Scientific Visualization
Project II

Milian Wolff

December 5, 2011

Our second project in the scientific visualization class by Eugene Zhang gave us the opportunity to get acquainted with the algorithms behind "pen-and-ink sketching" of a 3D mesh surface. First we explore the steps required to visualize silhouettes of a geometry. Then we calculate the vertex-based curvature and curvature tensors. Finally we will combine the both to draw a sketch-like version of a given geometry.

The source code of my exercise solutions can be found online under

`https://github.com/milianw/scivi`

Contents

1 Corner Table

As a foundation for the later tasks we first had to implement a corner table. I did this by following the steps outlined in class:

- iterate over all triangles
- add each corner in the triangle to a table
- sort table by $min(c.prev.vertex, c.next.vertex)$ and $max(c.prev.vertex, c.next.vertex)$
- iterate over sorted table and associate consecutive rows with equal min and max entries as opposite corners

The implementation can be found in `CornerTable.java`.

The resulting corner table is essentially a linked list which is straight forward to use. I additionally added a `Corner[] vertexNeighbors()` method to my `Corner` object, which essentially returns the N^1-ring vertex neighbors.

1.1 Validation

To verify the correctness of my implementation of a corner table, I wrote `Ex2_1.java`. When started, the user defines a geometry. Then the corner table for said geometry is computed. Now we iterate over each corner `c` and run a set of assertion-based tests in each step:

- `c.next != null`
- `c.prev != null`
- `c.next.triangle != c.triangle`
- `c.prev.triangle != c.triangle`

Furthermore, the correctness of the "opposite"-association is validated by asserting the following tests:

- `c.triangle != c.opposite.triangle`
- `c.next.vertex == c.opposite.prev.vertex`
- `c.prev.vertex == c.opposite.next.vertex`

- `c == c.opposite.opposite`
- `c.prev.opposite.next.vertex == c.next.vertex`
- `c.prev.opposite != c.next`
- `c.prev.opposite.prev.vertex == c.vertex`
- `c.prev.opposite.prev != c`
- `c.next.opposite.prev.vertex == c.prev.vertex`
- `c.next.opposite.prev != c.prev`
- `c.next.opposite.next.vertex == c.vertex`
- `c.next.opposite.next != c`

Keep in mind though that in open surfaces like e.g. `Gauss_3042.byu` not every corner can be associated to an opposite one. This is handled gracefully in my corner table implementation and unit tests.

Another implicit verification of my implication lies in the extensive usage of the corner table in the following exercises. Since the solutions there seem to be visually correct, the corner table seems to yield correct results.

2 Silhouette

Now we want to find the silhouette of a 3D geometry using two different algorithsm. My solutions to both can be found in `Silhouette.java` and `Ex2_2.java`.

2.1 Face-based approach

The first task was to find the silhouette by iterating over all edges and finding those, where only one of the adjacent triangles is "visible". Here, we consider a triangle T to be visible when

$$(\vec{c}_T - \vec{v}) \cdot \vec{n}_T < 0, \tag{2.1}$$

with $\vec{c}_T$ being the center and $\vec{n}_T$ being the normal of the triangle respectively. $\vec{v}$ is the position of the viewer.

Note that this approach yields undefined results for triangles that are exactly perpendicular to the view ray. There, the dot product will be zero and the triangle is either an invisible backside or a visible frontside. I ignore these edges, assuming that this case only happens for faces somewhere in the middle of the (in-)visible surface and are thus not part of the silhouette anyways.

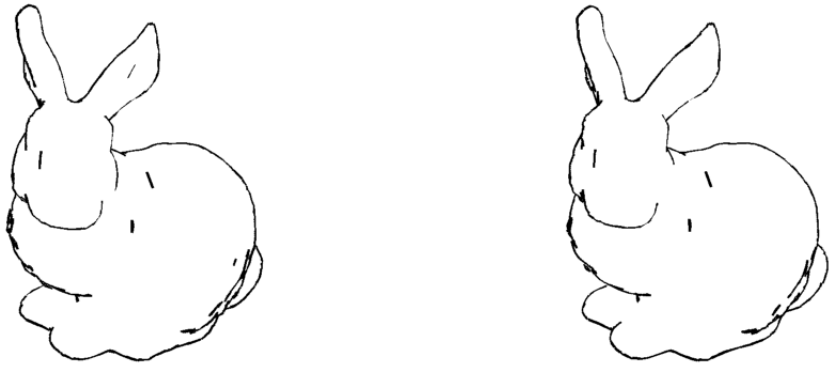
Now to actually find the silhouette, I first iterate over all triangles and store the associated index to a list L when the triangle is visible. Then I iterate over the corner table and find corners c within a visible triangle, i.e. $c.triangle$ is contained in L. Then the edge between $c.prev.vertex$ and $c.next.vertex$ is part of the silhouette, if either $c.opposite$ is undefined, or alternatively $c.opposite.triangle$ is not contained in L.

My implementation of the above can be found in `Silhouette.createFaceBasedSilhouette()`.

2.1.1 Problems

The silhouette is view dependendend and `Ex2_2` hence recalculates the silhouette whenever the view point changes. Now when rotating a given geometry, the main problem of this approach becomes apparent: Gradual changes in the view can have a big difference one the silhouette. Comparing e.g. fig. 2.1a and 2.1b shows this issue at the right ear. Additionally we also see three artifacts, two on the body and one at the left eye.

The same issue is also responsible for the jagged edge of the sphere silhouette as visible in fig. 2.2a.



(a) (b)

Figure 2.1: Face-based silhouette of the bunny under slight rotation

2.2 Vertex-based approach / zero level set

In an attempt to fix the issues of our first approach, we now apply a different scheme that computes the so called zero level set. That is the curve on the surface where the dot product between the curve normal and the view ray is always zero.

To compute this curve we iterate over the corner table and calculate the dot product between the view ray and the normal at the corner vertex c and its previous and next vertex p and n:

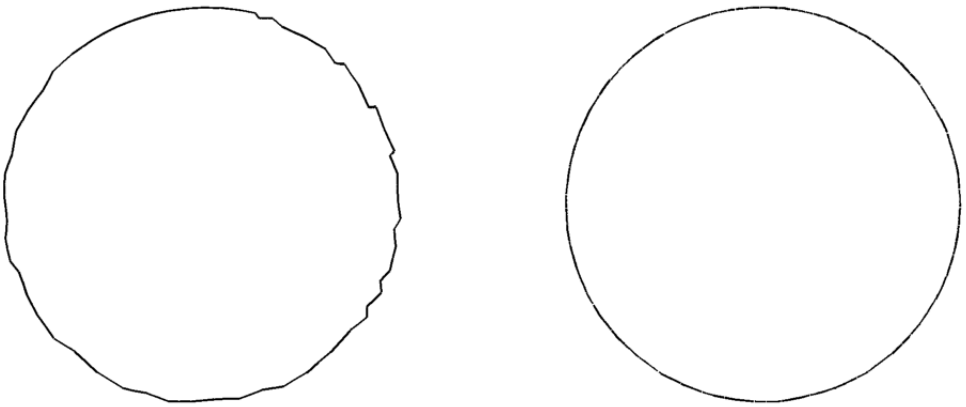
$$d_c = a = (\vec{c} - \vec{v}) \cdot \vec{n}_c \tag{2.2}$$

$$d_p = b = (\vec{p} - \vec{v}) \cdot \vec{n}_p \tag{2.3}$$

$$d_n = c = (\vec{n} - \vec{v}) \cdot \vec{n}_n \tag{2.4}$$

If we then encounter a triangle where either $a \leq 0$, $b \geq 0$ and $c \geq 0$ or $a \geq 0$, $b \leq 0$ and $c \leq 0$, then a zero level set line goes through this triangle. We assume it can be approximated linearly interpolating a and b onto the edge between c and p and analogously for a and c and then connecting the two points for which the interpolated dot product should yield zero.

The code for my implementation can be found in `Silhouette.createVertexBasedSilhouette()` and `Silhouette.findZeroLevel()`.



(a) face-based (b) vertex-based

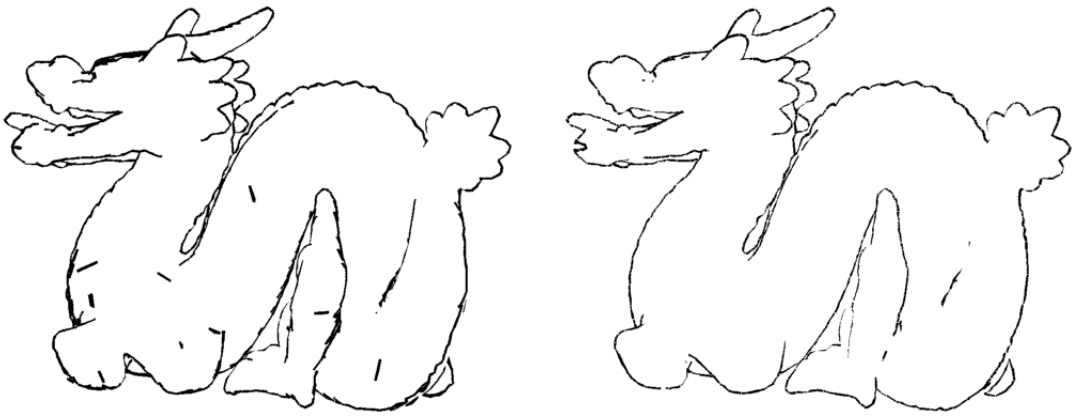
Figure 2.2: Comparison of face- and vertex-based silhouette for the sphere

2.2.1 Comparison

The vertex-based silhouette yields mostly better results, as can be seen in e.g. figs. 2.2 or 2.3. Interestingly enough though, the face-based approach seems to yield a more detailed silhouette for the feline, see fig. 2.4a.

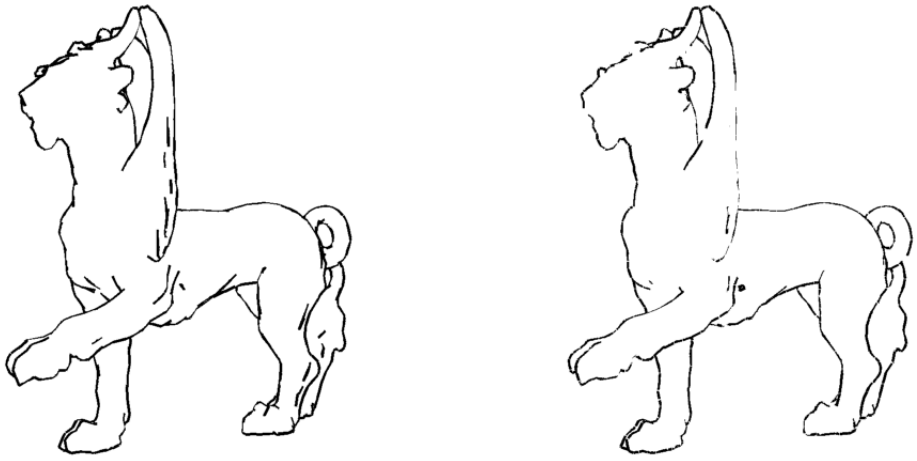
2.3 Silhouette in JavaView

To actually draw the silhouette in JavaView, one has to add the edges to a `PgPolygonSet` using `addPolygon()`. Furthermore we must prevent painting of obscured parts of the silhouette, which can be achieved by keeping the original geometry in the display but coloring its elements completely white using `PgElementSet.setGlobalElementColor(Color.white)` and by disabling lightning via `PvDisplayIf.setLightingModel(PvLightIf.MODEL_SURFACE)`.



(a) face-based (b) vertex-based

Figure 2.3: Comparison of face- and vertex-based silhouette for the dragon



(a) face-based (b) vertex-based

Figure 2.4: Comparison of face- and vertex-based silhouette for the feline

3 Curvature estimation

TODO: paper source