Wavelets and Multiresolution Modeling Project 4: Mesh Simplification due on Monday 3/19/2001

Martin Bertram, bertram@cs.utah.edu

1 Digital Surface Models

The digital revolution has recently conquered the market for audio/video products and equipment. In the same way as CDs and movies can be shipped around on the internet and displayed on of-the-shelf PSs, three-dimensional digital surface models become available for use in computer graphics and animation applications. An impressive example is the Digital Michelangelo project at Stanford University, see http://graphics.stanford.edu/projects/mich/ .

Triangulated surface models can be constructed from densely sampled objects providing an ideal representation for surfaces of arbitrary topology, *i.e.*, surfaces that may have handles or tunnels. The tremendous amounts of triangles, however, need to be reduced and compressed to make transmission and archiving feasible. Therefore, we need to construct algorithms for irregular triangle meshes that "mimic" the behavior of wavelet transforms. A fitting operation (or low-pass filter), for example, is implemented as a mesh-simplification procedure. Figures 1 and 2 show simplified triangle models obtained by collapsing edges.

1.1 Representing Triangle Meshes

A simple data structure for triangle meshes is just an array of vertices and an array of triangles. Every triangle has pointers to its three adjacent triangles and to its three vertices. Every vertex has a list of pointers to its incident triangles and it contains its own coordinates, see Figure 3. The triangle edges are not explicitly represented. We can traverse all edges by traversing the array of triangles and by considering for every triangle only the edges that are either boundary edges or incident to another triangle with lower index (than the current triangle).

You find four examples for triangulated surfaces on the CS6941 web page http://www.cs.utah.edu/classes/cs6941/. These are the Crater-Lake terrain model (19380 triangles), a re-meshed version of the Stanford bunny (10000

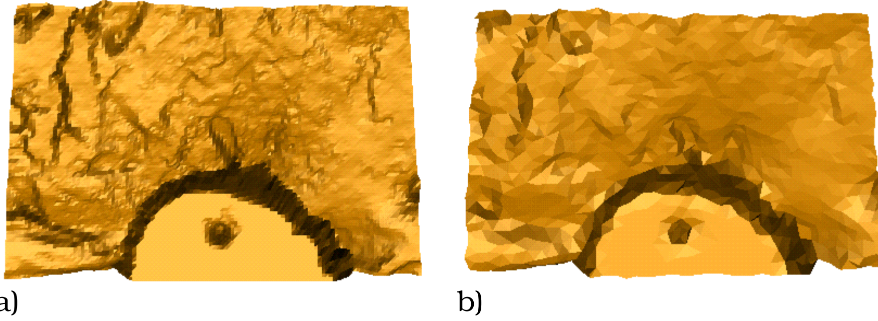

Figure 1: Crater-Lake terrain model, courtesy of USGS. This figure shows the original surface composed of 19380 triangles a) and a simplified surface model composed of 2000 triangles b).

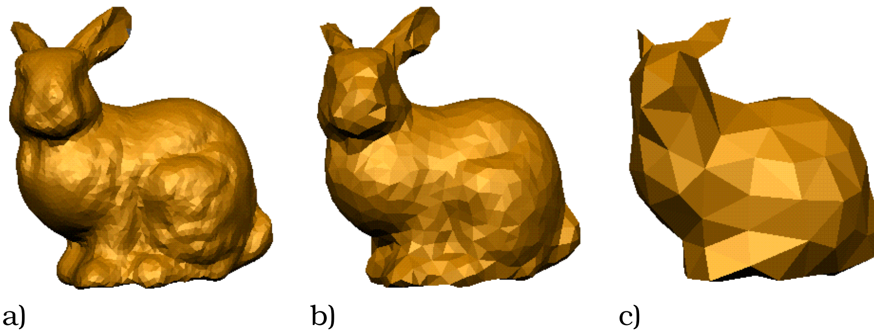

Figure 2: Stanford bunny, courtesy of the Stanford University Computer Graphics Laboratory. Simplified surfaces are obtained from a re-meshed version, composed of a) 10000, b) 2000, and c) 200 triangles.

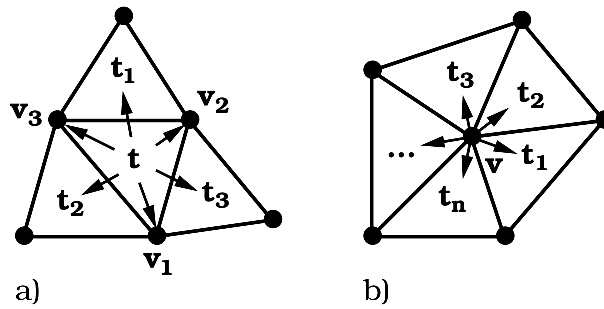

Figure 3: A simple data structure for triangulations. a) Every triangle t is linked to its three vertices and to its adjacent triangles. b) Every vertex has an associated list of its incident triangles (which is not sorted, in general).

triangles), an object with three tunnels/handles (genus-three object, 5000 triangles), and a very simple surface for debugging purposes (12 triangles). Every file contains the number of vertices, the number of triangles, three floating-point coordinates for every vertex, and six indices for every triangle. These are three vertex indices $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ (starting with index 0 in the array of vertices) and three indices to adjacent triangles $\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3$ (starting with index 0, boundary edges have index -1). The orientation for all triangles is consistent, such that triangle normals computed as

$$\begin{aligned} \mathbf{n}' &= (\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1), \\ \mathbf{n} &= \frac{\mathbf{n}'}{||\mathbf{n}'||} \end{aligned} \qquad (1.1)$$

will all point to the same side of a surface. Additionally, the triangle indices are in such order that the adjacent triangle $\mathbf{t}_i$ of $\mathbf{t}$ is located at the opposite side of $\mathbf{v}_i$, as shown in Figure 3.

Task1: Implement the following functions:

- **Reading a triangle mesh from file and generating the internal data structure (you do not have to use the data structure suggested here).**

- **Rendering the triangulation by computing a normal for every triangle and using flat shading. It should be possible to rotate the surface interactively around one or two axes.**

- **Verifying the correctness of the data structure and reporting possible errors. If you use the data structure suggested here, this procedure should verify for every triangle t, whether its vertices $\mathbf{v}_i$ point back to t, whether its adjacent triangles $\mathbf{t}_i$ point back to t, and whether the order of indices is correct, such that vertex $\mathbf{v}_i$ does not belong to triangle $\mathbf{t}_i$ (since it has to be on the opposite side). For every vertex, it should be verified, whether all its incident triangles point back to it. You may need this function later for debugging purposes.**

1.2 Collapsing Edges

The simplest way to reduce the number of triangles is by recursively collapsing edges. As illustrated in Figure 4, collapsing the edge $\mathbf{v}_i\mathbf{v}_j$ will replace these two vertices by their midpoint $\mathbf{v}$, remove the triangles $\mathbf{t}_a$ and $\mathbf{t}_b$, reshape all other triangles shown, and decrement the valences (the numbers of incident edges) of vertices $\mathbf{v}_a$ and $\mathbf{v}_b$. The valence of the new vertex $\mathbf{v}$ is the sum of the valences of $\mathbf{v}_i$ and $\mathbf{v}_j$ minus four (minus two when collapsing a boundary edge). Using the data structure defined earlier, an edge collapse requires the following changes:

- Re-connect the triangles adjacent to $\mathbf{t}_a$ (except for $\mathbf{t}_b$), thus destroying their references to $\mathbf{t}_a$.

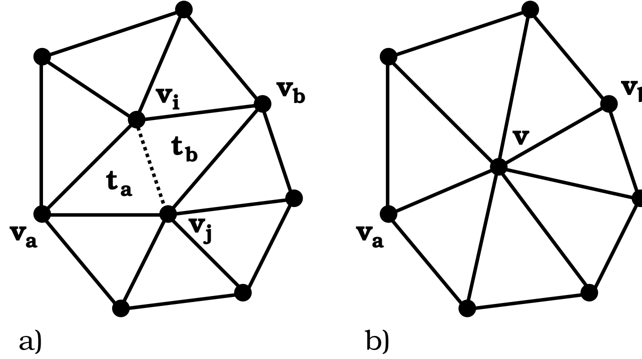

Figure 4: Collapsing the edge $\mathbf{v}_i\mathbf{v}_j$. This figure shows a local triangulation before an edge collapse a) and after the collapse b).

- Remove the references to triangle $\mathbf{t}_a$ from the vertices $\mathbf{v}_i$, $\mathbf{v}_j$, and $\mathbf{v}_a$ and decrement their valences.
- If the edge to be collapsed is not a boundary edge, then repeat the above steps with $\mathbf{t}_b$ and $\mathbf{v}_b$, instead of $\mathbf{t}_a$ and $\mathbf{v}_a$. Otherwise, $\mathbf{t}_b$ and $\mathbf{v}_b$ do not exist.
- Replace the vertices $\mathbf{v}_i$ and $\mathbf{v}_j$ by their centroid and merge their lists of adjacent triangles. Re-direct all pointers from incident triangles to the new vertex.
- Remove the triangles $\mathbf{t}_a$ and $\mathbf{t}_b$ (if exists).

We note that removing vertices/triangles from an array is a complex operation. Either, we mark these vertices/triangles as invalid and check for validity every time we traverse the corresponding array, or we overwrite the removed element by the last one in the array and decrement its size. In the latter case, all references to the element that was moved need to be updated. Our suggested data structure allows us to find these references locally, without any global traversal of the arrays. When efficiently processing large data sets, it becomes crucial that collapsing an edge is a local operation.

The next arising question is, which edge we choose for collapsing. One can estimate an error residual that is introduced to a mesh when collapsing a certain edge and then choose the edge that will introduce the smallest residual. The computation of such residuals is typically based on the similarity of normals of the merged vertices or of the removed triangles, or on some other local properties of the current mesh (*memoryless* simplification). More sophisticated approaches compute a *quadratic error metric* measuring the residual with respect to the original mesh, which is more expensive and does not always provide better results, however. For our implementation, we simply choose the shortest edge that can be legally collapsed.

An edge collapse is called *legal*, if it neither changes mesh topology, nor results in an invalid connectivity. This could happen, for example, if vertex $\mathbf{v}_a$ in Figure 4 has valence three, such that its adjacent vertices, except for $\mathbf{v}_i$ and $\mathbf{v}_j$, are identical. In this case, the two triangles adjacent to $\mathbf{t}_a$ (except for $\mathbf{t}_b$) would be identical after the collapse, resulting in a degenerate surface. To avoid any kind of degenerate triangulation after a collapse, all vertices shown in Figure 4a) must be disjunct. If this is the case, then an edge collapse is legal. We implement this criterion by constructing two lists containing all vertices adjacent to $\mathbf{v}_i$ and $\mathbf{v}_j$, respectively. Then, we merge these lists and remove one instance of the vertices $\mathbf{v}_i$, $\mathbf{v}_j$, $\mathbf{v}_a$, and $\mathbf{v}_b$ (if exists). The remaining vertices in the list must all be different from each other.

Task2: Implement a function determining whether an edge (defined by a triangle with an edge index) can be legally collapsed. Determine the shortest edge in a triangulation that can be legally collapsed. Implement the edge collapse as a local operation, *i.e.*, without traversing the whole data set, and verify correctness of the resulting mesh structure.

2 Project Specification

Implement a tool that reads a triangulated surface from file, collapses the n shortest edges that can be legally collapsed, and then displays the resulting surface from different views using flat shading. For large data sets it may be too inefficient to look for the shortest edge in the whole data set. To accelerate the algorithm, you may determine the shortest legal edge from a randomly selected subset of edges. Some other strategies, like pre-sorting the edges and updating the modified ones after every collapse, may work as well.

ATTENTION. You must develop your own code. You can use code from the preceding projects. Sharing code with fellow students is not permitted. Do not erase your code when the project is done.