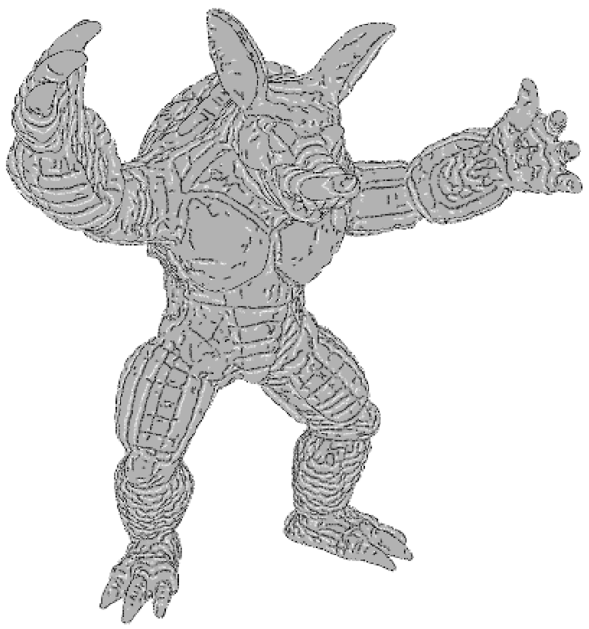
CS 348A (Winter 2013) Geometric Modeling and Processing Project: Mesh Simplification and Expressive Rendering

Potcharapol Suteparuk
neungs1@stanford.edu

Emmanuel Tsukerman
emantsuk@stanford.edu



1 Introduction

In this project we will implement a pipeline for rendering interesting features on a 3D mesh. It consists of three parts. We 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, we will implement part of a technique for finding suggestive contours highlighting key geometric features of the object being modeled. Finally, we implement additional features that extend the capabilities of the system.

2 Algorithm and Implementation

As mentioned above, the implementation consists of three parts.

2.1 Mesh Decimation

We follow the method described in Garland and Heckbert's "Surface Simplification Using Quadric Error Metrics" (SIGGRAPH 1997).

(a) Implement `initDecimation()`. Basically, for each vertex v, we iterate over its incident triangles. For each triangle, we find the plane equation $ax + by + cz + d = 0$. To do this, we simply let (a, b, c) to be the normal of the face, and $d = -n^T p$ where p is any point on the face (we choose v). Use the quadric class to evaluate to face quadric, and sum them up to get the vertex quadric.

(b) For each halfedge, we sum the two quadrics of the two end vertices. We then evaluate this quadric on the `to_vertex` of this halfedge. This is because we wish to collapse the `from_vertex` to the `to_vertex`, so it should make sense to evaluate the total quadric at the resulting point.

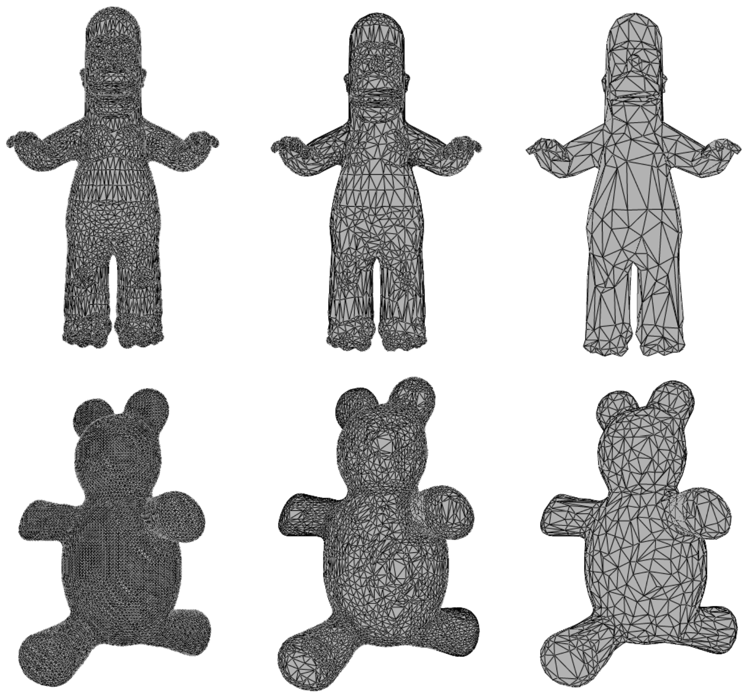


Figure 1: (left) Input meshes. (middle) 50% simplified. (right) 10% simplified.

(c) Implement `decimate()`. Extract the first vertex from the priority queue. Find the least priority halfedge associated with this vertex using `target()` function. Then collapse from the `from_vertex` to the `to_vertex`, checking first that `is_collapse_legal()`. We store all the neighborhood vertices of the `from_vertex`, i.e. the vertex we extracted from the queue, within a vector. For each of these, we need to enqueue them again to update their priorities since their incident triagles have changed. In addition, we all need to add the quadric of the collapsed vetex into the quadric of the `to_vertex`.

2.2 Rendering Suggestive Contours

We follow the method proposed by De Carlo et al.'s "Suggestive Contours for Conveying Shape" (SIGGRAPH 2003). The simplified version of the meshes will make the computation less time-consuming. Unless states otherwise, all the results from now on will be presented with the 50% simplified version of the meshes. We also use our computation of the curvature from the previous assignment as well.

(d) We want to find the edges where the two incident faces point toward different directions (wrt to the camera). We find the dot products of each face normal and the view vector of each face (a vector from the face to the camera). If the product of these two are negative, we return the edge as a silhouette edge. Note that we choose the view vectors from the centroids (barycenter) of each face. It might be a bit slower than simply using a vertex, but the cost should not affect the whole system much. Of course, if each vertex has a weight associated with it, then we can take those weights into account to find the best representative point of a face. (Although since the faces are quite small, there should not be much differences).

(e) We have calculated the principal curvatures κ_1 and κ_2, and the corresponding principal directions T_1 and T_2 using Taubin's algorithm from the previous assignment. We now want to find the curvature κ_w corresponding to the direction w, a unit projection of the view vector onto the tangent plane at each

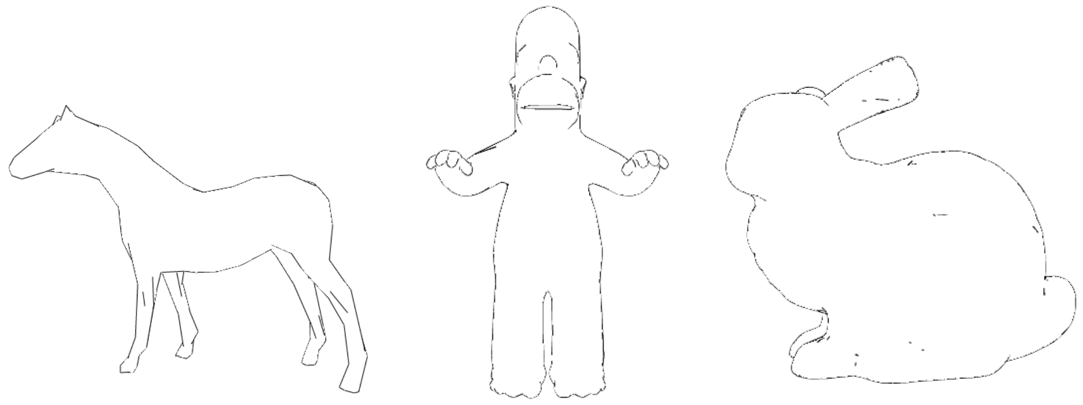


Figure 2: Silhoutte and sharp edges for some of the meshes.

vertex. For each unit tangent vector T_θ, we have the following relations

$$T_\theta = T_1 \cos\theta + T_2 \sin\theta$$
$$\kappa_\theta = \kappa_1 \cos^2\theta + \kappa_2 \sin^2\theta$$

Since T_1 and T_2 are unit orthogonal, taking the dot products $T_\theta \cdot T_1$ and $T_\theta \cdot T_2$ yield $\cos\theta$ and $\sin\theta$, respectively. Hence we have $\kappa_\theta = \kappa_1(T_\theta \cdot T_1)^2 + \kappa_2(T_\theta \cdot T_2)^2$. So for each vertex, we calculate the view vector v, project it onto the tangent plane by doing $v - (n \cdot v)n$, and w is this vector normalized. After getting w, we can find $\kappa_w = \kappa_1(w \cdot T_1)^2 + \kappa_2(w \cdot T_2)^2$ easily.

We are also given the directional derivative associated with each face Dw to use in part (f).

(f) We can iterate through each triangular face and check the signs of κ_w on each of its vertices. If they are not all the same, there must be exactly two pairs that differ in signs. For each pair, we can linearly interpolate along the connecting edge to find the point which has $\kappa_w = 0$, i.e. find t such that $(1-t)\kappa_{wA} + t\kappa_{wB} = 0$ and use that t to find the point $(1-t)A + tB$. Having two points, one on each edge, we can then draw a line that will be a part of the suggestive contours.

In order to add more features in the next part, however, we will store all the information as per-face property in OpenMesh called `contour`. This will store a struct `ContourInfo` which will then store five things, the two edges having discrepancy between the signs of κ_w, the two end points of the contour, and a boolean flagging whether the corresponding face has a suggestive contour on it or not.

As defined in the de Carlo paper, the suggestive contours contain all points which have $\kappa_w = 0$ and $Dw\kappa_w > 0$ when $n \cdot v > 0$. So when we get the two zero points, we need to check that the latter conditions both are satisfied as well. We tweak it a little bit by allowing one point to violate the $Dw\kappa_w > 0$ condition. Instead, since now the two points have different $Dw\kappa_w$ signs, we can find a point on the supposed-to-be-drawn contour where $Dw\kappa_w = 0$. We then can draw the partial contour from that point to the one with a positive sign.

As proposed in the paper as well, since this is a numerical computation, we need to allow a little threshold for the test of zeroness. So our conditions become $Dw\kappa_w > thresh_1$ and $n \cdot v/\|v\| > thresh_2$. We design the program so that the users can play with different thresholds by simply using '+' and '-' keys to increase or decrease the values of angle threshold, and using ARROW_UP and ARROW_DOWN to manipulate the directional derivative threshold. Note that we set the angle threshold around 6^o to 20^o, the other threshold is of magnitude 50 - 600. The results vary from mesh to mesh depending on these thresholds.

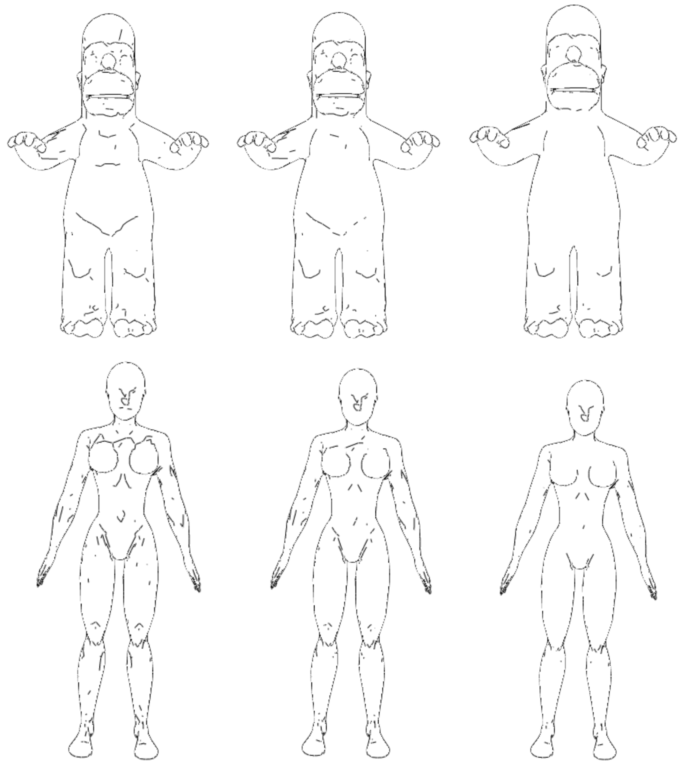


Figure 3: Suggestive contours experimenting on different thresholds.

3 Additional Features

In this part, we extend the program to show more interesting features.

3.1 Filtering spurious contours

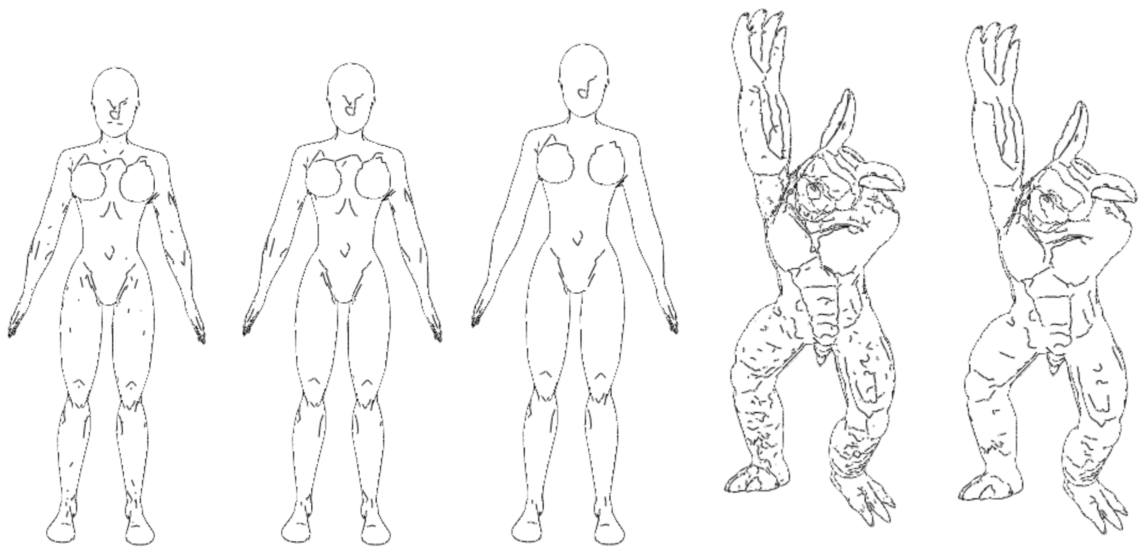


Figure 4: Filtering short contour chains helps clean up and leave only major contours.

We claim that spurious means that the contour "chain" is too short to describe any features. As such, we aim to filter out the chain of contour lines that are shorter than a certain threshold. (That is the contour is from only a few connecting triangles.) This is why we store the contour information in `ContourInfo` so that we can extract those information to determine whether a certain face has a contour or not, and determine what points and which edges connect the chain together. Again, users can play with the threshold from length 1-5. They can also turn filtering on-off by pressing 'f'.

The challenge here is you have to check which side are we iterating on the chain. Also we don't want to examine the same face over and over, as well as the same chain but in the opposite direction. We create another per-face property called `chainFlag` to keep track of this.

3.2 Suggestive Highlights

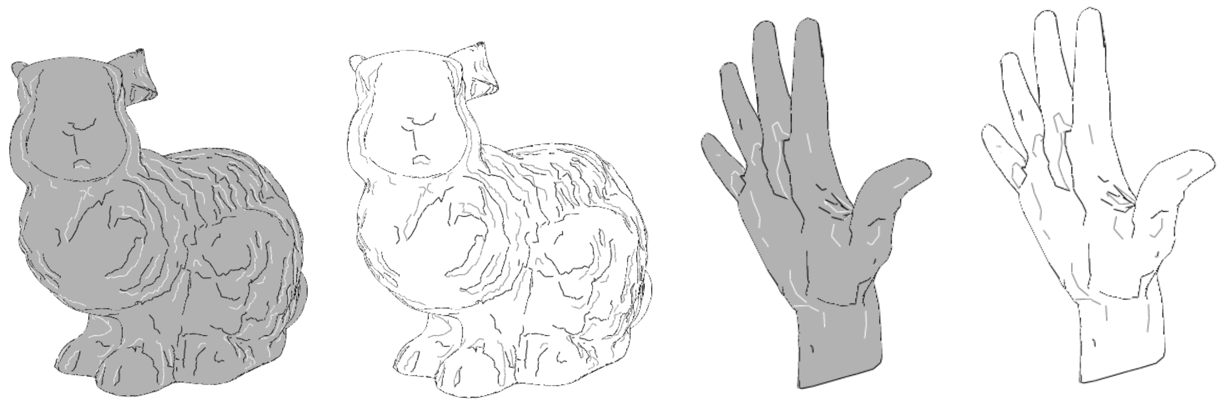


Figure 5: Highlighting the complement of suggestive contours. The lighter strokes are the highlights. The grey background is due to lights turned off.

The suggestive highlights are, by definition in De Carlo and Rusinkiewicz's paper (NPAR 2007), the complements of the suggestive contours in the sense that they are "positive maxima" or $n \cdot v$. That is they contain all points with $\kappa_w = 0$ same as the contours, but the conditions are $Dw\kappa_w < 0$ when $n \cdot v > 0$. This is in order to catch most of the meaningful image ridges. As proposed in the paper, such ridges appear where the intensity changes quickly that is along w and $w_\perp$. The first one is called *suggestive highlights* and the other one is called *principal highlights.* Together with the suggestive contours, we can create a nice looking meshes as well as emphasizing the contours.

Given the amount of time we have, we only manage to implement on the suggestive highlights. This is very similar to how we develop the algorithm for the suggestive contours, just changing the signs $Dw\kappa_w < 0$. The rests are all similar such as `HighlightInfo`, filtering, and thresholding.

The users can change the looking of the mesh by turning on off the surface with 's' key, turing on off the light with 'L' key, and change the colors of the suggestive contours and suggestive highlights to red and green by toggling spacebar.

However, we notice that the looks of the highlights really depend quite a bit on the viewing angle. For some meshes, it might look like random lines crossing the meshes. This might be because, as suggested by De Carlo, that they are not explicitly defined in terms of intensity ridges. We merely observe that they often occur near such ridges.

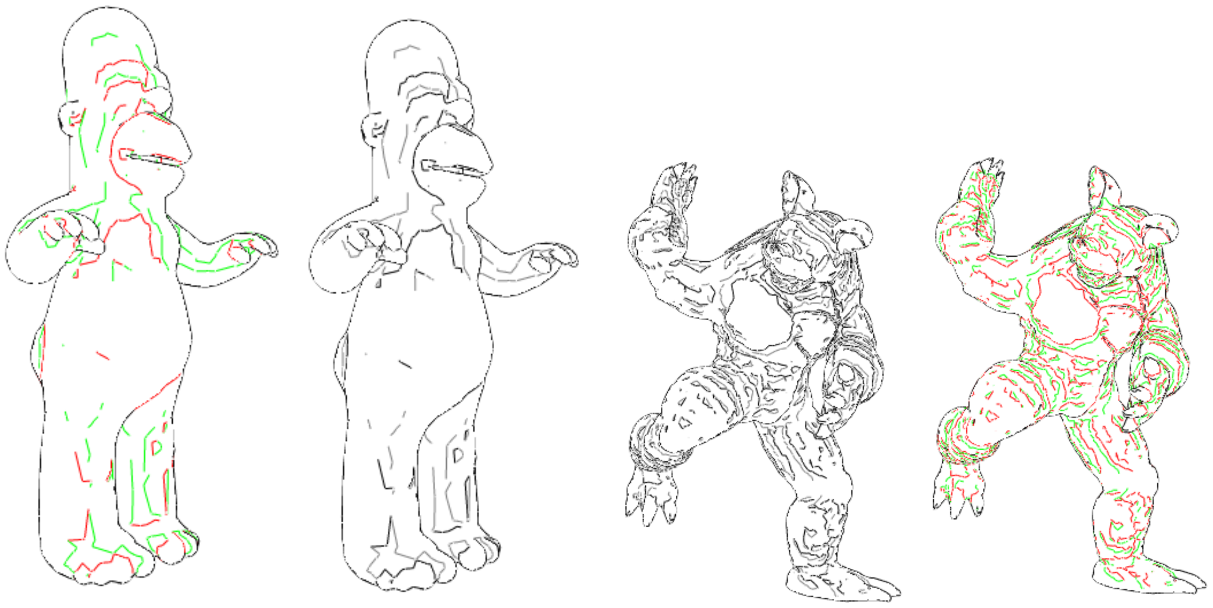


Figure 6: Another render of the highlights.

4 Usage

4.1 Mouse

Button	Function
Left	Rotate around center
Middle	Pan the camera
Right	Scale the meshes

4.2 Keyboard

Key (Uppercase and Lowercase)	Function
'S'	Show surface
'A'	Show axes
'C'	Show curvature cross marks (in red and blue)
'N'	Show normals (in green)
'M'	Show mesh triangles
'L'	Toggle lights on and off
'G'	Show suggestive contours
'H'	Show suggestive highlights
'F'	Filter the spurious lines
spacebar	Toggle red-green color of the contours and highlights
'W'	Write image (with contours) to an svg file
'Q'	Exit program
'+'/'-'	Increase and Decrease $n \cdot v/\|v\|$ threshold
ARROW_UP/ARROW_DOWN	Increase and Decrease $Dw\kappa_w$ threshold
PAGE_UP/PAGE_DOWN	Increase and Decrease filter threshold (length of contours chain)

5 Future Work

These are some features that we tried to implement, but fail.

5.1 Image Generation

This part is suggested in the assignment. Despite the failure of `isVisible` function, there are some bugs in the program that we couldn't manage to solve in time.

Our algorithm is to use quadratic Bezier curve to render the chain of contours. In the beginning, we try to fit every point into the spline by finding addition point to act as the middle control point for every segment of the chain. We are even ambitious to try to make it a C^1 curve, but soon realize that it's just over-fitting. It turns out that the splines wiggle back and forth instead. Finally, we try to make every connecting point of the chain to be control points. That is the spline will fit only half the points in the chain. Some of the splines look fine, but there are bugs in the program that generate something like below.

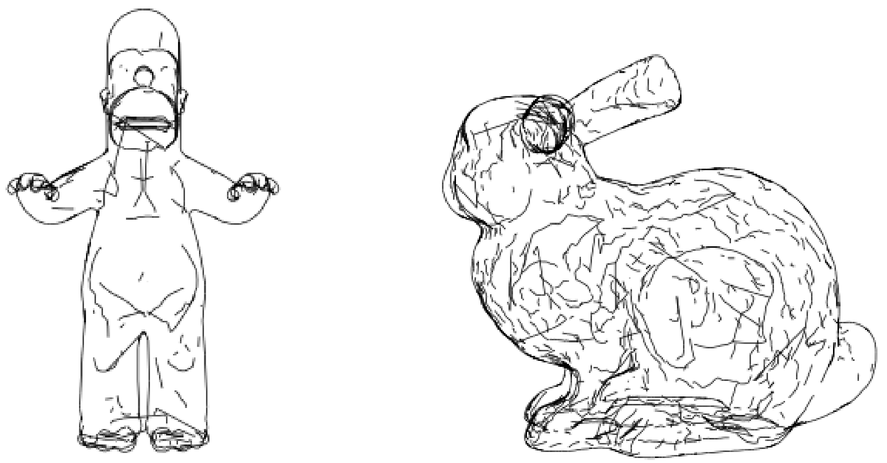


Figure 7: Failed attempts on image generation. This also involves some hard codes to eliminate weird long crossing lines across the meshes.

5.2 Animation Showing Fading Contours

We tried to do animation to rotate the meshes around so that we can see the contours and highlights fading and growing as the meshes rotate around. We expect this since, as described in section 2.2(f), we store the end points of the contour on each face as the real one with $D w \kappa_w = 0$ (or whatever threshold) at the end, not just discretely say it has a contour line crossing it or not at all. So it should behave somewhat continuously when it's changing, but somehow that was not the case here. It might also due to lack of speed in the redisplay or probably just our bad coding.