Homework #4: Mesh simplification and expressive rendering [95 points]
Due Date: Wednesday, 15 March 2017 in class — no late days will be available for this homework

This is the last homework of CS 348a, consisting of one programming problem in three parts. It is a more open-ended assignment compared to the earlier ones, allowing you to combine the modeling and algorithmic tools you have learned about in a variety of ways. The parts are designed to be worked on in parallel, but do get started as early as possible since each part is challenging!

Problem 1. [95 points]

Geometric Features for Non-photorealistiic Rendering

In this project you will implement a pipeline for rendering interesting features on a 3D mesh. You first will decimate the mesh to a smaller number of triangles, so that fewer triangles need to be processed in the second part of the project. Then, you will implement part of a technique for finding "suggestive contours" highlighting key geometric features of the object being modeled. Finally, you will extend the system to have additional capabilities of your choosing.

Mesh Decimation

In this part of the final project, you will implement surface decimation using the method proposed in Garland and Heckbert's "Surface Simplification Using Quadric Error Metrics" (SIGGRAPH 1997). The algorithm proceeds as follows:

1. Compute and save an initial quadric approximating the surface near each vertex.
2. Compute collapse priorities for each of the vertices by summing quadrics.
3. Merge adjacent pairs of vertices by collapsing halfedges when such a collapse is valid and preferable according to the priority value.

You are provided with two starter files, `src/decimation.cpp` and `include/decimation.h`. The `simplify` function performing the decimation takes a mesh in halfedge format and the desired ratio between the initial and final vertex count (a float value ≤ 1).

Decimation algorithm: The decimation algorithm uses a quadric error measure to collapse halfedges. Given an edge e joining vertices s and t, a halfedge collapse pulls s into t, removing s, e and the two triangles adjacent to e. This operation is illustrated in Figure 1.

The decimation algorithm operates on the following principles:

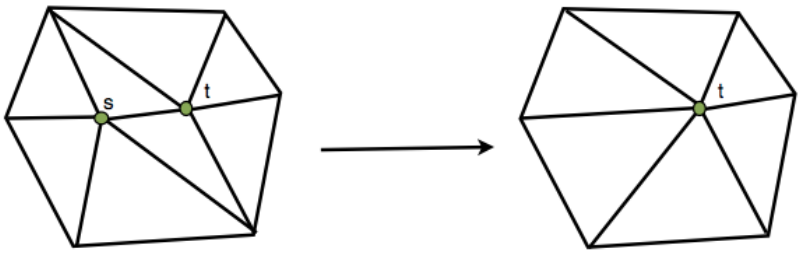


Figure 1: Halfedge collapse

1. A pair of vertices may be collapsed to a single vertex *only* if they are connected by an edge and they do not result in a triangle flip. This condition is checked by calling the `is_collapse_legal()` function.

2. A halfedge $h = s \to t$ will be assigned a priority $priority(h)$ based on the quadric error metric. We assign a priority to every vertex s such that the priority of s is equal to that of the halfedge leaving from s having minimum priority:

$$priority(x) = \min_{h=s\to t} priority(h)$$

   Label the opposite vertex of the minimal halfedge as t, that is, $t = target(s)$.

3. We will store a priority queue of vertices instead of halfedges so that each vertex has a priority based on the quality of the quadric approximation and a target t.

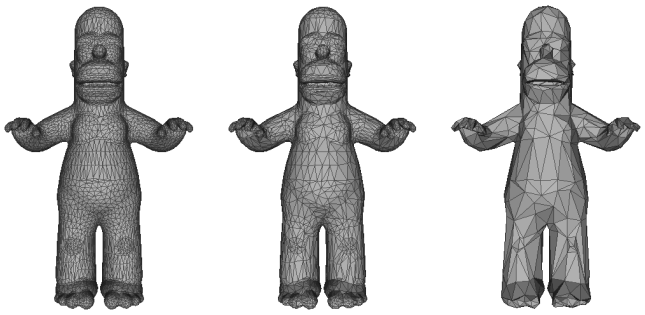
4. In the main loop of the algorithm, we release the vertex s having the minimum priority from the queue. Then the halfedge, $s \to target(s)$ is collapsed so that s coincides with $t = target(s)$. The properties of t and other affected vertices are updated in the queue.

Computing quadrics: Each triangular face is associated with a quadric measuring squared distance to the plane defined by its vertices. Then, the quadric associated with a vertex p is the sum of the quadrics of its adjacent triangles. In particular, these quadrics are given by a sum of squared distances:

$$\sum_i dist(q_i, p)^2 = \sum_i p^\top Q_i p = p^\top \left(\sum_i Q_i \right) p \equiv p^\top Q p$$

where $p = (x, y, z, 1)^\top$ is the homogeneous coordinates of the vertex p, $q_i = (a_i, b_i, c_i, d_i)^\top$ the unit length vector containing the coefficients of q_i's plane equation $a_i x + b_i y + c_i z + d_i = 0$, and the matrix Q_i is given by $Q_i = q_i q_i^\top$. Use the quadric structure is located in `decimation.h`.

(a) (10 points) Complete the `initialize()` function in `decimation.cpp` so that it calculates vertex quadrics from the quadrics of the incident triangles.



(a) (b) (c)

Figure 2: (a) An input mesh, (b) the mesh simplified to 50% of its vertices, and (c) the mesh simplified to 10% of its vertices.

(b) (10 points) Compute the priority of a halfedge using the sum of its two end-vertex quadrics and return it in the function `compute_priority()`.

(c) (15 points) Implement the main loop of the `decimate()` function. In every iteration there are three steps:

1. Take the vertex with the lowest priority from the queue by calling the `top()` and `pop()` function.
2. The vertex priority values in the queue can be outdated. Check whether the halfedge is valid using `is_vertex_priority_valid()` function.
3. If the corresponding collapse is legal (using the `is_collapse_valid()` function), then collapse the halfedge corresponding to this vertex.
4. Update the properties of the vertices in the queue whose adjacent triangles have been changed using the `enqueue_vertex()` function.

It may be useful to check your output using a mesh viewer such as MeshLab, which can be downloaded for free from `http://meshlab.sourceforge.net`. Your final result should look like Figure 2 for the simplification to 50% and 10% of the vertex count of the `homer.off` mesh.

Rendering Suggestive Contours

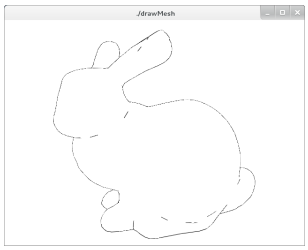
Now that you have simplified the mesh to a reasonable number of triangles, we will use our principal curvatures and directions from the last assignment to implement a rudimentary method for finding curve features on meshes based on "Suggestive Contours for Conveying Shape" by DeCarlo et al. (SIGGRAPH 2003).

Your first task is simple: Import your code from the last homework for computing principal curvatures and directions. This code should be stable to make sure that the remaining parts of the program work as advertised. If you have switched project groups since the last assignment, you can use any group member's code.

"Suggestive Contours" is an example of a method for *nonphotorealistic rendering,* a class of techniques for rendering meshes by emphasizing interesting features rather than applying the physics of light. We will be implementing a part of the paper for finding two types of contours on the surface that highlight what the surface looks like from a given camera angle. Our mesh simplification from the first part of the assignment will allow us to preview these curves in real time.

We will first identify silhouette edges on our input meshes. These edges denote the boundaries between visible and invisible parts of a mesh.

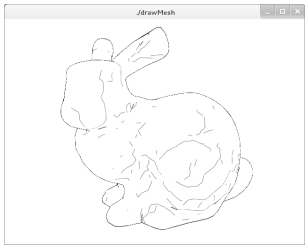
(d) (5 points) Complete the `isSilhouette` function in `src/mesh_features.cpp`. The definition of a silhouette edge is one for which one neighboring triangle points toward the camera and the other points away. Also, complete `isSharpEdge` to detect edges for which the dot product of the normals of the adjacent faces is less than $1/2$.



In the starter code, you can make the mesh invisible by pressing `s`; note that the mesh still is rendered in white to remove hidden lines. You should still see an silhouette curves as shown on the right. These curves obviously are important for conveying the shape of a 3D object, but are far from enough for communicating important geometric detail.

Suggestive contours make use of the mesh curvature for finding additional expressive features. In particular, take a point p on a surface. Take v to be the vector pointing from p to the viewer, and let w be its projection onto the tangent plane S at p normalized to unit length. From Assignment 3, we know how to compute κ_w (denoted κ_r in the DeCarlo paper), the curvature of the surface in the w direction.

(e) (5 points) Complete the `computeViewCurvature` method in `src/curvature.cpp`. To do so, compute κ_w at each vertex of the mesh, and store the result in the mesh property `viewCurvature`.



A suggestive contour is a zero crossing of κ_w (in our notation). Since we have one value of κ_w per vertex of the mesh, the zero crossings will happen along the edges – that is, if you linearly interpolate κ_w along an edge using the values at its vertices, if there is a sign change then somewhere in between the view curvature will be zero. Triangles for which κ_w has different signs on different vertices will contain such edges; the suggestive contour can be thought of as a line segment connecting the $\kappa_w = 0$ points on the edges.

As explained in the paper, you should only show those contours for which the derivative of κ_w in the w direction — notated $D_w\kappa_w$ — is positive. We provide code for computing the gradient of κ_w on the mesh as one vector per face and adding it to the property

`viewCurvatureDerivative`. Take the dot product of this gradient and w to obtain the directional derivative $D_w\kappa_w$.

(f) (20 points) Complete the `renderSuggestiveContours` method in `src/main.cpp`. To do so, determine which faces will have suggestive contours running across them, and then draw segments across these faces. As proposed in the paper, you should eliminate curves where $D_w\kappa_w$ is positive but small and ones where the angle between the surface normal and the view vector is small.

Additional Features

The final part of the project will be more open-ended. We would like you to add an additional *geometric* feature to the rendering system to improve or extend it:

(g) (30 points) Add an additional feature.

You are welcome to contact the course staff with ideas or to help figure out what will be feasible or interesting for this part of the project.

In case you're low on ideas, we have provided you with starter code for one potential extension. In particular, our viewer shows feature curves as chains of line segments, but we'd like to see smooth curves. So, in `src/image_generation.cpp` we have provided code to help generate 2D images with small numbers of smooth curves instead of large numbers of 3D line segments. In particular, we provide you with a method `toImagePlane` to take 3D points and convert them to 2D image locations (the third coordinate is the depth buffer value) and an additional method `isVisible` using the depth buffer to check if a 3D point is visible.[1]

In this additional feature, you should generate a `.svg` file in the `writeImage` function containing a curve-based drawing of the mesh. Project feature line segments from the second part of the assignment to curves on the image plane and remove segments that are invisible. Then, propose a method to replace the long chains of line segments with approximations using polynomial curves. The `svg` format has a `path` element that will be useful here. Your final output should be viewable in Inkscape, an open source image editor you can download. Note this is a difficult problem, so your final writeup should include a discussion when your technique works and when it fails.

Other ideas include:

- Eliminate spurious feature curves that are too short or uninteresting. Smooth out the remaining curves before rendering them in OpenGL.
- Trace principal curves (curves whose tangents are principal directions T_1 or T_2) along the surface to obtain additional feature curves. Propose a way for choosing a set of "interesting" principal curves, since two exist starting at any point.

[1] This method is highly inaccurate and should be replaced for more accurate visibility detection! You may choose to implement a more complex geometric visibility technique that doesn't read the OpenGL depth buffer, or you can add tolerances to render slightly less-than-visible points.

- Use the principal curvatures or other geometric features to shade or texture the mesh in a way that makes it easier to see key features.
- Implement a more stable principal curvature computation method and explore whether it improves the suggestive contours.
- Animate the mesh as it rotates, and identify stable suggestive contours during this motion so that the animation looks stable.

You are provided with an OpenMesh tutorial with this homework. You can use it to better understand OpenGL syntax before beginning to work on the assignment.

What you will be given

You will be provided with meshes in `.off` format, with the assumption that each mesh is fairly smooth and non-degenerate. Some sample meshes have been provided in the `models/` directory of the starter code.

What to hand in

Remember that in programming assignments you are allowed to work in teams of up to three students, and that each team needs to hand in only a single write up. On Friday morning, 10 March 2017, you will receive the actual test data that will be used for grading. On or before the due date, you must submit the following:

- Final version of your source code
- Images demonstrating the capabilities of your program
- Paper write-up: Give an overview of your program and explain how your algorithms work.

We will set up time periods on Friday, 17 March 2017, during which you will have to demo your program to the instructor and the CA.