Lecture 13 - Mesh simplification

Philip Caplan

CSCI 0422 - Geometric Modeling (Spring 2022)
Learning objectives

By the end of this lecture you will be able to:

• implement an algorithm for decimating a surface mesh by contracting vertices so as to minimize a quadric error metric,
• compute the optimal location of a vertex when contracting an edge,
• implement a collapse operator using a half-edge mesh data structure.
Getting started...

Switch Host & Client today!

```
$ git pull
$ make update
$ cmake .
$ make template_class13_simplification
$ cmake .
```

Compiling and running the exercise:

```
$ make class13_simplification
```

Compiling and running the solution (after class):

```
$ make class13_simplification_sol
```
Recall surface reconstruction from last class.

there might be poorly-shaped triangles that we want to remove.

more generally: how can we remove triangles without sacrificing geometry fidelity?
And sometimes, you just want a low-poly look.
Not directly removing triangles → remove edges - but how?
Doing this iteratively (picking the right edges) gives...
Doing this iteratively (picking the right edges) gives...
Doing this iteratively (picking the right edges) gives...
Ingredient #1: order edges to remove by "cost" (impact on geometry).

main idea: model error at a vertex & use this model to determine location of new vertices.

Garland 1997 - *Surface Simplification Using Quadric Error Metrics*

\[
e = v^T Q v, \quad \text{with} \quad v = [v_x, v_y, v_z, 1]
\]

i.e. \( v \) is a 4d vector: 3d vertex represented in homogeneous coordinates. How to pick \( Q \)?
Ingredient #1: computing the data for faces and vertices.

**main idea:** minimize the distance to the planes surrounding a vertex.

**step 1:** calculate face data

\[
p = [n_x, n_y, n_z, -v^T n_\Delta]
\]

\[n_\Delta = [n_x, n_y, n_z]: \text{triangle normal}\]

\[v: \text{any point on the face.}\]

**step 2:** calculate vertex data

\[
Q = \sum_{\Delta \in T(v)} K_\Delta
\]

\[K_\Delta = pp^T (\text{outer product})\]

\[T(v): \text{triangles around vertex } v \text{ (one-ring).}\]
Ingredient #1: calculating optimal vertex coordinates and error of an edge.

**step 3:** calculate edge data

\[ \bar{Q} = Q_1 + Q_2 \]

Solve:

\[
\begin{bmatrix}
\bar{q}_{11} & \bar{q}_{12} & \bar{q}_{13} & \bar{q}_{14} \\
\bar{q}_{21} & \bar{q}_{22} & \bar{q}_{23} & \bar{q}_{24} \\
\bar{q}_{31} & \bar{q}_{32} & \bar{q}_{33} & \bar{q}_{34} \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\bar{v}
\end{bmatrix}
\begin{bmatrix}
0 \\
0 \\
0 \\
1
\end{bmatrix}
\]

\( \bar{v} \) is the optimal point of the contracted edge. The "cost" of the edge is then

\[ e = \bar{v}^T \bar{Q} \bar{v}. \]
Ingredient #1: saving the data in our half-edge data structures.

**note:** not storing 4x4 matrix $K$ on every face (just $p$, so we will need to compute $K = pp^T$).

```c
struct HalfFace {
    HalfEdge* edge; // pointer to one edge which has the face on the left
    int index = -1; // index of the face (i.e. element number)
    vec4d p; // [nx, ny, nz, -dot(x,n)] where x is a point on the face
              // n is the face normal (used in simplification algorithm)
};

struct HalfVertex {
    HalfVertex( int dim, const double* x ) {
        flux_assert( dim == 2 || dim == 3 );
        for ( int d = 0; d < dim; d++ )
            point[d] = x[d];
    }
    HalfEdge* edge; // pointer to one edge emanating from this vertex
    vec3d point; // the coordinates of this vertex (always 3d!)
    int index = -1; // index of this vertex (i.e. vertex number)
    mat44d Q; // error quadric (used in simplification algorithm)
};

struct HalfEdge {
    HalfVertex* vertex; // pointer to the 'origin' vertex of this edge
    HalfEdge* twin; // pointer to the 'opposite' edge (i.e. parallel but in opposite direction)
    HalfEdge* next; // pointer to the next edge around the face in CCW order
    HalfEdge* prev; // pointer to the previous edge around the face in CCW order
    HalfFace* face; // pointer to the left face of this oriented edge

    // data used in simplification algorithm
    vec4d vbar; // optimal point (in homogeneous coordinates) where the endpoints should be collapsed
    mat44d Qbar; // sum of endpoint Q's, i.e. Qbar = Q1 + Q2
    double cost; // the cost (error) of this edge: vbar^T * Qbar * vbar
};
```
Ingredient #1: ordering the edges by cost with a priority queue.

```cpp
/**
 * \brief Provides a comparison operator to order edges in a set/multiset
 */
struct CompareCost {
    bool operator() ( const HalfEdge* e1 , const HalfEdge* e2) const {
        flux_assert( e1 != nullptr && e2 != nullptr);
        return e1->cost < e2->cost;
    }
};

// defines a set of HalfEdge pointers ordered by their cost
// (allows for multiple edges with the same cost)
//
// examples:
//
// CollapsePriorityQueue edges; // declare a priority queue of edges to collapse
//
// edges.insert( e ); // e is some HalfEdge pointer
//
// HalfEdge* top = *edges.begin(); // get the HalfEdge* with the highest priority (lowest cost)
//
// edges.erase( top ); // remove a HalfEdge* from the priority queue
//
typedef std::multiset<HalfEdge*,CompareCost> CollapsePriorityQueue;
```
Ingredient #2: checking if the collapse is valid.

before a collapse is performed, check:

$$|\mathcal{R}_1 \cap \mathcal{R}_2| = 2$$

where $\mathcal{R}_1$ and $\mathcal{R}_2$ are the vertex one-rings of the edge endpoint vertices.

→ you can also check if the face normals flip direction before/after the collapse.
Ingredient #2: update the half-edge entities to reflect the edge removal.

You should use the `remove` functions defined for the `HalfEdgeMesh` class, which can be used to remove a `HalfVertex*`, `HalfFace*` or `HalfEdge*` from our `HalfEdgeMesh` container.

And update the priority queue with the new cost of affected edges!
Summary of the simplification algorithm.

1. pre-process:
   (a) build up the half-edge representation of the mesh,
   (b) loop through every face and calculate the vector \( \mathbf{p} \) and the matrix \( \mathbf{K}_\Delta \) for this face,
   (c) loop through the vertices and compute \( \mathbf{Q} \),
   (d) loop through the edges and compute \( \bar{\mathbf{Q}}, \bar{\mathbf{v}} \) as well as the cost \( \bar{\mathbf{v}}^T \bar{\mathbf{Q}} \bar{\mathbf{v}} \).

2. insert every edge into a priority queue using the cost of the edge to order the edges.

3. while the number of faces is greater than the target \( \text{nf\_target} \):
   (a) retrieve the edge with the highest prority (lowest cost),
   (b) collapse the edge, setting the coordinates of the receiving vertex to \( \bar{\mathbf{v}} \),
   (c) update the \( \mathbf{Q} \) value of the receiving vertex (now equal to \( \bar{\mathbf{Q}} \) of the collapsed edge). Also update the \( \bar{\mathbf{v}}, \bar{\mathbf{Q}} \) and cost values for every edge affected by the collapse (i.e. in the new one-ring of the receiving vertex, including twins).
   (d) Remove and re-insert any affected edges into your priority queue with the new cost values.

4. for visualization: convert the half-edge representation of the mesh to a connectivity-based representation (see the extract function defined for HalfEdgeMesh).
Exercise: calculating the **HalfFace**, **HalfVertex** and **HalfEdge** data.

We will practice with steps 1 & 2 today.
One more example!
Your TODO list . . .

- add tests for `HalfEdgeMesh::remove` functions (in `halfedges.h,cpp`), which you can use for Project 3,
- add tests for `Sphere<Triangle>` class (in `sphere.h,cpp`),
- start Project 3.