# 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 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.

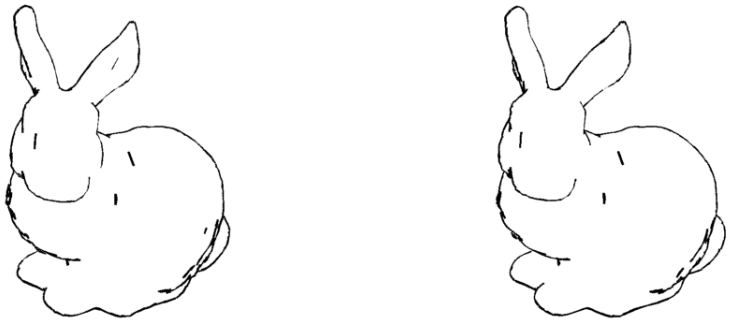
Now to actually find the silhouette, I first iterate over all triangles and store the associated index to a list $L$ when the triangle is visible. Then I iterate over the corner table and find corners $c$ within a visible triangle, i.e. $c.triangle$ is contained in $L$. Then the edge between $c.prev.vertex$ 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 $n$:

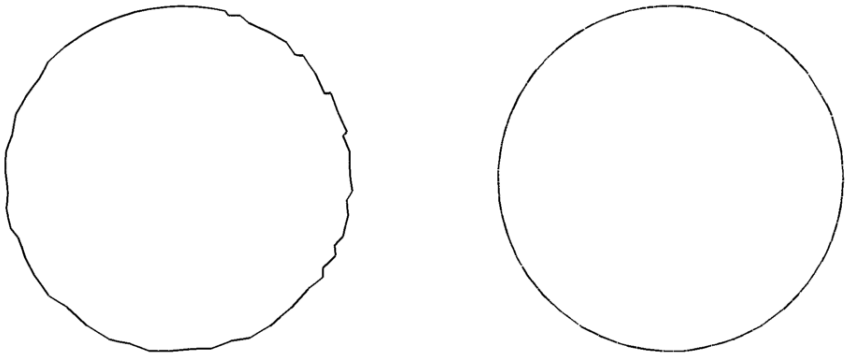
$$d_c = a = (\vec{c} - \vec{v}) \cdot \vec{n}_c \tag{2.2}$$
$$d_p = b = (\vec{p} - \vec{v}) \cdot \vec{n}_p \tag{2.3}$$
$$d_n = c = (\vec{n} - \vec{v}) \cdot \vec{n}_n \tag{2.4}$$

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()`.

### 2.2.1 Comparison

The vertex-based silhouette yields mostly better results, as can be seen in e.g. figs. 2.2, 2.3, 2.4a or 2.5a.



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

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

Personally I find the comparison between the two algorithms as applied to the dragon in fig. 2.3 and the feline in fig. 2.4a the most interesting. While in both cases, the vertex-based approach yields subjectively more correct silhouettes, the face-based approach yields a visually more pleasing result for the silhouette in fig. 2.4a. Here the rendering artifacts, that look bad in the dragon, actually make the image more detailed and hence more similar to an actual "pen-and-ink" scetch. The same happens for the buddha in 2.5a.

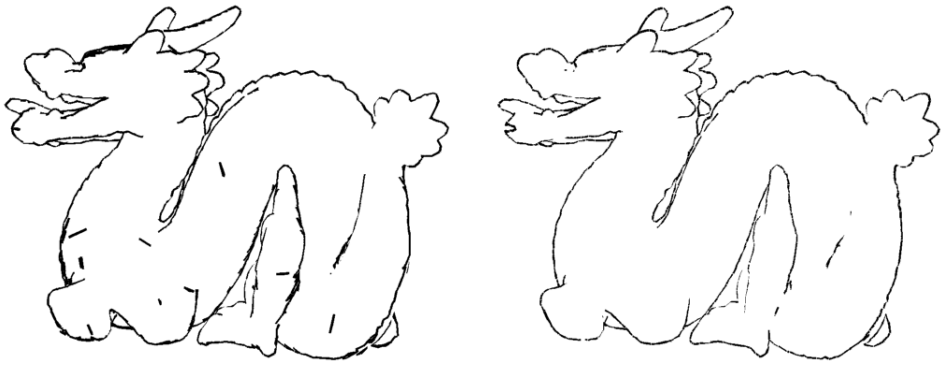
## 2.3 Features

As I just layed out, the silhouette is only one part of an actual painting, albeit a very important one. Generally though, an object has other important features, like e.g. the face of the buddha which is completely bare when viewed up-front in fig. 2.5. As we learned in class this can be handled by additionally taking ridges and valleys into account. Especially the apparent ridges algorithm shown in class should yield good results when applied to the buddha geometry.

For feature-less objects though, like e.g. the sphere, the ridges and valleys approach will not yield any good results. Here we should rather apply an algorithm that takes lightning and curvature into account, like we do later on in the last exercise.

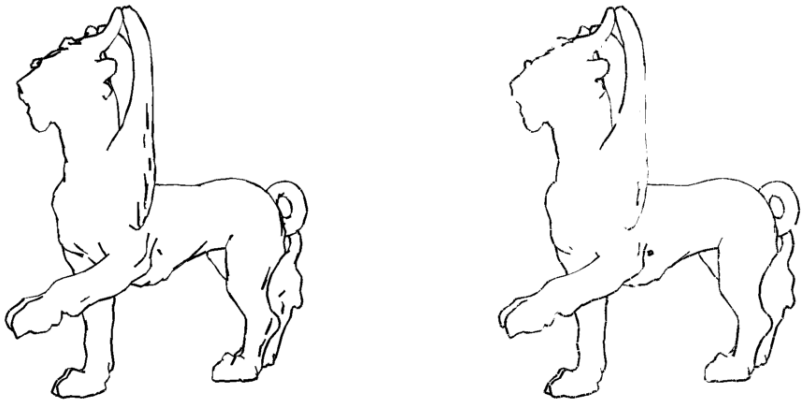
## 2.4 Silhouette in JavaView

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



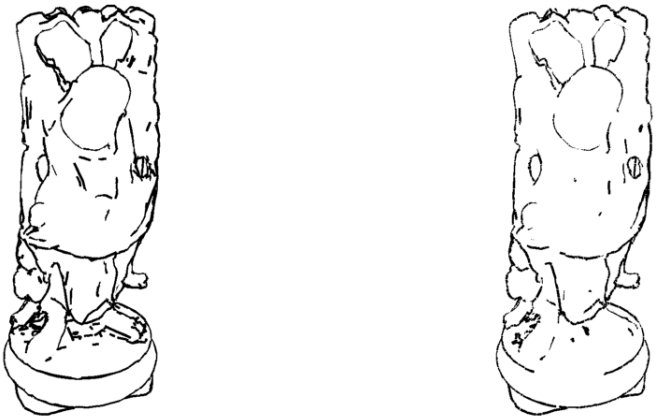
(a) face-based (b) vertex-based

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



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

Figure 2.5: Comparison of face- and vertex-based silhouette for the buddha

coloring its elements completely white using `PgElementSet.setGlobalElementColor(Color.white)` and by disabling lightning via `PvDisplayIf.setLightingModel(PvLightIf.MODEL_SURFACE)`.

# 3 Curvature estimation

In the following we are following the steps and algorithms as outlined by Meyer et al. to estimate the curvature of a discretely triangulated mesh surface. The code to my implementation can be found in `Curvature.java` and `Ex2_3.java`.

## 3.1 Curvature Calculation

The first step is to calculate the mean and gaussian curvature, which can be done in one go. We iterate over our corner table and for each corner accumulate the partial addition to the gaussian curvature $\kappa_G$, mean curvature $\kappa_H$ and area $A_{mixed}$ of the associated vertex. The partial discrete gaussian curvature is simply the angle $\theta_j$ of the corner, compare to eq. 9 of the paper by Meyer et al. To the mean curvature operator we accumulate $(\cot \alpha_j + \cot \beta_j)(\vec{x}_i - \vec{x}_j)$, following eq. 8 of the paper.

Both final curvature values need to be normalized by the area $A_{mixed}$ wich we also accumulate in each iteration step following fig. 4 of the paper.

Additionally I added the possibility to visualize the minimum and maximum curvature values using eqs. 10 and 11 from the paper by Meyer et al.

### 3.1.1 Color Schemes

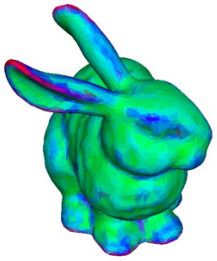
To visualize the curvature estimations, I developed two simple color schemes. Both set vertex colors which are then propagated to the triangles using `geometry.showElementFromVertexColors(true)`.

Both color schemes leverage the HSV color space with maximum saturation and value but varying hue component.

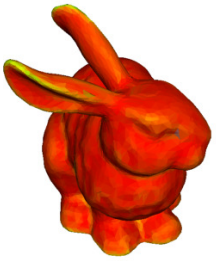
#### Maximum Color Scheme

My first approach was to simply map the range $[min, max]$ to the hue range $[0, 360]$, which can be seen in e.g. fig. 3.1b or 3.1d. The minimum value is zero for the mean curvature and $max$ is the maximum of the computed curvature values. For gaussian curvature we compute $max = -min$ to be the absolute maximum of the computed curvature values.

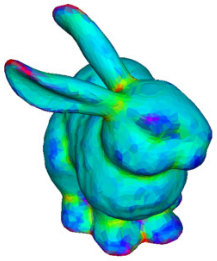
This approach turned out to be a bad choice since the curvature values vary over a great range. Especially the extreme values are only encountered rarely though, making the linear mapping between curvature range and hue range impractical. Due to that this color scheme mostly creates rather equicolored images.



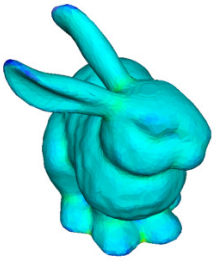
(a) mean curvature, deviation color scheme



(b) mean curvature, maximum color scheme



(c) gaussian curvature, deviation color scheme



(d) gaussian curvature, maximum color scheme

Figure 3.1: Mean and gaussian curvature of the bunny

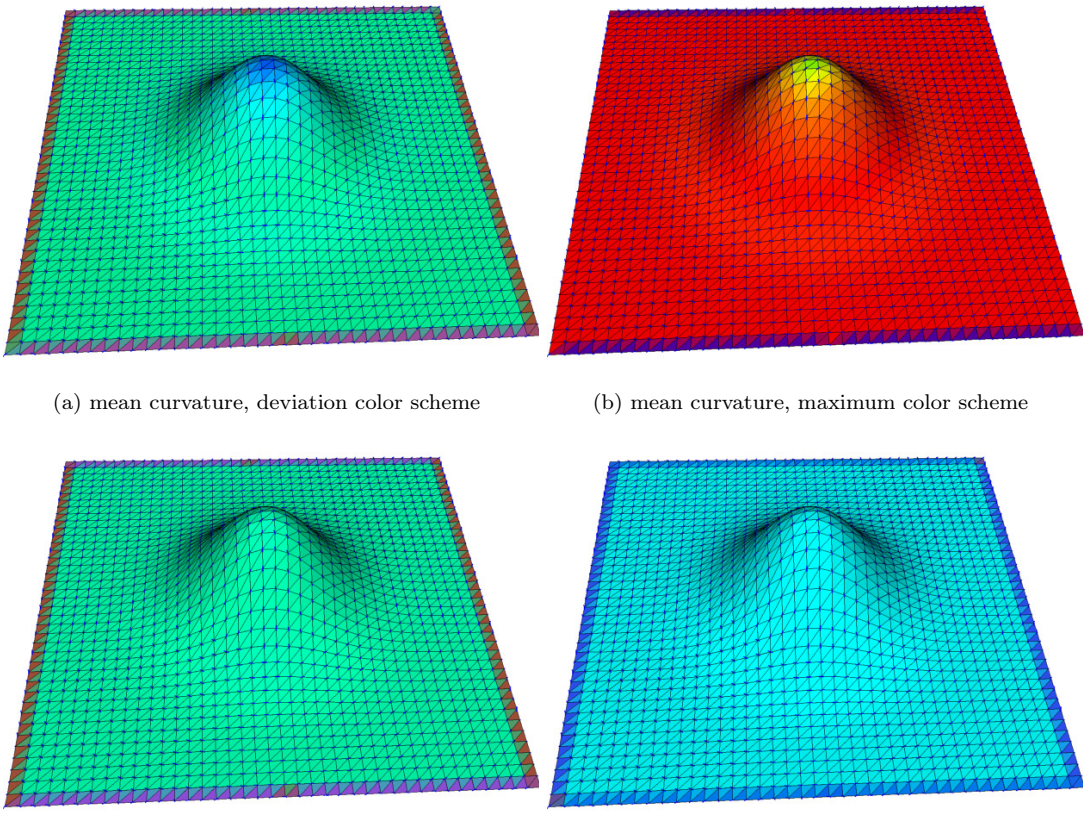
#### Deviation Color Scheme

To fix the issues of the maximum color scheme, I decided to assume the curvature to be normal distributed. By calculating the mean value $\mu$ and the standard deviation $\sigma$ I can now map the curvature values in the tripled deviation interval $[\mu - 3\sigma, \mu + 3\sigma]$ to the hue range $[0, 360]$. This covers 99.73% of the curvature values, the rest is simply capped to the maximum hue values 0 or 360.

This coloring scheme can be seen in e.g. 3.1a and 3.1c.

### 3.1.2 Open Surfaces

To handled open surfaces like e.g. in fig. 3.2 somewhat gracefully, vertices at the edge of a surface are ignored.



(a) mean curvature, deviation color scheme

(b) mean curvature, maximum color scheme

(c) gaussian curvature, deviation color scheme

(d) gaussian curvature, maximum color scheme

Figure 3.2: Mean and gaussian curvature of the `Gauss_3042.byu` model with open surface mesh

## 3.2 Curvature Tensor

To compute the curvature tensor $\mathbf{B}$ we iterate over all vertices and then calculate the $\vec{d}_{i,j}$ values (see p. 14 of the paper by Meyer) for the 1-ring neighbors using the corner table. This gives us a potentially over-determined linear equation system in the curvature tensor variables $l$, $m$, $n$. Following the steps outlined in class 7 (pages 16, 17 of the slides), we can multiple the eqation system from left with the transposed coefficient matrix, yielding a square coefficient matrix that can then be solved by applying Cramer's rule.

While it is quite easily explained, this took me quite some time to implement. The code can be found in `Curvature.computeCurvatureTensor()`.

### 3.2.1 Tensor Visualization

To visualize the curvature tensor, we compute the principle directions of it, i.e. the two dimensional eigen vectors $\vec{e}_{minor}$ and $\vec{e}_{major}$. These are easily mapped to three dimensions by blending the three dimensional tangent plane vectors $x$, $y$ by the associated value of each eigen vector.

These vectors can then be added to a `PgVectorField` in JavaView with a user defined length to visualize the curvature tensor.

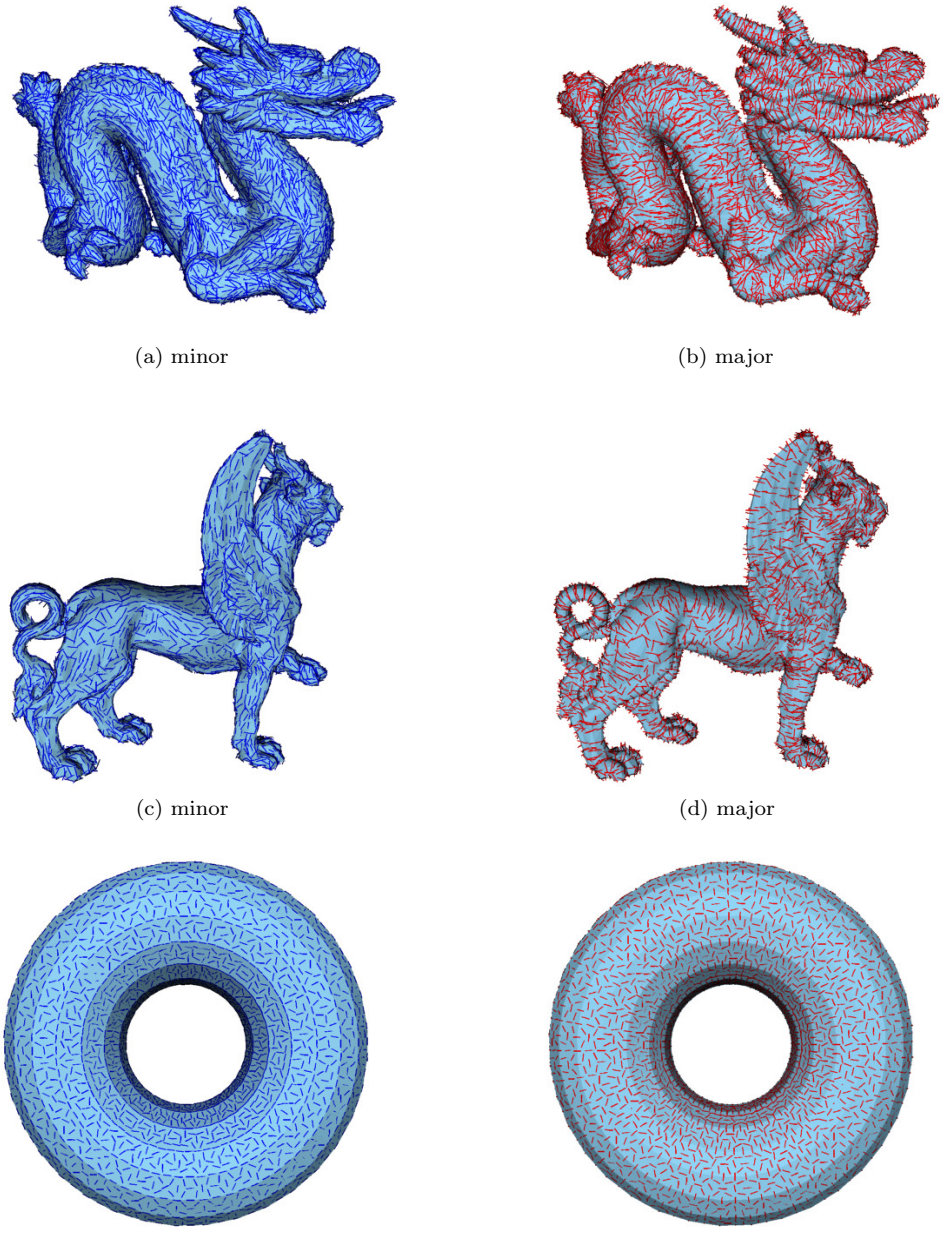
A few results can be seen in 3.3, the code to my implementation can be found in `Curvature.computeCurvatureTensorFields()`.

## 3.3 Total Angle Deficit

The sum of the angles $\theta_{i,j}$ turns out to always be an integral multiple of $2\pi$. And indeed I find $N - \sum_{i,j} \theta_{i,j} = \chi$, with $N$ being the total number of vertices in the geometry and $\chi$ the euler characteristic of the geometry.

This does not work for geometries with curvature singularities though, like e.g. the hand. There the angle deficit will be improperly calculated and hence does not map to the euler characteristic anylonger.

To verify this, run `Ex2_3.java` on a selected geometry. The total angle deficit and euler characteristic will be outputted to the console.



(a) minor (b) major

(c) minor (d) major

(e) minor (f) major

Figure 3.3: Minor and Major principle directions of the curvature tensor

## 3.4 Tensor Smoothing

As seen in fig. 3.3, the raw curvature tensor is rather noisy. For a more visually pleasing image, we must smoothen it.

To do this we first scale the local $2x2$ curvature tensor into the global $3x3$ frame, following the steps outlined in class and in the recitation. The source code for this can be found in $VertexCurvature.globalCurvature()$.

Now we can smoothen the tensors using either the explicit or forward-Euler scheme or the Gauss-Seidel scheme. Both can be run for a user-defined number of times with a user-defined step size. Furthermore we can apply different weights, either uniform, cord, cotangent or mean value based. See the slides to class 9 for details.

My implementation can be found in `Ex2_3.java` and `Curvature.smoothTensorField()`.

### 3.4.1 Weighting Comparison

To compare the weighting types, see e.g. fig. 3.4.

#### Uniform

The uniform weighting is fast and gives reasonable results.

#### Cord

Weighting the tensors by the reziproke edge distance turns out to give worse results compared to the uniform weighting.

#### Mean Value

Weighting the tensors by mean value gives better results than the cord weighting, yet interestingly enough does not seem to be as effective as the uniform weighting. It is also noticeably slower for large geometries or number of steps.

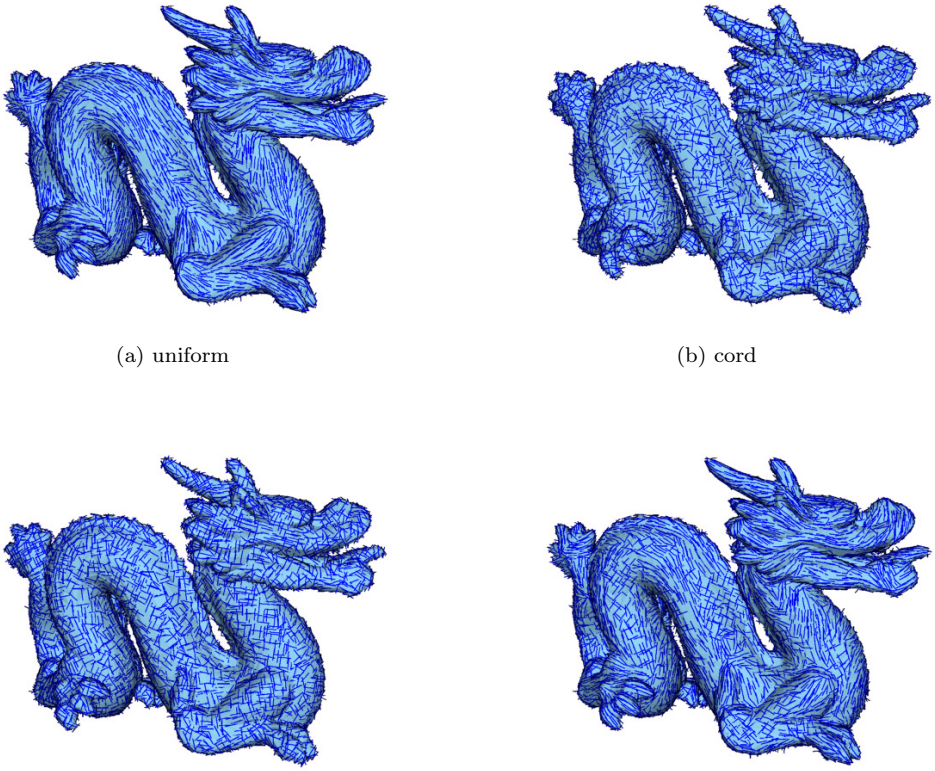
#### Cotangent

The cotangent weighting yields comparable results to the mean value approach.

### 3.4.2 Scheme Comparison

The forward-Euler smoothing-scheme works well for small step sizes and hence requires many steps to achieve reasonable results. This of course directly means that it is comparatively slow since the Gauss-Seidel scheme works well for large step sizes and hence does not require that many steps. See e.g. 3.5 for a comparison.

Personally I see the reason for faster convergence to the smoothened field when applying Gauss-Seidel in the fact that it takes the potentially smoothened tensors, and not the results from the previous step.



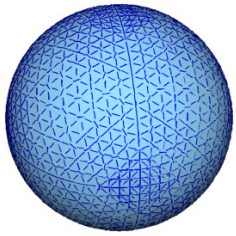
(a) uniform

(b) cord

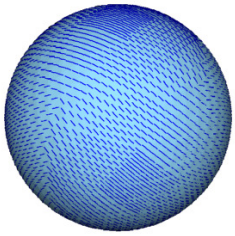
(c) mean

(d) cotangent

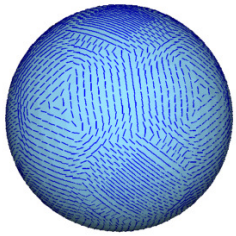
Figure 3.4: smoothened tensor field of the dragon using different weights: steps: 10, step size: 0.1, scheme: forward-euler



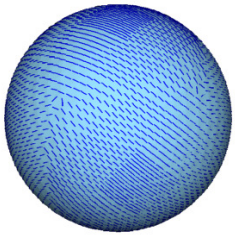
(a) raw



(b) Gauss-Seidel, 10 steps, 1.0 step size



(c) Forward-Euler, 10 steps, 0.1 step size



(d) Forward-Euler, 20 steps, 0.1 step size

Figure 3.5: tensor field of the sphere before and after smoothing

# 4 Pen-and-Ink

After hours of work we can now finally plug it all together to create a "pen-and-ink" sketch image of a 3D geometry. `Ex2_4.java` does just that: First we compute the smoothened tensor field of a geometry, then trace the original geometry in the direction of either the minor or major principle direction. We then render the scene four times, once to obtain a grayscale image, once to find the silhouette and then once for the major and once for the minor principle direction trace. These four images are then combined such that:

- the silhouette is always painted
- bright areas stay white
- greyish areas copy either the major or minor trace image
- dark areas copy both, the major and minor trace images

## 4.1 Tracing

I sadly do not have the time to persue this task in-depth, and thus only created a proof-of-concept trace. My implementation in `Ex2_4.getTrace()` iterates over the triangles in the geometry and averages the global curvature tensors of its vertices. This is then considered to be an estimation of the curvature tensor at the center of the triangle.

We then scale the tensor down into the local $2x2$ frame of the triangle and compute the intersection points with the edges of the triangle. This we then scale up into the global $3x3$ frame again and draw a line between the points using `PgPolygonSet` similar to the silhouette drawing.

## 4.2 Greyness Segmentation

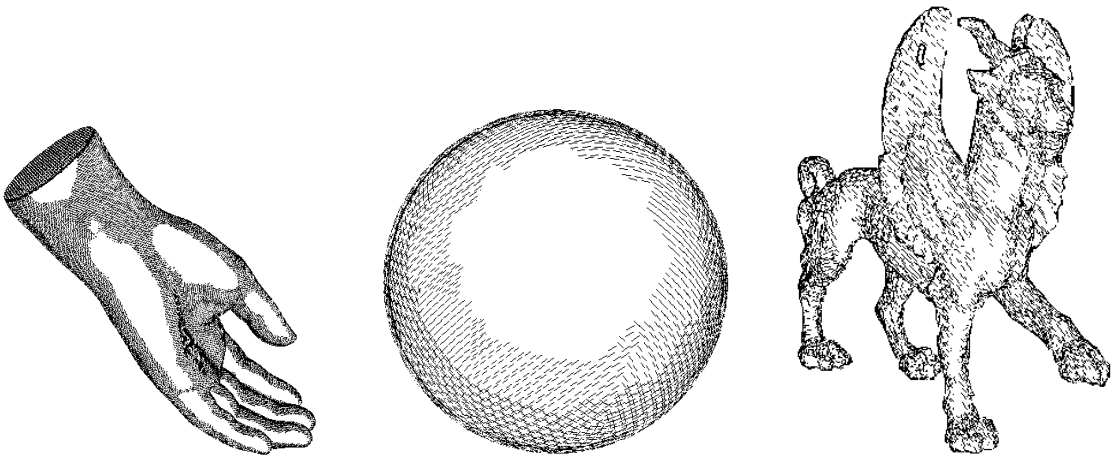
To decide whether a region is bright or dark, we compute the greyness integer value at a given pixel position of the grayscale image, using the perceived greyness formula:

$$greyness = 0.212671red + 0.715160green + 0.072169blue \tag{4.1}$$

This gives us a value in the range of $[0, 255]$. The user can then define a threshold below which a pixel is considered to be in a "dark" area, and another threshold above with the pixel is considered to be in a "bright" area. Between these two lies the greyish area.

## 4.3 Results

A few of my results can be seen in fig. 4.1. It is apparent that even my proof-of-concept tracing already gives interesting results. Considering that it only paints one line per vertex, it already produces quite noisy images for surfaces with a fine mesh. I am interested to see how the algorithms described in the exercise turn out, and whether they are less noisy - which I severly doubt since they seed every element but paint the trace lines into adjacent vertices. Most probably the only real improvement will be a trace algorithm using geodesics to get equidistant, non-overlapping trace lines.



(a) hand (b) sphere (c) feline

(d) dragon (e) bunny (f) buddha

Figure 4.1: pen-and-ink scetches of different 3D geometries, traces in minor direction