnext(const DEdgeT& de) const;
  void SetOrg(VertexT* v) const;
};
```

1.2 Vertices

```
class VertexT {
public:
  VertexT();

  DEdgeT DEdge() const;
  DEdgeT& DEdge();
  // returns a d-edge whose origin is this vertex
  // and if this vertex is on the boundary then
  // the returned d-edge is a boundary edge as well
  // which may be important when you go around a vertex
  // collecting its one ring

  bool Boundary() const;

  int Valence() const; // number of edges containing this vertex

  // these are all the data associated with the vertex
  //   position
  const XVecf& Pos() const;
  XVecf& Pos();

  //   normal vector
  const XVecf& Normal() const;
  XVecf& Normal();
```

```
  //   color (r,g,b)
  const XVecf& Color() const;
  XVecf& Color();
};
```

1.3 Edges

```
class EdgeT {
public:
  EdgeT();

  // _dir = 0 or 1
  // edges store two Enext d-edges
  // so that DEdgeT(e, dir).Enext() = e.DEdge(dir)
  DEdgeT DEdge(int _dir) const;
  DEdgeT& DEdge(int _dir);

  // edges store two vertices
  // so that DEdgeT(e, dir).Org() = e.Vertex(dir)
  VertexT* Vertex(int _dir) const;
  VertexT*& Vertex(int _dir);
};
```

1.4 Mesh itself

```
class MeshT {
public:
  typedef std::vector<EdgeT*> EdgeCt;
  typedef std::vector<VertexT*> VertexCt;
public:
  MeshT();
  ~MeshT();
  const EdgeCt& Edges() const;
  const VertexCt& Verts() const;

  void AddVertex(VertexT* v);
  void AddEdge(EdgeT* e);

  // a simple procedure that properly assigns all the d-edges for vertices
  // even on the boundary
```

```
  void AssignVertEdges();

  // helper functions
  static void SetNeighbors(DEdgeT& de, DEdgeT& den);
  static void AssignVertexDEdge(const DEdgeT& des);
};
```

1.5 Topology preservation for surfaces with boundaries

Create a virtual vertex w. Connect all the boundary loops to that vertex with cones. Then the topology preservation condition for edge (a, b) collapse is

$$Lk^w a \cap Lk^w b = Lk^w ab.$$

Note that $Lk^w ab$ for any non-virtual edge consists of two vertices. If ab is an internal edge then those two vertices are non-virtual, if ab is on the boundary then one of the vertices is virtual, and the other one non-virtual.

We can simplify calculations by changing mesh class – representing virtual vertex as a zero pointer. The boundary convention will change then.

2 Progressive meshes

Efficient face-based storage of meshes obtained with half-edge collapses. Precomputed order of removal. Go between levels of detail very fast. Good for games. Index vertices in order of removal backwards – last vertex gets removed first. Sort faces in the inverse order of removal – first faces to get removed come last. Collapse array: stores vertex index transformations during collapses.

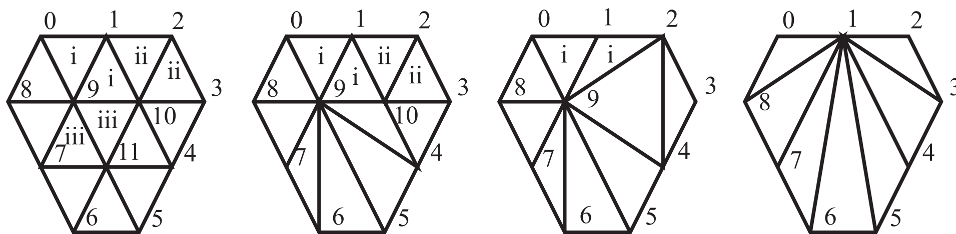

Figure 1: *A triangular mesh example.*

Vertex coordinates are stored in a vertex buffer $p_0, p_1, \ldots, p_1 1$. The last three vertices will be removed. The vertex buffer does not have to change at all. The collapse table is $(9 \to 1, 10 \to 2, 11 \to 9)$.

The triangles are stored in an index buffer, from which the index buffer for a current level is easily generated by substitutions from the collapse table.

```
(0, 8, 9)
  ...
(10, 5, 4)
% the faces above exist on all the levels,
% including the coarsest one
% the following two faces appear on level I
(0, 9, 1)
(1, 9, 10)
% the following two faces appear on level II
(1, 10, 2)
(2, 10, 1)
% the following two faces appear on level III
(9, 7, 11)
(9, 11, 10)
```

Below is an example of substitutions for level III. We first change the substitution table to obtain $(9 \to 1, 10 \to 2, 11 \to 1)$. Then we perform index table generation for level III.

```
(0, 8, 9)  -> (0, 8, 1)
  ...
(10, 5, 4) -> (2, 5, 4)
% the following two faces appear on level I
(0, 9, 1) -> (0, 1, 1) % degenerate!!! everything below is too
(1, 9, 10)
% the following two faces appear on level II
(1, 10, 2)
(2, 10, 1)
% the following two faces appear on level III
(9, 7, 11)
(9, 11, 10)
```

We can stop whenever first degenerate face is hit.

3 Other simplification techniques

Preventing mesh inversion Compute normals of all the faces before and after the edge contraction and if the normals flip by too much then prohibit the contraction.

Complexity Every edge contraction requires on average $\log N_E$ operations to maintain the heap. There are N_E edges to remove hence we get $N \log N_E$ simplification algorithm for local error estimation methods.

3.1 Vertex clustering: Rossignac and Borrel 1993

Introduce grid in 3D, for every cell collapse all the vertices into one. Replace all the vertices in a cell by their average. Discard all the degenerate triangles. Does not care about manifoldness.

3.2 Lindstrom 2000: OoCS

Problem: size of real data. Priority queue: 160*n bytes of RAM. For meshes with billions of points that's a problem.

Lindstrom assumes that output model fits in RAM. Input can be arbitrarily big.

Represent input as a triangle soup T_{in}.

$$
\begin{array}{r}
(x_{11}, y_{11}, z_{11}), (x_{12}, y_{12}, z_{12}), (x_{13}, y_{13}, z_{13}) \\
(x_{21}, y_{21}, z_{21}), (x_{22}, y_{22}, z_{22}), (x_{23}, y_{23}, z_{23}) \\
\ldots
\end{array}
$$

```
Compute bounding box
Make a grid N_x by N_y by N_z
map<(ix, iy, iz) -> (Q, p, index)> qp;
for every triangle t in T_in {
  Compute quadric Q_t
  for every vertex (x_tk,y_tk,z_tk) in t (k=1,2,3) {
    Lookup (x_tk,y_tk,z_tk)->(ix,iy,iz)->qp[ix,iy,iz]
    if needed make a new entry with a new index j_k
    qp[ix, iy, iz].Q += Q_t  // update cell quadric
  }
  if index triple (j_1, j_2, j_3) is non-degenerate
    then insert (j_1, j_2, j_3) into T_out
}
for every active cell in qp {
  find optimal position based on the stored quadric
  store it in the vertex coordinates array at the appropriate index
}
```

4 Mesh fairing details

Very often it is important to predict vertex position from its neighbors. Indeed many curvature and mesh fairing methods can be interpreted that way. Suppose that for every vertex v and its neighbor w we have a weight $\alpha_{vw} > 0$ and such that for a vertex v the sum of all the weights for its neighbors sums up to one: $\sum_{w\in\omega(v)} \alpha_{vw} = 1$. Then we can predict any vertex using $p'(v) = \sum_{w\in\omega(v)} \alpha_{vw} p(w)$. This can serve as a base for an explicit fairing algorithm in which we iterate through all the vertices and reassign their positions using the following expression:

$$p(v) \leftarrow (1-\tau)p(v) + \tau p'(v) = (1-\tau)p(v) + \tau \sum_{w\in\omega(v)} \alpha_{vw} p(w).$$

Note the relation to curvature based fairing because of the following equality:

$$\begin{aligned} (1-\tau)p(v) + \tau \sum_{w\in\omega(v)} \alpha_{vw} p(w) &= \\ p(v) + \tau \sum_{w\in\omega(v)} \alpha_{vw}[p(w) - p(v)] &= p(v) + \tau K[p](v), \end{aligned}$$

where $K[p]$ is a particular normalized curvature operator. Simple umbrella mesh fairing uses $\alpha_{vw} = 1/valence(v)$, while a better set of weights is given in Desbrun's paper.