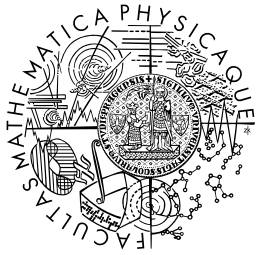

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tomáš Nekvinda

Fixing of Facial Triangle Meshes

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	RNDr. Josef Pelikán
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2018

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank Mr. Pelikán for patience, willingness, and all consultations during the three semesters; David for his calming and for substituting a rubber duck. I also would like to thank Lucy for sharpening my language, and I have to thank you all who provided me with food supplies and did not disturb me while reinventing the wheel.

Title: Fixing of Facial Triangle Meshes

Author: Tomáš Nekvinda

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán, Department of Software and Computer Science Education

Abstract: The purpose of this work is to create an algorithm for automatic cleaning and trimming of three-dimensional facial scans. In view of that, we have developed an algorithm which consists of three central parts. The first part is a novel landmark detection algorithm based on discrete differential geometry and machine learning methods. The second part is represented by a method removing defects, spikes, and blobs and the last part is an enhancement of an algorithm for hole filling. The outcome of this work is a program which can automatically clean and trim three-dimensional facial scans and moreover, it can detect nose tip, nose root, and mouth and eye corners. As our testing has shown, the program performs well on facial scans produced by the optical scanner Vectra3D.

Keywords: triangle mesh, face, 3D scanner, cleaning, landmark detection, trimming, hole filling

Název práce: Čištění obličejových trojúhelníkových sítí

Autor: Tomáš Nekvinda

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán, Katedra softwaru a výuky informatiky

Abstrakt: Hlavním cílem této práce je navržení algoritmu pro automatické čištění a ořezávání obličejových skenů. Za tímto učelem jsme vyvinuli algoritmus, který sestává ze tří hlavních částí. Jednou z těchto částí je nový algoritmus pro detekci obličejových landmarků, který je založený na poznatcích diskrétní diferenciální geometrie a na metodách strojového učení. Další část se zabývá rozpoznáváním a odstraňováním geometrických a topologických defektů. Poslední část si klade za cíl vyplňování děr v trojúhelníkových sítích, k čemuž je využíváno vylepšení jistého stávajícího algoritmu. Výsledkem této práce je program, který dokáže automaticky ořezávat a odstraňovat typické nedostatky obličejových skenů. Navíc dokáže detekovat sadu výrazných obličejových bodů – špičku a kořen nosu, ústní a vnitřní oční koutky. Při testování se ukázalo, že si program dokáže bez potíží poradit s daty produkovanými optickým skenerem Vectra3D.

Klíčová slova: trojúhelníková síť, obličej, 3D skener, čištění, detekce landmarků, ořezávání, vyplňování děr

Contents

1. Introduction

Nowadays, three-dimensional imaging and scanning technologies have become more affordable. Due to precision improvements, they have been more frequently used to solve and study novel problems. Three-dimensional scanning has been useful in prognostic and therapeutic decision-making for a wide range of medical professions [Farahani et al., 2017]. As seen at the Department of Anthropology and Human Genetics (DAHG), Faculty of Science at the Charles University, high-resolution facial scans help us to understand the development of sexual dimorphism [Koudelová et al., 2015a], make the comprehension of the facial growth possible [Koudelová et al., 2015b] or facilitate the analysis of medical interventions [Moslerová et al., 2018].

However, models produced by contemporary three-dimensional scanners usually need post-processing. It is needed because of limitations of scanning technologies on the one hand, and because scanners prefer precision to correctness on the other. The primary purpose of this work is to create a program which would help DAHG by making the post-processing of three-dimensional facial scans automatic or at least semi-automatic.

The post-processing part can be characterised by two steps: the object trimming and the removal of defects. So far, employees of DAHG have been trimming facial areas of three-dimensional scans manually with the help of a commercial mesh processing software called Rapidform 2006 (developed as Geomagic Wrap at present). That is why we have decided to compare our results with tools and features of the commercial Geomagic Wrap 2017 (by 3D Systems Inc.) and also of an open source alternative – MeshLab v2016.12 (by IST-CNR).

1.1 Contribution

We developed an application for cleaning three-dimensional facial scans. It can automatically detect six facial landmarks, trim the facial area, remove some geometrical defects and repair the topology. Moreover, the resulting cleaned objects can be rotated into a uniform direction. The program focuses on batch processing and thus can be run on large datasets without a need for user interaction.

1.2 Outline

First, we summarise frequently used terminology (Chpt. 2). In Chapter 3, we analyse our development data which we were given by DAHG. We also discuss the representation of three-dimensional objects (Chpt. 4). The next chapter (5) describes our cleaning algorithm. The description begins with the algorithm outline (Chpt. 5.1) and continues with an introduction to the means of geometry used in this work (Chpt. 5.2). Afterwards, essential steps of the algorithm are explained: the landmarks detection (Chpt. 5.3), the trimming (Chpt. 5.4), the removal of defects (5.5), the hole filling (Chpt. 5.6) and remaining steps (chapters 5.7, 5.8). At the end of this work, we present some technical information (Chpt. 6) and user documentation (Chpt. 7).

2. Terminology

Mesh – a collection of vertices, faces, and edges which form an object. Meshes formed by triangular faces are called triangle meshes.

Vertex – a data structure which describes a spatial point. It stores spatial coordinates of the point and usually the corresponding normal vector or some more information about the point, such as colour, and texture coordinates.

Edge – a boundary segment joining a pair of two different vertices of a polygon.

Face – a data structure defining a polygon formed by vertices. In general, faces can have many vertices, but we consider only triangular faces in this work (i.e. faces having three vertices). Hereafter, the terms face, triangular face and triangle will be interchangeable.

Normal – a directional vector which is perpendicular to the surface tangent plane at a certain point of the surface. A normal can point in two opposite directions. However, if it is possible to make a consistent choice of the surface normal vector at every point, it can be used to describe the surface orientation. In this work, we suppose normals pointing out from the scanned faces (in the view direction of the faces).

Corner – a data structure which stands for the part of a face where two edges meet. Every corner usually stores a pointer to its containing face and also to the vertex which is formed by the intersection of the two mentioned edges.

Dihedral angle – an angle between two intersecting planes (in fact, there are two angles, but as they are conjugate, we should not be misunderstood). If we refer to the dihedral angle of two triangles, we mean the dihedral angle between the planes defined by the triangles.

n-manifold – a topological space whose each point has a local neighbourhood which is homeomorphic to the Euclidean space of dimension n; 2-manifolds are called surfaces.[1]

[1]For better understanding, imagine a railway. Naively, it may seem as a two-dimensional object because we can change our position (longitude and latitude) by train. However, a blind train operator (which can control just the forward-backwards lever) could think without any doubts that the railway is a one-dimensional object. Thus, we could state that the railway is a 1-manifold.

3. Data Analysis

We are now going to analyse our development data. First, we will briefly explain principles of three-dimensional scanning. Then, we describe the set of development meshes – their typical properties, errors, and quality. All of our 194 facial scans were acquired by the optical scanner Vectra3D [Canfield Scientific, Inc., 2018]. However, we believe that all meshes produced by contemporary scanners are similar and that the following description will characterise meshes produced by scanners in general.

3.1 Principles of Scanning

There are many types of different scanning principles. Medical imaging methods (computed tomography or magnetic resonance) usually need to capture the inner structure of the human body; thus they create a discrete three-dimensional volumetric representations of objects.

However, we focus on different type of scanners. These scanners produce surface meshes and usually use cameras together with lasers or complex light patterns which are projected onto the scanned object surface. The time-of-flight or the triangulation principles are then used to collect a set of data points within three-dimensional space. The set is called a *point cloud*. The point can be combined with surface textures or colours caught by scanner cameras to fully reconstruct a three-dimensional model. Accordingly, the reconstruction step involves combining the collected data and converting the point cloud into a triangle mesh. However, the conversion is the primary source of topological and geometrical errors. On the one hand, the scanner software must preserve the original captured data – the data can have inaccuracies caused by the point cloud capturing. On the other hand, surface reconstruction is a difficult task and scanner software might create more errors.

3.2 General Characteristics

As we have already mentioned, we obtained 194 facial scans. These scans were taken by the optical scanner Vectra3D. The meshes have approximately 200 thousand vertices and around 400 thousand faces. Most mesh triangles are almost equilateral, the triangulation is mainly regular, and vertices usually have about six neighbouring vertices.

None of the available scans forms a single connected object. It means that there are multiple connectivity components forming separate groups of vertices. Therefore, it is impossible to walk from a component to another component by edges. We call the component with the largest number of vertices *major component* and all other components *minor components*. An average mesh has 150 connectivity components, and minor components contain around 10 thousand vertices.

We should also mention the scale of the meshes. Usually, scanners produce data with a unit equal to one millimetre in reality, although the unit-to-millimetre

correspondence can be set in the configuration file of our program (see Appx. A). We will suppose that a mesh unit is equal to a millimetre further in this work.

The meshes are also provided with textures. A single texture consists of two photos, one from the left side and one from the right side of the face (Fig. 3.1). Texture coordinates are defined for each triangle separately.

Figure 3.1: Left: *An example of the texture produced by the optical scanner Vectra3D.* Right: *A texture produced by Rapidform 2006 post-processing. Note that it was obtained from the texture on the left.*

3.3 Topological Errors

We would like the result of our algorithm to be a disk surface without any holes or any handles (not like a teapot). Despite this, we do not apply any complex checks of the input mesh topology. We are mainly interested in the mesh holes, and we check whether the mesh is a valid surface or rather a proper 2-manifold [Lee, 2011]. Now, we are going to describe which situations usually do lead to an invalid definition of a topological surface.

Non-2-manifold vertices

A mesh can be an improper 2-manifold because of a vertex which is lying on multiple surfaces. It means that it is impossible to walk around the vertex so that we visit all adjacent triangles and we do not go through the vertex (Fig. 3.2). This type of error is usual and is present in all scans we were given. However, there are usually around 50 erroneous vertices which is not much compared to the total number of mesh vertices – hundreds of thousands.

Non-2-manifold edges

Similarly to vertices, edges can also violate 2-manifoldness. Every edge of a proper surface should be adjacent to at most two faces. So three or more faces cannot have a common edge (Fig. 3.2). Although we check edges for this type of error, scanner software probably does not produce meshes with non-2-manifold edges because all the scanned meshes we have seen do not contain any.

Holes

Scanners fill some holes during the reconstruction of the surface. It is evident from an unusual triangulation in some parts of meshes. Though small holes can be sometimes missed and then the mesh is no longer without holes. Our development meshes contain up to ten holes typically.

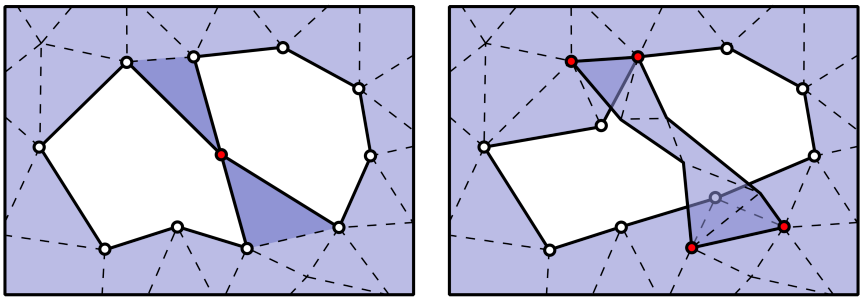

Figure 3.2: Left: *Two holes and a red vertex which corrupts 2-manifoldness.* Right: *Two non-2-manifold edges. Four vertices that are adjacent to these edges are highlighted.*

3.4 Geometrical Errors

Unlike topological errors, geometrical errors are usually present in scanned meshes because scanners are uncertain in some areas. The uncertainty can be caused by adverse light conditions during scanning or by the reflectivity of scanned objects. In case of facial scans, geometrical errors are usually located in eyes, eyebrows, and hair. Here is a summary of the observed problems.

Intersecting triangles

Sometimes, it is impossible to reconstruct parts of a scanned object due to missing data. If a missing area is small, the scanner software will probably decide to fill the hole. However, hole filling is a considerably complex task and intersecting triangles might be created during this process. There are 1.2 % of intersecting triangles on average in our development set of meshes.

Spikes and blobs

This kind of defects originates from the effort of scanners to be precise. Some tiny details or moving objects might be captured only from a particular point of view or at a specific time, and then the reconstruction process might produce weird artefacts in the resulting scan. See Fig. 3.3 with some examples.

Thin triangles

We have also noticed thin triangles in general meshes. These triangles are not so problematic, but zero-area triangles (a special case of a thin triangle where all

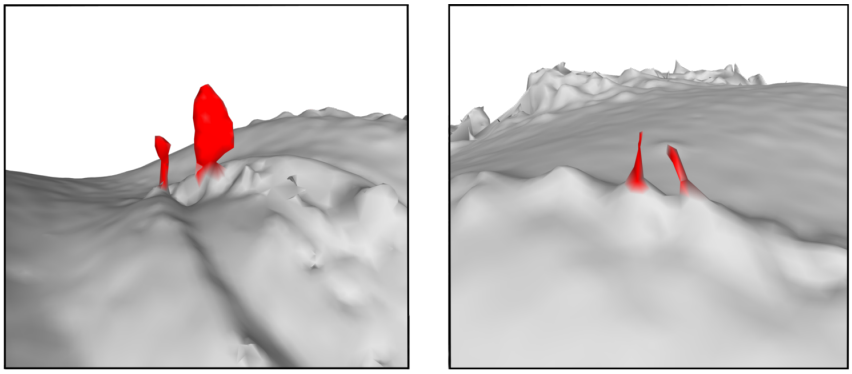

Figure 3.3: Left: *A detail of an eye with two large blobs (red colour).* Right: *A part of an eyebrow with spikes (red colour).*

three points are on the same line) might cause problems while processing meshes. That is why the resolving of thin triangles is helpful. Moreover, it might improve the quality of a mesh triangulation.

4. Triangle Mesh Representation

At the beginning of our algorithm, we need to load an input mesh. We would like to store it in memory in a suitable way so that we can do all operations (which will be briefly described in the following chapters) easily. An action we do very often is obtaining vertex neighbours, so we have decided to store a list of neighbouring vertices for each vertex. Further, the desired data structure should allow us to go through a vertex one-ring neighbourhood clockwise or counter-clockwise. It would also be useful to have the ability to get an opposite vertex (with regard to an edge).

4.1 Corner Structure

To satisfy these requirements, we use a corner data structure [Rossignac et al., 2003]. Consequently, there are three principal objects – vertices, corners and faces (Fig. 4.1). A vertex holds lists of its neighbours, adjacent corners and other vertex-specific data like its normal or curvature (Chapter 5.2). Every corner stores which face and which vertex it belongs to and what are the previous and the next corners. It also manages the list of opposite corners. We do want corners to be able to store a list of opposite corners because it allows us to represent a mesh with edges having more than two adjacent triangles (which should not be present in a 2-manifold mesh). Every face holds an array of its three corners. These are stored for each face in the same order – clockwise or counter-clockwise. Faces have to handle a texture coordinate for each of its corners.

Let us denote the number of vertex neighbours M. We suppose that M is typically a small number and can be considered the same for all vertices. Then by using the described structure, we can obtain opposite vertices and neighbours of a vertex in $O(1)$ time, boundary neighbours of a boundary vertex in $O(M)$ time and we can do vertex addition (including connecting with neighbours by M faces) or vertex removal in $O(M^2)$ time.

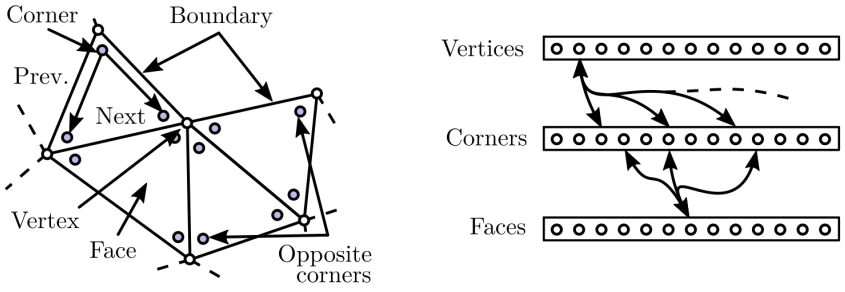

Figure 4.1: Left: *There is a local vertex neighbourhood with vertices, corners (also a previous and a next corner), faces and a boundary depicted.* Right: *A visualisation of corner structure. Vertices hold adjacent corners and vice versa. All faces have exactly three corners. A corner can also get its corresponding face.*

4.2 Supported File Formats

We support importing and exporting of Wavefront .OBJ files and Polygon File Format (.ply) files both in binary and ASCII formats. We are going to describe the formats briefly; please see Wavefront Technologies [1998] and Turk [1994] for the full description of .OBJ and .ply files, respectively. We expect only triangle meshes with or without per-vertex normals and per-face texture coordinates.[1] Any other meshes or materials (e.g. vertex colours) are not supported and are ignored.

A Wavefront .OBJ file contains at most one definition of a vertex, a vertex texture coordinate, a vertex normal or a face per line (Fig. 4.2 left). A vertex can be specified in a line starting with the letter `v` followed `x` , `y` and `z` coordinates delimited by whitespace characters. Texture coordinates and normals are defined in the same way, but the line must start with the letter `vt` or `vn`, respectively. Faces join the defined objects together and can be defined by a line starting with `f` followed by three entries of the format `vertex_idx/txt_coord_idx/normal_idx`. The index of an object is given by order of lines defining a specific type of objects.

The first part of a .ply file is a header (Fig. 4.2 right). It starts with the line containing just `ply`. Then, there must be present the format specification (ASCII, binary). The header itself contains definitions of elements which can have some properties. Properties might include a single number or a list of numbers. The type of numbers must be specified (e.g. `float32`, `int32`). Element and property definitions must declare its unique name. Moreover, the element definition is required to specify the number of these elements in the file. The header ends with the `end_header` line. Past the end of the header, element data are stored in the precisely same order and with the same properties and types as stated in the header. ASCII and binary variants differ in the way of storing the data parts.

```
# this is a comment
# definition of vertices
v 0.0 1.0 0.0
v 1.0 0.0 0.0
v 0.0 0.0 0.0
# definition texture coordinates
vt 0.5 1
vt 0.75 1
vt 0.25 0.75
# definition of normals
vn 0 0 1
# definition of faces
f 1/1/1 2/1/2 3/1/3
```

```
ply
format ascii 1.0
element vertex 3
property float32 x
property float32 y
property float32 z
element face 1
property list uint8 int32 vertex_idx
end_header
0.0 0.0 0.0
0.0 0.0 1.0
0.0 1.0 1.0
3 0 1 2
```

Figure 4.2: Left: *A Wavefront .OBJ file defining a single triangle with texture coordinates and normals.* Right: *A triangle stored in the .ply format.*

[1] As the future work, the program can be extended to also support per-face normals and per-vertex texture coordinates. However, the current implementation cannot be easily modified to obtain that.

5. Algorithm Description

In this chapter, we will outline our cleaning algorithm, its individual parts and connections between those parts. Later, we will describe and discuss some of the parts in more detail.

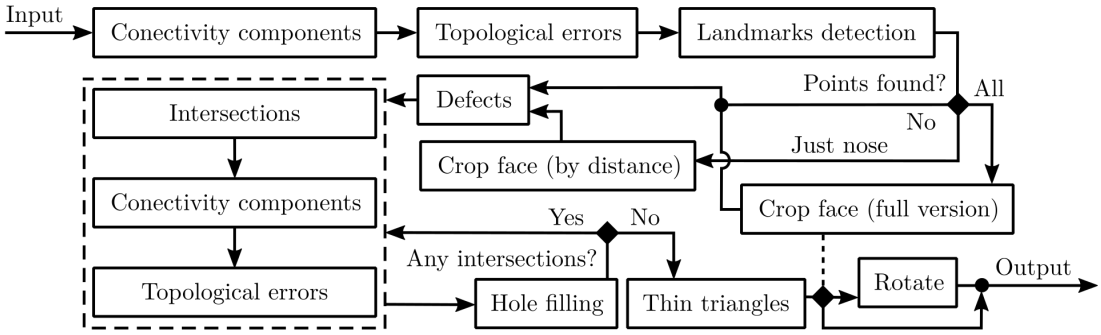

Figure 5.1: *The algorithm scheme. Arrows connect subsequent phases; rhombi depicts decision nodes. Dashed rectangle groups some phases together and dashed line specifies the dependency on the previous phase execution.*

5.1 Algorithm in Outline

The whole algorithm consists of several phases (Fig. 5.1). Some phases are aimed to solve specific problems (e.g. face cropping) and some of them are general and can also be used to solve other types of tasks (e.g. hole filling). We are going to introduce the phases in the same order as they are processed in the algorithm:

- In the first phase, the input facial mesh is reduced by removing all minor components. The components are found using the Depth-first Search algorithm. We suppose that minor components do not have any importance and the whole facial area is connected. If any of the components to be removed is too large, the user is informed about this rare situation. This step is the first one because it can reduce the size of the data noticeably and can enable further phases to be processed faster.

- In the next step, all topological errors are removed. Edges which are adjacent to more than two faces are considered to be erroneous and so are any non-manifold vertices. Inappropriate edges can be found by a single pass through mesh faces and their corners. Each corner remembers a list of its opposite corners (as described in the previous chapter). Therefore, if the length of this list is not equal to zero or one, the corner's opposite edge is not valid. Vertices violating 2-manifoldness can be recognised in a single pass through mesh vertices and its corners. For each vertex, we take a corner which is adjacent to a boundary edge. If the boundary edge does not exist, an arbitrary corner can be chosen. Then we start moving to neighbouring corners of the vertex clockwise or counterclockwise. We continue until we visit all vertex corners or until we encounter the initial

corner. The vertex will not be valid unless we visit all the vertex corners during the pass.

- Subsequently, the face is trimmed and only the facial area is preserved. The algorithm needs to know some information about basic facial landmarks (the nose tip and root, mouth corners and inner eye corners). Therefore, it attempts to detect these points. If the detection is not fully successful, some features (cropping by normals) or subsequent phases (face rotation) are skipped.

- Afterwards, defective vertices and vertices which form intersecting triangles are removed. A vertex is marked as defective if its curvature is odd or if an incident edge is the common edge of two triangles with the too small or too large dihedral angle. After the removal of defective vertices, we need to remove minor components and non-manifold vertices again.

- Subsequently, all holes (including those created in previous steps) are filled. As we will explain later, triangle intersections can originate from the hole filling, that is why the hole filling and more removals of intersecting triangles are performed in a loop.

- Finally, thin-shaped triangles (this also includes zero-volume triangles) are resolved, and the cleaned and hole-free face is rotated with respect to the landmarks computed in a preceding step.

Note that the removal of minor components and the detection of non-manifold vertices or edges is a standard feature of a mesh processing software. Both MeshLab and Geomagic offer tools for resolving these problems.

5.2 Curvature

In order to recognise geometrical errors or to obtain information about aspects of a mesh, we need to describe the local shape of the mesh around a particular vertex. Because of that, we decided to use means of the discrete differential geometry. We use the term *curvature* throughout the algorithm, so let us first define it formally. We are also going to mention methods of its approximation for triangular meshes.

5.2.1 Definition

Curvature is a quantity which measures how bent a geometric object like a curve or a surface is [Pressley, 2005]. Intuitively, the curvature of a circle can be seen as the change in the tangent vector and that is why the curvature of a circle with radius r is simply $\kappa = 2\pi/2\pi r = 1/r$.

Given a plane curve, for each point on this curve, there is a unique circle which best approximates the curve near the point (an osculating circle). It means that the circle has in the particular point of the curve the same tangent direction and also the same first and the second derivatives as the curve. The curvature κ of the curve at the point is then given by the curvature of the osculating circle.

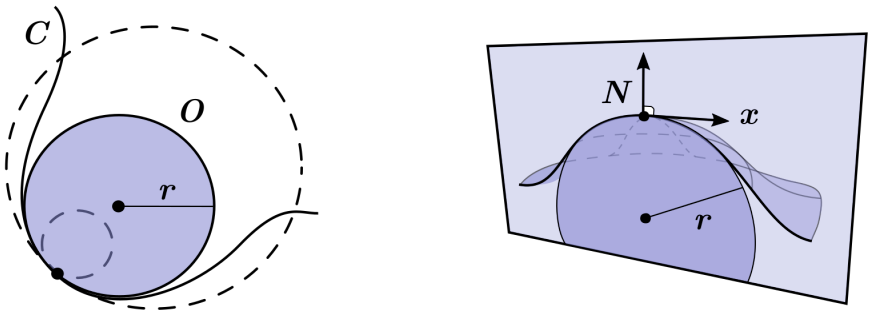

Figure 5.2: Left: *A curve C and the only osculation circle O of a point. Dashed circles are other circles with the same tangent direction as O, though they are not osculating.* Right: *A cut of a surface by a plane defined by a normal vector N and a direction* $\mathbf{x}$*. An osculating circle of the arisen curve is also depicted.*

Let $\mathbb{S}$ be a surface embedded in $\mathbb{R}^3$ and $\mathbf{x} \in \mathbb{R}^2$ be a unit tangent direction at a particular point $\mathbf{P}$ on the surface. This direction and the corresponding normal vector $\mathbf{N}$ uniquely define a plane. The intersection of this plane and the surface $\mathbb{S}$ forms a curve and we call the curvature of this curve the *normal curvature* in the direction $\mathbf{x}$ [Crane et al., 2013]. We denote it by κ_n. Please notice that the normal curvature is a signed quantity, so it can be expressed whether the surface is bent towards or away from the normal.

We can also express the normal curvature in terms of a matrix $\mathbf{II}$ which is known as the *second fundamental form* or the *curvature tensor*:

$$\kappa_n = \mathbf{x}^T \, \mathbf{II} \, \mathbf{x}, \qquad \mathbf{II} = \begin{pmatrix} \frac{\partial n}{\partial u}\mathbf{u} & \frac{\partial n}{\partial v}\mathbf{u} \\ \frac{\partial n}{\partial u}\mathbf{v} & \frac{\partial n}{\partial v}\mathbf{v} \end{pmatrix} = \begin{pmatrix} e & f \\ f & g \end{pmatrix}$$

where $\mathbf{u}$ and $\mathbf{v}$ together form an orthonormal coordinate system in the tangent plane [Rusinkiewicz, 2004]. We can get the derivative of the normal in an arbitrary direction just by multiplying the second fundamental form by a unit tangent vector corresponding to the direction.

The two orthogonal directions $\mathbf{x_1}$ and $\mathbf{x_2}$ which are associated with extreme values of all the normal curvatures are called *principal directions* or *principal vectors* and the corresponding curvatures κ_1 and κ_2 are called *principal curvatures*. The second fundamental form can be diagonalised to obtain principal curvatures and directions.

Based on the two principal curvatures, we also define the *mean curvature*: $\kappa_H = (\kappa_1 + \kappa_2)/2$ which is the arithmetic mean of principal curvatures, the *Gauss curvature*: $\kappa_G = \kappa_1 \kappa_2$ which is the square of the geometric mean and also the *curvedness*: $\kappa_C = \sqrt{(\kappa_1^2 + \kappa_2^2)/2}$ which specifies the amount of the surface curvature (Fig. 5.3).

5.2.2 Computation

There is a popular approach proposed by Meyer, Desbrun, Schröder, and Barr [2003] to calculate the curvature approximation of a discrete surface, especially of a triangular mesh. This method is widely spread and very easy to implement, so

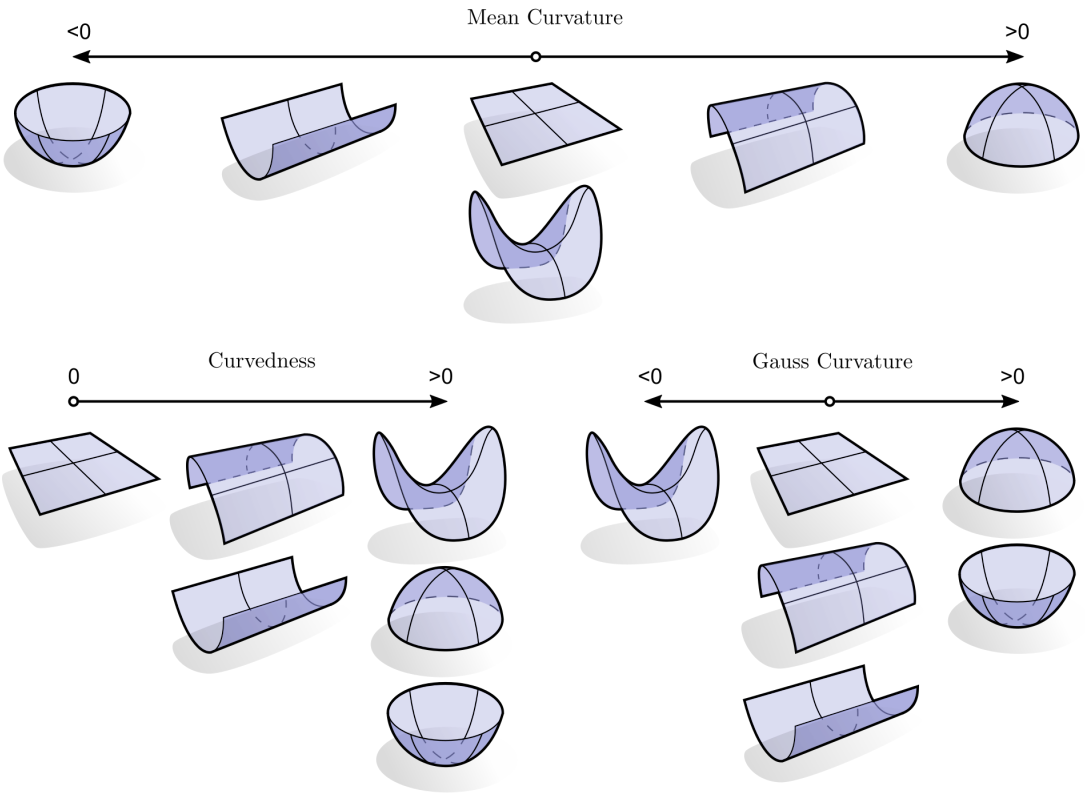

Figure 5.3: Top: *Visualization of the dependence between mean curvature and shape. Positive mean curvature implies concave surface and negative mean curvature indicates convex shape.* Bottom left: *The Dependence between curvedness and shape. Curvedness expresses the magnitude of curvature.* Bottom right: *The dependence between Gauss curvature and shape. Note that Gauss curvature is minimal for saddle points and maximal for local extrema.*

we are going to describe it briefly – just for completeness. Then we will explain why we did not use it in our implementation.

Meyer et al. [2003] state that the computation of mean and Gauss curvatures at a vertex $\mathbf{x}_i$ of a triangular mesh is as difficult as solving these equations for each mesh vertex:

$$\kappa_H(\mathbf{x}_i) = \frac{1}{4A_i} \left\| \sum_{j \in N_1(i)} (\cot \alpha_{ij} + \cot \beta_{ij})(\mathbf{x}_i - \mathbf{x}_j) \right\|,$$

$$\kappa_G(\mathbf{x}_i) = (2\pi - \sum_{j \in F(i)} \theta_j)/A_i$$

where $N_1(i)$ denotes the set of one-ring neighbours of the vertex $\mathbf{x}_i$, $F(i)$ is the set of faces adjacent to the vertex $\mathbf{x}_i$, α, β, θ are angles as shown in Fig. 5.4 (right and middle) and A_i is the surface area of the vertex $\mathbf{x}_i$.

The area can be obtained by summing up areas of corners which are adjacent to the vertex. The corner area of a corner which belongs to a triangle t with an area a_t is, in the case when t is non-obtuse, equal to its corresponding Voronoi area. In the other case, the corner is either obtuse and then its area is a half of a_t or non-obtuse and then the area is just a quarter of a_t (Fig. 5.4 left).

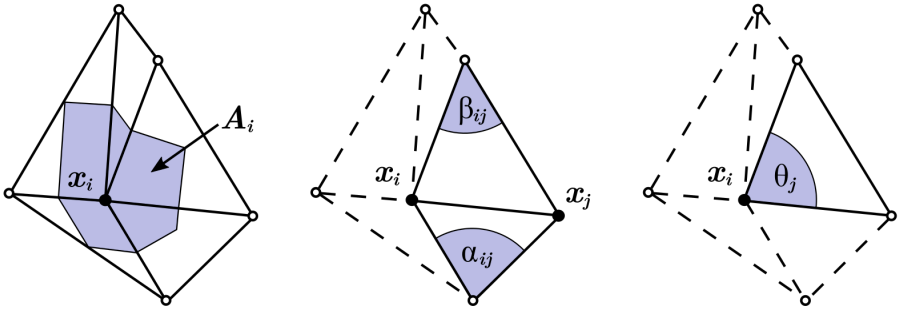

Figure 5.4: Left: *The area of a vertex. All possible triangle variants are depicted.* Middle: *Angles used to compute mean curvature in case of two neighbouring vertices* $\mathbf{x}_i$ *and* $\mathbf{x}_j$. Right: *A corner angle* θ_j *used to calculate Gauss curvature.*

Once mean and Gauss curvatures are gained for each vertex, principal curvatures can be computed using these two equations:

$$\kappa_1(\mathbf{x}_i) = \kappa_H(\mathbf{x}_i) + \sqrt{\kappa_H(\mathbf{x}_i)^2 - \kappa_H(\mathbf{x}_i)},$$
$$\kappa_2(\mathbf{x}_i) = \kappa_H(\mathbf{x}_i) - \sqrt{\kappa_H(\mathbf{x}_i)^2 - \kappa_H(\mathbf{x}_i)}.$$

Also, principal directions can be further obtained.

Although this method gives good results and can approximate continuous surfaces well, there exist degenerate configurations of neighbouring vertices which make the method unstable. Weird results appear while calculating curvatures of vertices with adjacent triangles which are very thin or have a very dissimilar area (Fig. 5.5 left). To avoid the difficulty, we implemented another algorithm developed by Rusinkiewicz [2004].

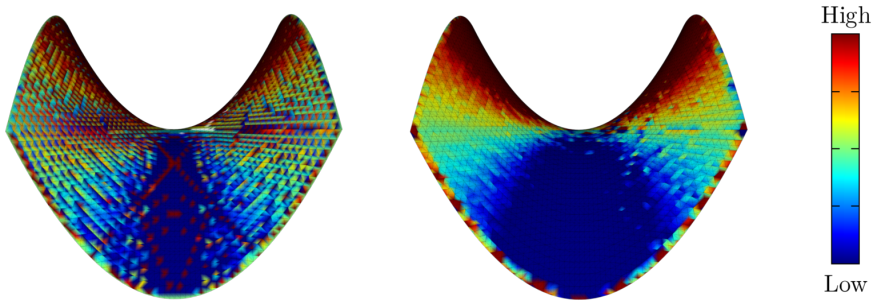

Figure 5.5: *A saddle mesh with a lot of degenerated triangles. Vertices are coloured according to their mean curvature.* Left: *There are curvatures computed by Meyer's method. There are many discrepancies; neighbouring vertices on an almost smooth surface have absolutely different mean curvature.* Right: *The colouring produced by the method proposed by Rusinkiewicz. Mean curvature values are smoothly changing through the whole surface as expected.*

Despite the fact that the equations for **II** hold only for smooth surfaces, Rusinkiewicz suggests solving this set of linear equations in order to get an approximation of the second fundamental form for a single triangle of a triangular mesh:

$$\mathbf{II}\begin{pmatrix}\mathbf{e}_0\cdot\mathbf{u}\\ \mathbf{e}_0\cdot\mathbf{v}\end{pmatrix}=\begin{pmatrix}(\mathbf{n}_2-\mathbf{n}_1)\cdot\mathbf{u}\\ (\mathbf{n}_2-\mathbf{n}_1)\cdot\mathbf{v}\end{pmatrix},$$

$$\mathbf{II}\begin{pmatrix}\mathbf{e}_1\cdot\mathbf{u}\\ \mathbf{e}_1\cdot\mathbf{v}\end{pmatrix}=\begin{pmatrix}(\mathbf{n}_0-\mathbf{n}_2)\cdot\mathbf{u}\\ (\mathbf{n}_0-\mathbf{n}_2)\cdot\mathbf{v}\end{pmatrix},$$

$$\mathbf{II}\begin{pmatrix}\mathbf{e}_2\cdot\mathbf{u}\\ \mathbf{e}_2\cdot\mathbf{v}\end{pmatrix}=\begin{pmatrix}(\mathbf{n}_1-\mathbf{n}_0)\cdot\mathbf{u}\\ (\mathbf{n}_1-\mathbf{n}_0)\cdot\mathbf{v}\end{pmatrix}$$

where $\mathbf{e}_i$ stands for an edge vector, $(\mathbf{u},\mathbf{v})$ is a selected orthonormal coordinate system and $\mathbf{n}_i$ are normals at vertices of the triangle.

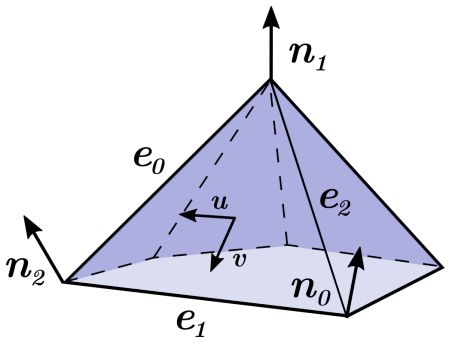

Figure 5.6: *A face with an example orthonormal coordinates* ($\mathbf{u}$, $\mathbf{v}$) *and normals of its vertices.*

If normals along with vertices are not provided, we estimate them by a method proposed by Max [1999]. Let us suppose a vertex $\mathbf{x}_i$ with adjacent faces f_i; the normal is then

$$\mathbf{n}_i=\sum_{i,j,k\in f_i}\frac{(\mathbf{x}_i-\mathbf{x}_j)\times(\mathbf{x}_i-\mathbf{x}_k)}{\|\mathbf{x}_i-\mathbf{x}_j\|^2\,\|\mathbf{x}_i-\mathbf{x}_j\|^2}.$$

Thus the normals can be efficiently computed by a single pass through all faces.

Once the equations proposed by Rusinkiewicz [2004] are solved for each triangle, we can compute principal curvatures at each vertex. To do this, the presented method averages contributions from adjacent triangles. Let's assume that every vertex has its orthonormal coordinate system $(\mathbf{u}_v,\mathbf{v}_v)$ which is defined in the plane perpendicular to its normal. At first, the curvature tensor of an adjacent triangle with a coordinate system $(\mathbf{u}_t,\mathbf{v}_t)$ is rotated to make both orthonormal systems coplanar. Denote the rotated system $(\mathbf{u}'_t,\mathbf{v}'_t)$. Then elements of the second fundamental form of the vertex can be expressed in the rotated coordinate system:

$$e_p=\mathbf{u}_p^T\,\mathbf{II}\,\mathbf{u}_p=\begin{pmatrix}\mathbf{u}_p\cdot\mathbf{u}_{f'}\\ \mathbf{u}_p\cdot\mathbf{v}_{f'}\end{pmatrix}^T\mathbf{II}\begin{pmatrix}\mathbf{u}_p\cdot\mathbf{u}_{f'}\\ \mathbf{u}_p\cdot\mathbf{v}_{f'}\end{pmatrix}$$

and similarly $f_p=\mathbf{u}_p^T\,\mathbf{II}\,\mathbf{v}_p$ and $g_p=\mathbf{v}_p^T\,\mathbf{II}\,\mathbf{v}_p$. This new tensor weighted by the triangle's area (the area computation described earlier can be used) is then added to the curvature tensor of the vertex.

The whole algorithm which we use to compute vertices curvature is summarised in Fig. 5.7.

We can now calculate curvature in any vertex of the mesh, but the curvature is computed only from the one-ring neighbourhood of the vertex. Using the same principle of rotation and summation of curvature tensors as seen while computing vertex curvatures, we can smooth curvatures just by looking at a broader neighbourhood and by weighting tensors (Fig. 5.8). To assign weights, we use a Gaussian function $exp(-\frac{l^2d^2}{2c^4})$ where l denotes the mean edge length of the mesh and d is the Euclidean distance between two vertices whose curvature tensors are going to be combined and c is the scale factor (see Appx. A, `scale` parameter).

```
// prepare per-vertex normals
compute_vertex_normals()

// contruct an initial coordinate system
foreach vertex in vertices:
   vertex.u = (an_adjacent_edge × vertex.normal()).normalized()
   vertex.v = vertex.normal() × vertex.u

// compute curvature tensor for each triangle
foreach triangle in triangles:
   edges = get_edge_vectors(triangle)
   normal_diff = get_normal_differences(triangle)
   // define local coordinate system
   u = edges[0].normalized()
   v = ((edges[0] × edges[1]) × u).normalized()

   // compute second fundamental form using least squares
   tensor = solve_second_fundamental_form(edges, normal_diff, u, v)
   foreach corner in triangle:
      vertex = corner.vertex
      weight = corner.area()
      vertex_tensor = project_from_old_to_new_coords(tensor, u, v,
                                                     vertex.u, vertex.v)
      vertex.tensor += weight * vertex_tensor

// normalize tensors and extract principal curvatures
foreach vertex in vertices:
   vertex.tensor /= vertex.area()
   vertex.curvature = diagonalize(vertex.tensor)
```

Figure 5.7: *The algorithm for curvature computation.*

5.3 Point Detection

One of the primary purposes of our algorithm is the mesh trimming. Facial scanners usually capture the face and surroundings with ears, neck and hair. However, these marginal parts are often full of errors and defects and are not desired for further face processing. The trimming algorithm can be used to obtain only the interesting part of a facial scan – the actual face.

However, the trimming is impossible without basic information about the face, such as position, orientation and size. Because we want the process to be done automatically without user assistance, we are introducing a technique of detecting interesting points in a mesh. Once we can find points like a nose tip, a nose root, etc., we can get dimensions, the position and the orientation of an arbitrary face. On top of that, acquiring these points by an automatic process can provide useful hints for methods which need an initial rough mesh alignment (e.g. DCA – Hutton et al. [2001]) and which are often necessary for anthropological research.

We detect the facial landmarks just by exploring the shape of a local neighbourhood of each mesh vertex. Thus, we do not rely on the texture information or the mesh orientation, dimensions or completeness.

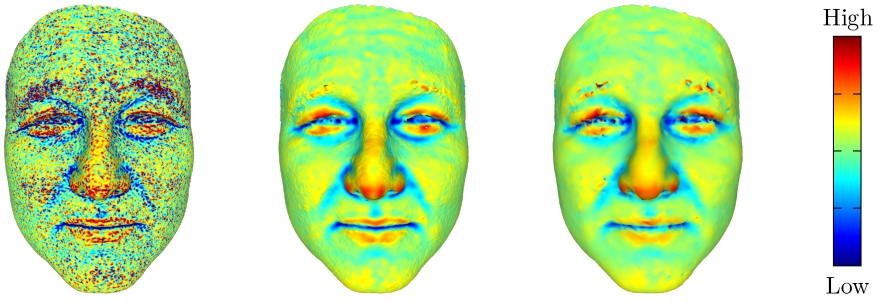

Figure 5.8: Left: *A mesh with approx. 300k triangles. Vertices are coloured with respect to the mean curvature.* Middle: *The same mesh as on the left, but vertices are coloured with respect to the smoothed mean curvature.* Right: *The same mesh after a massive face reduction (approx. 10k faces). Note that there is almost no difference between smoothed variants of original and reduced meshes.*

In the several following chapters, we will describe the detection algorithm; we will discuss which points we wish to find and which features and methods are used in our implementation.

5.3.1 Algorithm outline

We are now going to explain some basic concepts and terms of the machine learning. One of the machine learning tasks is called *supervised learning* – the computer is provided with a set of input-output pairs, and the task is to develop an algorithm which can approximate the mapping function just by observing the pairs. We call these pairs *examples* and the analysis of examples through the machine learning algorithm is called *training*. The set of examples used for the training and also for the verification of the algorithm is called *development set*. We call the input part of an example *feature vector* and the other (output) part *label* or *class* [Alpaydin, 2010].

Because we would like to detect landmarks by exploring the shape of vertex neighbourhoods, we use means of discrete differential geometry, especially the mean and Gauss curvatures. For each mesh vertex, we form a representation (a feature vector) of curvatures of its local neighbourhood. Then we use a machine learning method to train a classifier which can give us the probability that an arbitrary vertex is an interesting point of a particular landmark type. After the classification, we filter the output and mark some vertices as the landmarks with respect to the probabilities given by the classifier.

5.3.2 Desired points and development data

We use a set of six basic landmarks — nose tip, nose root, eye corners and mouth corners (Fig. 5.9 left). These points are distinguishable through curvatures and are also located at very specific facial areas. Thus, they should not be mistaken for other facial points. The set of six landmarks is sufficient to determine the face orientation and dimensions. Moreover, it forms a shape which can be validated due to assumptions about face proportions. This ability is important because, as

we will explain in the next chapter, our method produces many candidates for each landmark type and this helps us to filter them out.

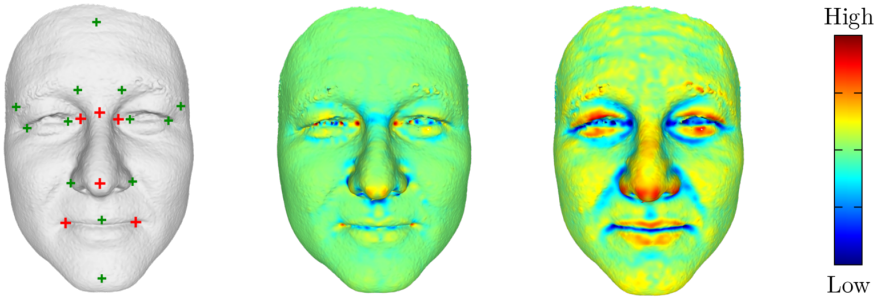

Figure 5.9: Left: *Manually marked facial landmarks. We would like to detect the red points; the green points are used as negative labels in classifier training.* Middle: *Vertices are coloured with respect to thier smoothed Gauss curvature.* Right: *A visualisation of the smoothed mean curvature.*

Once we have thought over which landmarks we want to detect, we should think about points we would like to include into our development set. The set should, of course, contain all important points, but also points which have similar properties and could have possibly been misclassified. Having various development set in that way forces machine learning algorithms to sense the right features. Thus, we include some outstanding facial points (Fig. 5.9 left) and also some random neighbours of the important landmarks into the development set. This construction of the development set gives us over 35 thousand examples from just 194 faces.

5.3.3 Feature vector design

We can summarise requirements on the feature vector:

- It should **describe the local shape** of a vertex neighbourhood.
- We cannot make any assumptions about the facial area orientation or size, so our point-detection algorithm should be **scale and rotation invariant**.
- Also, the mesh **density should not be a property the detection relies on** because different scanners can produce meshes of different density. Besides, a mesh might not be triangulated regularly.
- Typically, high-quality meshes produced by contemporary scanners contain hundreds of thousands of vertices and we need to calculate a feature vector for each vertex. Thus, the feature vector **computation should not take a long time**.

To fulfil these requirements, we propose a feature vector which consists of two distinct parts which vary only in the usage of the smoothed mean curvature and the smoothed Gauss curvature. The first part describes three histograms of the mean curvature of unequally-sized local neighbourhoods of a vertex. There is

also the value of the mean curvature of the vertex at the end of this feature vector part. Let d be the radius of a neighbourhood. We propose three different neighbourhoods with a view of adding geometric information to the feature vector which cannot be directly described by a single histogram of curvatures. We set radii of the two smaller histograms to $0.65\,d$ and $0.3\,d$. These values were chosen in order to divide d roughly equally.

As defined, the feature vector is not scale-invariant, because the absolute size of neighbourhoods must be specified. We define it by means of mesh units. The mesh unit-to-millimetre correspondence was described earlier in Chapter 3.2. We suppose that one mesh unit is equal to one millimetre further in this chapter.

For the computation of a vertex feature vector, we need to find all vertices which lie within the distance d from the vertex. As the computation of a feature vector has to be fast, d needs to be limited, because its large value leads to a quadratic algorithm which is for common data with 200k vertices unacceptable. Setting d too small causes the local neighbourhood to be tiny so that the detector may overlook some information about the surface shape. We use $d = 10$ in our algorithm because the time of the calculation is still reasonable and a patch with the diameter of 2 centimetres still describes essential face details. We use a simple depth-first search to find the vertex neighbourhood. We found out that using the k-d tree data structure does not bring any advantage or time savings.

To make the method invariant with respect to the mesh density, we weight curvatures of vertices in a local neighbourhood by areas of the vertices. The computation of the vertex area was described earlier in Chapter 5.2.2. Each histogram must be, of course, normalised by the sum of vertex weights. Please notice that the detector cannot be used to find points in sparse meshes or meshes with a low resolution because there can be few vertices contributing to histograms and the resulting shape of a histogram might be coarse.

We mentioned histograms in previous paragraphs and we are now going to describe them. Histograms are defined just by three parameters – lower and upper bounds and the bin size. Smoothed mean and Gauss curvatures of our development data were analysed and we decided to set the lower and upper bounds according to the average values of 0.1% and 99.9% quantiles respectively. Fig. 5.10 shows the distribution of desired quantile values for our data. For Gauss curvatures, the lower bound has the mean equal to $-0.0245\,\mathrm{mm}^{-2}$ and the standard deviation of $0.0044\,\mathrm{mm}^{-2}$, the upper bound $0.1017\,\mathrm{mm}^{-2}$ and $0.0862\,\mathrm{mm}^{-2}$. In the case of mean curvatures, the lower bound has the average value of $-0.2850\,\mathrm{mm}^{-1}$ and the standard deviation $0.0681\,\mathrm{mm}^{-1}$, the upper bound has the mean $0.2421\,\mathrm{mm}^{-1}$ and the deviation $0.0971\,\mathrm{mm}^{-1}$. Based on these values, we set ranges $(-0.025, 0.05)\,\mathrm{mm}^{-2}$ for histograms of Gauss curvature and $(-0.25, 0.2)\,\mathrm{mm}^{-1}$ for histograms of mean curvature.

The number of histogram bins should be much lower than expected number of neighbours in the smallest histogram because we would like to populate all the bins. Moreover, the number of bins has a significant impact on the feature vector length. Thus using a large number of bins can make the usage of machine learning methods impossible (Curse of dimensionality – Donoho [2000]). We tried to set it to values 8, 16 and 32. We use 16 in our implementation, but the decision on which size to use is tightly related to the performance of our classifier, so we postpone the comparison of different histogram sizes to the next chapter.

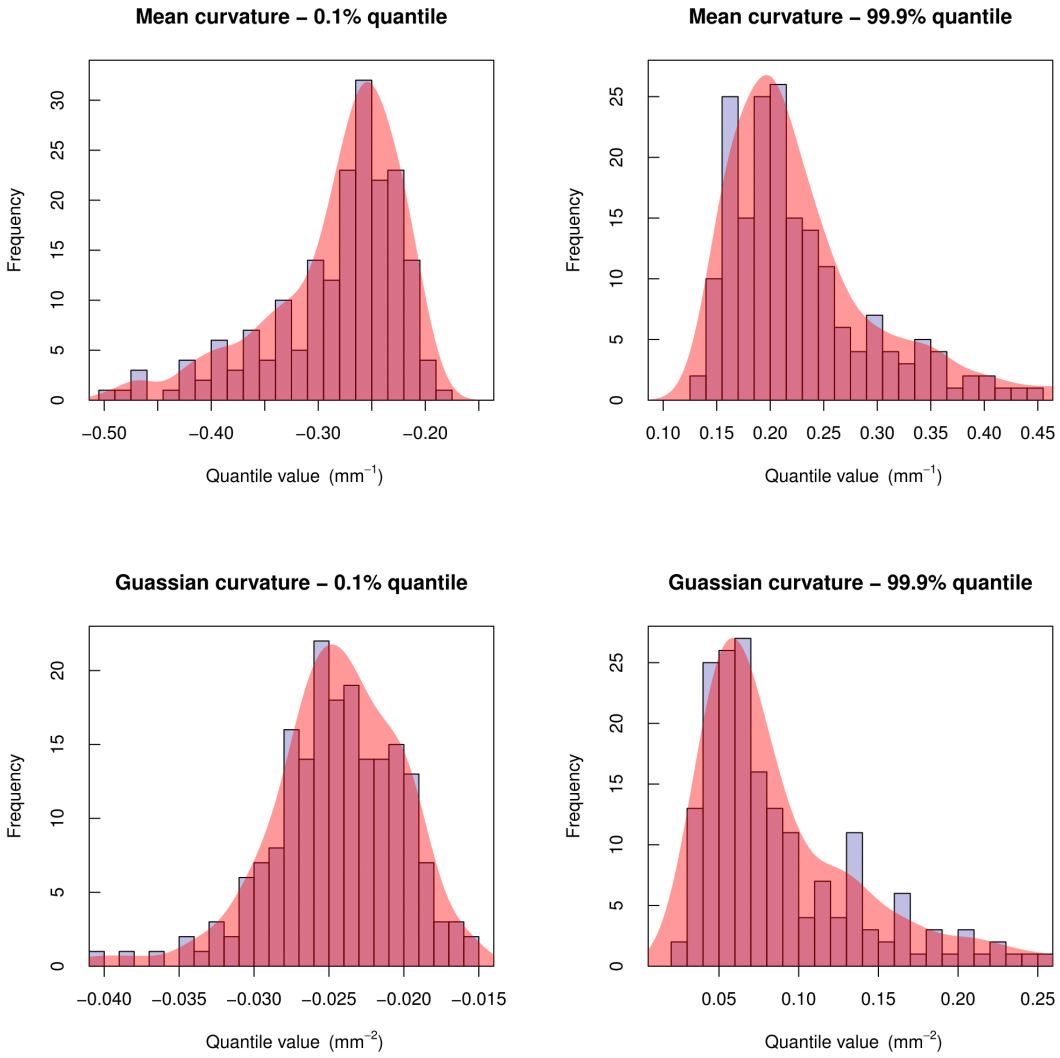

Figure 5.10: *Distributions of quantiles of mean and Gauss curvatures. The quantiles were obtained from our development data.*

5.3.4 Comparison of classifiers

We decided to train binary classifiers for each class of landmarks – nose tip, nose root, eye corners and mouth corners. This approach allows us to divide training examples into two classes – positives and negatives. It means that we suppose a single original class to be positive and all other original classes to be negative.

Now, we are going to compare several classifiers obtained by different machine learning algorithms. An explanation of the algorithms is not essential for this work. We are going to mention parameters we found to be optimal for the selected classifier, so that it will be possible to reproduce our method. The implementation of the machine learning method we have been using in this work is described in the technical documentation (Chapter 6).

In order to compare our classifiers, we measured their performance, especially *accuracy* and *F-score*. The accuracy is defined as the portion of correctly classified examples. Let us define *precision* as the number of correctly determined positives divided by the number of all positives returned by the classifier and *recall* as the

number of correctly determined positives divided by the number of all relevant samples. F-score is the harmonic mean of precision and recall. We measured the performance just on the mouth corner landmark class because it is the most challenging type of landmark and we use the same *hyper-parameters* to train classifiers for other classes. The following table 5.1 contains the average accuracy and the average F-score of Random Forests (RF), Support Vector Machine (SVM) and Neural Network (NN) models obtained by 7-fold cross-validation. We used the development data described in the previous chapter 5.3.2 for training and evaluation.

Table 5.1: *Summary of performance of three different classifiers during mouth corner detection. We trained three variants of the classifiers using different histogram sizes. There are summarised accuracies and F1-scores obtained by 7-fold cross-validation (that is why there are also mean values and deviations).*

Method	Histogram Size	Accuracy Mean	F1-score Mean	Accuracy Std. Dev.	F1-score Std. Dev.
RF	8	0.94355	0.79978	0.00322	0.01558
	16	0.94288	0.79709	0.00263	0.01272
	32	0.94150	0.79062	0.00262	0.01204
SVM	8	0.94368	0.80169	0.00462	0.01990
	16	0.94275	0.79786	0.00408	0.01423
	32	0.94179	0.79169	0.00533	0.01990
NN	8	0.94125	0.80038	0.00519	0.01801
	16	0.94328	0.79773	0.00443	0.01393
	32	0.94275	0.79413	0.00272	0.01778

RF – random forests, SVM – support vector machine, NN – neural network

According to the table, performance of all classifiers is very similar. Histograms of 16 bins have lower standard deviations of both accuracy and F1-score. So the classifiers are not so sensitive to the composition of training sets while using the size of histograms and that is why we decided to set the histogram size to 16. Because the comparison of accuracy and F1-score measures is done just on the set of development examples, we also visually compared how the classifiers work on the whole meshes (Fig. 5.11).

We found out that the SVM tends to produce many false positives. Event though the RF model produces few false positives, the returned probabilities are quite low even in the right spots. Moreover, RF does not work well with meshes with massively reduced vertices. The best of our models was the NN model. It does not produce a lot of false positives and it works correctly even with the reduced meshes. Now, we are going to describe the model in more detail.

We do not include the explanation of neural networks in this work. Please see Goodfellow et al. [2016] for information about all terms we use in the following paragraphs.

Our neural network model is straightforward. It has two *dense layers* with *ReLu activations*. The first layer has 512 *neurons* and is shared by all histograms

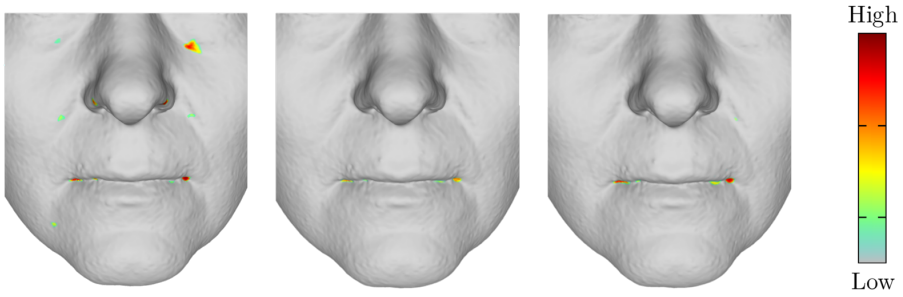

Figure 5.11: *The lower part of the face with visualisation of mouth corner predictions.* Left: *The SVM results. There are noticeable false positives (e.g. under the right eye)* Middle: *The result of the RF model. There are no false positives, but the probability in the right spots is not so high (green colour instead of red)* Right: *Output produced by the NN model. We can see high probabilities in the right spots.*

of the feature vector. It means that each neuron of this layer has 16 inputs. The output (for all histograms) of the first layer is concatenated and input values of mean and Gauss curvatures are also added. That intermediate vector of the size $6 * 512 + 2$ is connected to the second dense layer with 1024 neurons. The result of the second layer produces predictions of the model. During the learning process, we used *dropout* with the 50% probability and *learning rate decay* with the starting value 0.001 and the final value 0.0001.

Users can specify their own sets of models (see Appx. A.3), but the models have to meet requirements which are described in the technical documentation (Chapter 6).

For illustration, the Fig. 5.12 shows detected areas for eye corners, nose tip and nose root. Note that the mesh was not used during the training process.

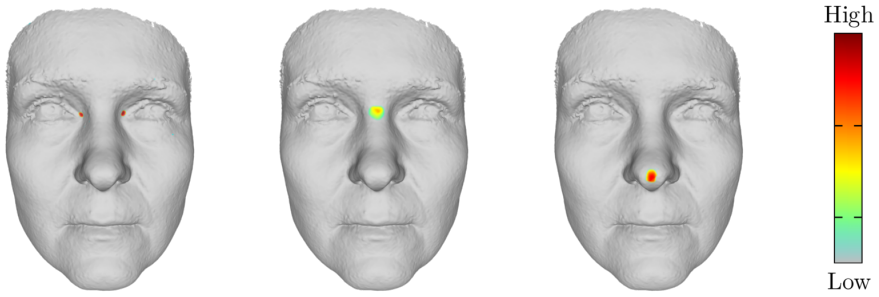

Figure 5.12: *Visualization of probabilities returned by NN models.* Left: *Eye corners.* Middle: *Nose root.* Right: *Nose tip.*

5.3.5 Filtering

These four trained classifiers give us four different probabilities for each vertex. We need to select only a limited number of vertices for each landmark type – only

one vertex as a nose tip, nose root and exactly two vertices as mouth corners and eye corners. Choosing a vertex with the greatest probability as the desired landmark does not work well, therefore we group vertices with the probability higher than 0.5 into connectivity components and we assign weights to these components. The weights are summations of probabilities of contained vertices for the nose tip class and the geometric mean of probabilities of contained vertices for all other landmark types. Then we choose a single representative for each component – geometric median [Eftelioglu, 2015] weighted by the probabilities. It is difficult to efficiently find the weighted geometric median, but there are available some algorithms [Vardi and Zhang, 2000]. However, we do not expect the components to be large, so we use just a trivial algorithm in our implementation. We compute a cost for each vertex of the component and then we mark the vertex with the minimal cost as the geometric median.

Now, we are going to introduce a filtering method to choose a single set of six valid facial landmarks. We were given a set of 194 facial models and both sexes were included – 79 males and 115 females. The sample consists of volunteers from the Czech population, with age ranging from 20 to 82 years. Thus, the following parameter values are not general and should be changed for very different populations (see Appx. A.3). We manually extracted some proportions and distances in order to set the constraints, please see Table 5.2.

Table 5.2: *Summary of some facial proportions and distances. We obtained these values from our development data – 194 facial models of Czech population.*

Constrain	$\boldsymbol{\mu}$	$\boldsymbol{\sigma}$	**Min**	**Max**
Distance between eye corners (EE)	24.302	4.366	16.547	37.388
Nose tip to nose root distance (NR)	48.322	3.990	37.099	57.694
Distance between mouth corners (MM)	47.350	4.941	36.343	60.383
Nose tip to eye corner over NR	1.150	0.070	1.006	1.374
Nose root to eye corner over EE	0.786	0.080	0.604	1.026
Nose root to mouth corner over NR	1.712	0.100	1.475	2.032
Nose tip to mouth corner over MM	1.170	0.100	0.908	1.425

number units of non-rational quantities – mm

These values of the mean and the standard deviation together with the three-sigma rule constitute all criteria we use to filter out possible sets of six landmarks. The three-sigma rule states that for normally distributed variables, at least 99.7% of cases should fall within the interval $(\mu - 3\,\sigma, \mu + 3\,\sigma)$. If there are still present more sets after the filtering, we order them by the weight of the nose component, then by the sum of weights of the eye components and then by the sum of weights of the mouth components. Then we take the set of landmarks which is the first in the list of possible sets sorted in descending order.

5.3.6 Results

We randomly selected 20 % of our development meshes and we used them as the test dataset. Other scans were used to train classifiers. A set of six landmarks was successfully found in all test meshes.

The table 5.3 summarises the accuracy of our algorithm. The detector is most accurate when predicting the nose tip location. The detection of eye corners and mouth corners has larger standard deviation. The inaccuracies are caused by an inability to recognise the right spot using just a local neighbourhood on the one hand, and by the filtering on the other. Even if the classifier marks the correct location of a mouth or an eye corner, there are usually more candidate locations around these places (wrinkles around the mouth, caruncles). However, the filtering has to choose only one set of landmarks. The selection of the right set is complicated and is the main reason for worse results in the detection of eye and mouth corners.

Table 5.3: *Results of our algorithm – summarising precision of landmark detection (regarding the distance to manually placed landmarks) on a set of 39 randomly selected scans (the remaining 155 meshes were used for the training).*

Landmark type	Mean error	Std. dev. error	Max. error
Nose tip	1.999	1.241	4.468
Nose root	1.953	1.541	7.376
Inner eye corners	1.577	2.934	7.717
Mouth corners	2.000	2.625	11.455

number units – mm

Despite the fact that our algorithm is not so precise compared to other methods [Guo et al., 2013], it is sufficient for our purposes – obtaining a rough face orientation – and works on meshes with arbitrary dimensions or orientation.

5.4 Face Cropping

Now, we are going to discuss our technique of the facial area extraction. Raw facial scans typically contain hair, neck and ears, but our goal is to produce meshes without these margins because they usually contain many errors and because they are not desired for the purposes of anthropological research.

If the trimming step or just the complex variant (explained in the following chapter) are not desired, one can skip it by changing a flag in the configuration file (see Appx. A).

5.4.1 Vertex removal

Once we are able to find out where the important facial landmarks are, we can also trim the mesh. However, as we explained, the set of landmarks might not be detected, or only the nose tip may be detected. That is why we propose a technique of facial scans cropping which includes two variants. The first one depends only on the nose tip position and the other one uses all six points to get the facial area. If all six landmarks are found, we use both variants together.

In order to trim the face, the first method takes into account two criteria:

- It removes all vertices whose distances from the nose tip are greater than a specified threshold. This threshold can be changed in the configuration

file (see Appx. A.4). By default, it is set to 130 millimetres. The default value is adequately distant from the nose tip, so it removes obvious outliers and usually even ears (Fig. 5.13).

- Since we want to get the facial area and remove hair, the second criterion is based on the vertex curvedness. The threshold for curvedness specifies the amount of the surface curvature. Therefore, we remove all vertices whose distance from the nose tip is greater than 100 millimetres (Fig. 5.13) while having curvedness greater than $0.08\,\mathrm{mm}^{-1}$. The curvedness threshold is based on our experience and can separate the facial area from ears and hair very well. However, it can be changed in the configuration file (Appx. A.4).

Reducing the distance thresholds should be considered while cropping children's faces. We recommend to reduce it by one or two centimetres.

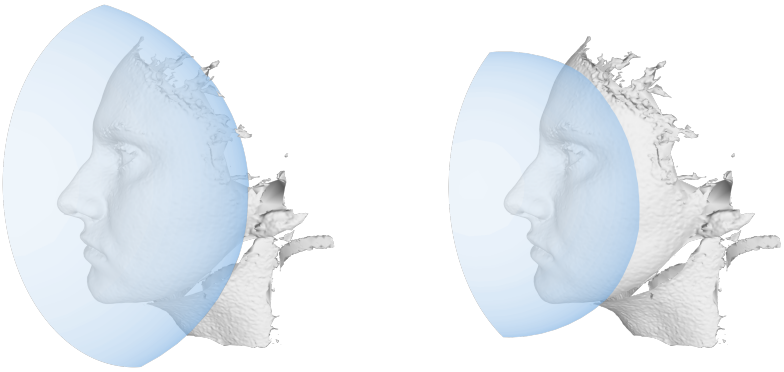

Figure 5.13: Left: *A mesh with a blue sphere with the radius of 130 mm which is centred in the nose tip. The radius should be roughly equal to the distance from the nose tip to ears.* Right: *The same mesh as on the left but with a blue sphere with the radius of 100 mm.*

The second variant utilises the knowledge about all six landmarks and can also remove unwanted areas which are almost flat (e.g. neck). First, we calculate the direction to which the face is heading, the face centre and the vertical direction. Because the point detection may not always be exact, we have to define the two directions and the centre with caution. The following pseudocode describes the computation of these three things:

```
eyes_mid = (left_eye + right_eye) / 2
lips_mid = (left_lips + right_lips) / 2

face_centre = (lips_mid + eyes_mid) / 2

basic = (lips_mid - eyes_mid).normalized()
n = ((eyes_mid - right_lips) × (left_lips - right_lips)).normalized()
exact = (nose_tip - eyes_mid - n * (n · (nose_tip - eyes_mid)).normalized()

vertical_dir = (basic + exact) / 2
sight_dir = (nose_tip - eyes_mid - basic * ((nose_tip - eyes_mid) · basic))
if (sight_dir · (nose_pos - face_center) < 0) sight_dir = -sight_dir
```

Now, we are going to define two more cropping criteria which are based on the computed directions and the face centre:

- The first criterion uses information about vertex normals. For each vertex, we calculate the magnitude of the angle between the direction of sight and the normal of the particular vertex. Then it is decided whether to remove the vertex or not. The decision is based on the magnitude, and users can specify two thresholds – one for vertices which are situated in the vertical direction (the neck threshold) and another for vertices which are situated in the direction perpendicular to the vertical direction and the sight direction (the ears threshold). These two thresholds are smoothly mingled. The criterion is summarised in this pseudocode:

```
vertex_dir = vertex - face_centre
projection = (vertex_dir - sight_dir * (vertex_dir · sight_dir))
projection.normalize()

weight = projection · vertical_dir
angle = angle_between(sight_dir, vertex.normal())
ratio = ears_threshold / neck_threshold
return ((1 + (ratio - 1) * weight) * angle > ears_threshold)
```

  We have chosen values 75 degrees and 125 degrees for neck and ears threshold, respectively. It gives us good results and it does not require any additional changes. However, it can also be changed in the configuration file (Appx. A.4).

- The second criterion is similar to the first, but we do not check normals. Instead, we check the angle between the vector oriented from the face centre to the vertex and a plane defined by the direction of sight and the face centre. And as seen before, two thresholds have to be provided. We describe the whole check by this pseudocode:

```
vertex_dir = vertex - face_centre
projection = (vertex_dir - sight_dir * (vertex_dir · sight_dir))
projection.normalize()

weight = max(0, (projection · vertical_dir) ^ 3)
angle = angle_between(sight_dir, vertex_dir) - PI / 2
ratio = ears_threshold / neck_threshold
return ((1 + (ratio - 1) * weight) * angle > ears_threshold)
```

  Thresholds were set with respect to the area we want to preserve. The default values are 10 degrees and 45 degrees and can be changed in the configuration file (Appx. A.4).

Both checks are performed only on vertices which are more distant from the nose tip than the nose tip-to-nose root distance multiplied by the average ratio of the nose tip-to-chin distance to nose tip-to-nose root distance. The average ratio that was obtained from our development meshes is 1.398 with standard deviation 0.085.

5.4.2 Boundary smoothing

Since the mesh trimming method introduced in the previous chapter produces jagged boundaries with numerous protrusions (Fig. 5.14), we need to smooth the

mesh boundary. To do this, we use the method that is going to be described in the following paragraphs.

Let d be a distance called *protrusion range*. We start removing boundary vertices v which together with other boundary vertices v_1, v_2 form an outer angle greater than a given threshold. Let us call the threshold *wide protrusion angle*. The vertices v_1 and v_2 are selected with respect to the distance d from the vertex v. Specifically, the part of the boundary between v_1 and v and also v_2 and v is roughly of length d. Moreover, v_1 lies on the other side than v_2. When there are no vertices v which should be removed, we reduce the distance d in half and change the threshold. The removal step is repeated and we continue the decreasing until d is approximately equal to the mesh mean edge length. The change of the threshold is proportional to the reduction of d and reaches value of *tiny protrusion angle* at the end.

The starting and ending thresholds – wide protrusion angle and tiny protrusion angle – can be specified in the configuration file. By default, they are set to 200 degrees and 270 degrees, respectively. The protrusion range can be also modified. The default value is equal to 6 millimetres, so 12 mm neighbourhood is considered at the beginning.

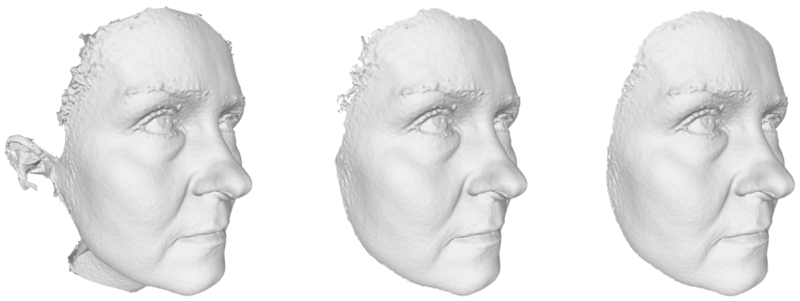

Figure 5.14: Left: *A raw facial scan.* Middle: *The same mesh as on the left, but after applying cropping criteria.* Right: *The cropped mesh as in the middle after the boundary smoothing.*

5.4.3 Results

The cropping strongly depends on the landmark detection, but if the important points are detected at the correct places, it works. We tested the algorithm on the entire development set of meshes. We obtained good results, but a problem occured when cropping faces of overweight people. The difficulty is related to unclear boundary between the neck and the face. Thus, the criterion which uses information about vertex normals to cut off the neck is almost useless.

The task of the facial area trimming is very specific, so we cannot conduct a comparison to an existing algorithm. Instead, we are going to mention features of MeshLab and Geomagic which can be used to crop facial meshes.

MeshLab contains only two features for a large-scale vertex selection – *Select vertices by plane* and *Select vertices in a rectangular region*. These tools are not very powerful and trimming of a face mesh is quite difficult.

Unlike MeshLab, Geomagic offers a wide range of vertex-selection tools: *Lasso selection*, *Rectangle selection*, *Ellipsis selection* and many others. There are also tools called *Trim with curve* (smoothing of the curve can be set as well) and *Trim with plane* which are very suitable for the facial area trimming. Using this, the facial area can be trimmed within a minute, but it still has to be done manually.

5.5 Defects Removal

The detection of intersecting triangles and defective vertices will be discussed in this chapter. First, we are going to explain our implementation of the triangle-to-triangle intersection test. Afterwards, we are going to define defective vertices and propose a method for blobs and spikes removal.

5.5.1 Intersecting triangles

As we decided to remove all intersecting triangles, we need to detect them first. A trivial approach would check every pair of mesh triangles for an intersection. This approach is, of course, optimal in general, because every pair of faces can be intersecting. Regardless of this fact, usual meshes have much fewer self-intersections, and thus we can use a faster algorithm.

The idea of the faster algorithm is straightforward. Initially, we subdivide a mesh to obtain small sets of nearby triangles. Then we check every pair of triangles of these sets for an intersection. Note that the check can be done in parallel. Initially, all faces of the mesh form a single set. Then the set is divided into two sets by a division plane which is perpendicular to an axis of the coordinate system and which goes through the middle of the set with respect to the axis. If a triangle interferes with both new sets, we duplicate it and add to both halves. The subdivision is made recursively. The recursion is stopped if the size of a set of triangles is less than n or if it is impossible to subdivide the set any further. We choose $n = 100$ which gives reasonable computation times.

The implementation of the triangle-to-triangle intersection itself is based on the Method of Separating Axes by Eberly [2008]. A fundamental observation states that two planar convex polygons do not intersect if there exists a line for which intervals of projection of the two objects onto that line do not intersect. The line is called separating axis. Eberly claims that it is sufficient to consider only lines perpendicular to polygon edges as possible separating axes. However, this check works for two-dimensional objects. To find out an intersection of two three-dimensional triangles, we need to check if planes defined by that triangles separate them. If none of the planes separates the triangles, then there are two possibilities for the remaining potential separating planes or axes. The first case is that the triangles are parallel (and should be also coplanar) and then the two-dimensional test can be used. The second case is that the triangles are not parallel and both triangles are split by the plane of each other. In that case, the remaining separating planes are defined by directions which are given by cross products of edges of one triangle and edges of the other triangle.

5.5.2 Blobs and spikes

The definition of blobs and spikes (Fig. 3.3) is difficult because we cannot make a strict boundary between geometrical objects which should and which should not be considered as spikes or blobs. So it is not apparent which objects should be marked as defective. As the human face is smooth, we can make basic assumptions about curvatures and use them to set thresholds which, at least partly, reveal blobs and spikes.

Now, we are going to describe some measurements which help us to detect vertices with odd curvatures. We call them *outliers*. It has turned out that outliers removal also removes most of the blobs and spikes. We have already discussed 0.1% and 99.9% quantiles of smoothed mean and Gauss curvatures. However, setting a threshold for more extremal quantiles is hard due to the diverse curvatures of outliers.

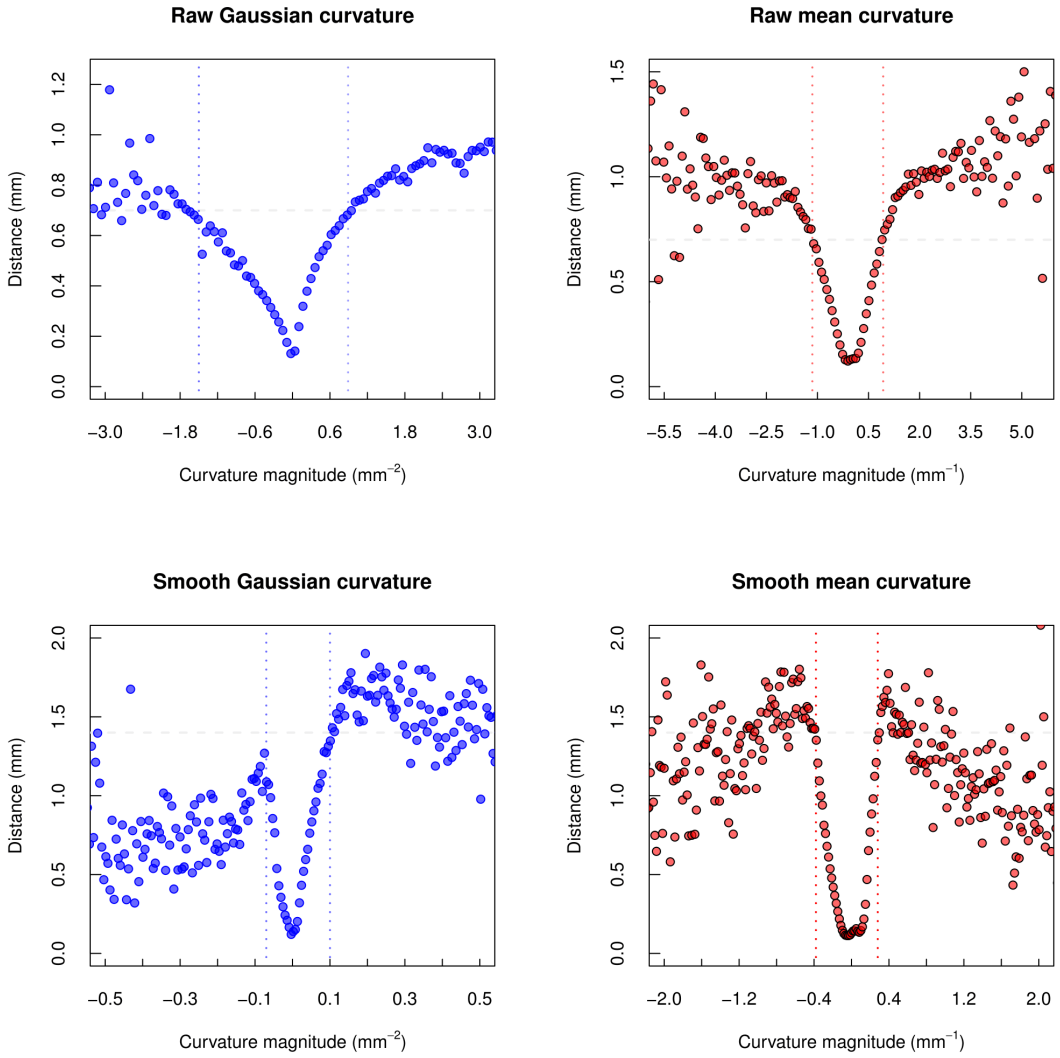

Figure 5.15: *Scatter plots of the curvature-distance dependence. For all vertices of all our development meshes, we determined the distance to a smooth surface obtained by the Poisson Surface Reconstruction algorithm. The points in graphs show the average distance of vertices with very similar curvature.*

Now, we are going to measure the distance of each vertex to a surface without geometrical errors for each mesh from the development set. Although we do not have these surfaces available, we used MeshLab to perform the *Poisson Surface Reconstruction algorithm* (PSR) and to create new smooth meshes. Parameters of the reconstruction were set in order to ignore spikes. However, due to this setup unrealistically smooth facial meshes were produced. Moreover, all vertices of new meshes were created and no original vertices were preserved. These are the reasons why PSR cannot be used to obtain a clean mesh for the purposes of the anthropological research. Note that the distance measurement is just hint and should not be taken as the only decision criterion, because there still might be spikes with low distance to an optimal surface and vice versa. We computed curvatures of all vertices of all available development data and divided them into bins. The Fig. 5.15 shows the dependency of vertex distances on vertex curvatures. The points in the scatter plots correspond to the average distance of all vertices in a particular bin to the smoothed surface.

The size of bins is not so important, because the trend of the growing mean distance with more extremal curvatures remains. There is an obvious V shape in all graphs. The grey horizontal line marks the 1.4-millimetre and 0.7-millimetre distance for smooth and raw curvatures, respectively. We set limits for curvatures (depicted as vertical lines) with respect to these distances and by observing which vertices would be removed on some development meshes. The bounds are summarised in Table 5.4 (these bounds can be changed – see Appx. A.5).

Table 5.4: *Curvature bounds which we use to detect blobs and spikes. These values were chosen with respect to curvature-distance plots (Fig. 5.15) and by studying the effect on cleaning of development meshes.*

Curvature type	**Lower bound**	**Upper bound**
Raw mean curvature	-1.15	0.93
Smooth mean curvature	-0.38	0.28
Raw Gaussian curvature	-1.50	0.89
Smooth Gaussian curvature	-0.07	0.10

mean curvature units: mm^{-1}*, Gauss curvature units:* mm^{-2}

The removal of vertices which beyond these bounds can remove some spikes and blobs, but there is still a problem with folded faces. To avoid that, we set bounds for dihedral angles. The analysis of values of 0.1% and 99.9% quantiles gives us distributions depicted in Fig. 5.16. Histograms show the distribution of the quantiles values for available development meshes. The lower quantile has the mean equal to 1.9183 radians and the standard deviation of 0.284 radians. The upper quantile has values 4.3461 radians and 0.2581 radians for the mean and the standard deviation, respectively. We considered these values and set the acceptable range of dihedral angles to the interval $(1.9199, 4.3633)$ radians which is equal to the range $(110, 250)$ degrees.

We hope that these principles of outliers detection together with assumptions about the scanned object surface can be used to set bounds also for types of scans different from the facial ones.

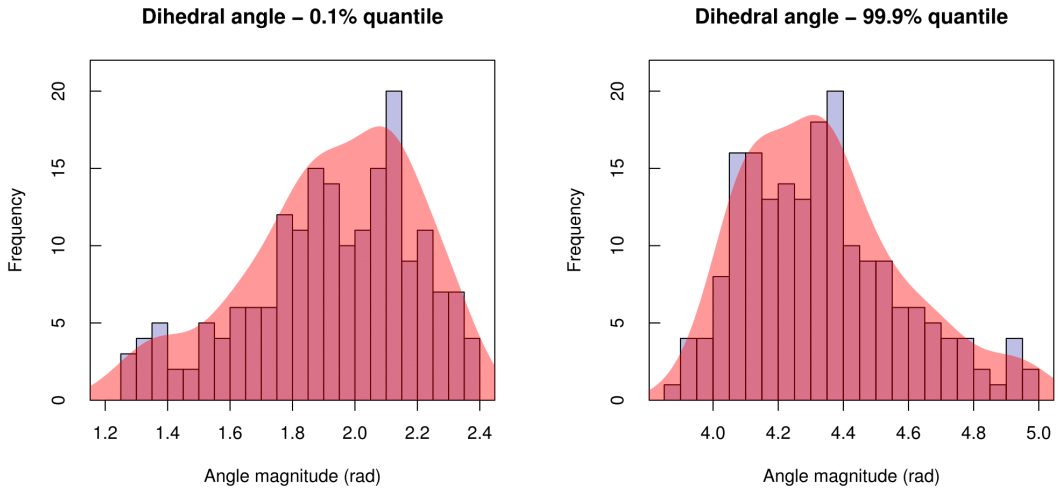

Figure 5.16: *Distributions of quantiles of dihedral angles (of 194 facial scans).*

5.5.3 Results

We used the algorithm for cleaning our whole development set and we have not encountered any serious problem or failure. The Fig. 5.17 shows the difference between distance distributions of a development facial scan before and after the defects removal step.

Both MeshLab and Geomagic can manage self-intersections. There is no tool for the spikes and blobs removal in MeshaLab, but Geomagic has some features which could be used to remove these defects. The first one is called *Select by curvature*, but the tool has only one parameter – *sensitivity* – which does not give us much control of the selected area. It tends to select really curved areas such as whole eyes, ears or eyebrows, so it is impossible to select just spikes using this tool. Another tool is called *Remove spikes*. As the official documentation states, it can detect and flatten single-point spikes on a polygon mesh. It seems that it smooths the mesh surface in some spiky areas. Its tolerance can be controlled by the *Smoothness level* parameter. However, the tool still fails if huge spikes or blobs are present.

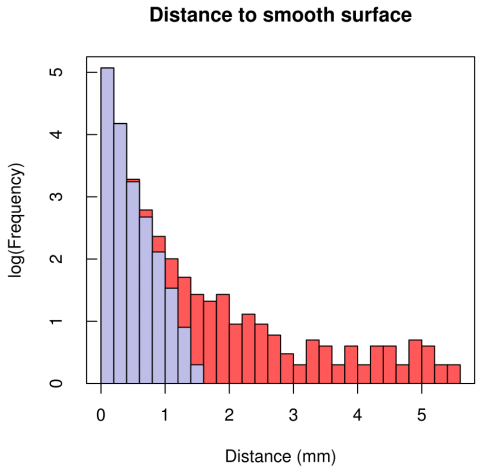

Figure 5.17: *Histogram of the vertex-to-smoothed surface distance of a random development mesh. Purple colour depicts the distribution after the cleaning. Red columns mark the difference to the original mesh. Around* 0.73 % *of vertices were removed.*

5.6 Hole Filling

We said that there exist holes even in raw facial scans. However, new holes might be created during the intersections and defects removal. Now, we are going to introduce our implementation and main principles of hole filling in general.

5.6.1 Methods overview

As mentioned in a survey by Ju [2009], there are three main principles of filling holes of triangular meshes:

- Example-based methods search for a similar patch somewhere (on the same or an additional mesh) and then use this patch to fill the hole. As stated by Ju, this method is widely used to complete large incomplete areas of facial scans from an existing database of examples. We do not use this type of algorithm because we usually need to fill many small holes or holes in very variable areas (e.g. eyebrows). Therefore, using this type of algorithms would be excessive on the one hand and not so confident in variable areas on the other.

- Other methods fit a geometrical object to the local neighbourhood of a vertex on a hole contour or to the whole hole contour. Then add new vertices at positions which are chosen with respect to the fitted object in order to reduce the size of the hole. We implemented a technique by Tekumalla and Cohen [2004] which fits a plane using a local Moving Least Squares approximation. The algorithm performed well on regular holes without a complex shape, but it was impossible to fill funnel-shaped holes or holes with complicated contour.

- The third class groups robust methods together. Initially, the holes are spanned with a naive triangulation. Heuristic searches have to be used to find an intersection-free and simply-connected 2-manifold triangulation because it was shown by Barequet and Sharir [1995] that it is an NP-complete task. Then the triangulation is improved by some fairing algorithms. We modified the approach proposed by Liepa [2003] and we also adopted few ideas from the algorithm by Pernot, Moraru, and Veron [2012]. Our algorithm is discussed in the next chapter.

5.6.2 Our implementation

In the first step, holes are identified by searching a closed cycle of boundary edges. All holes can be found in a single pass through mesh vertices because our data structure remembers a list of corners for each vertex. So we can find out whether a vertex resides on a hole boundary by passing corners of the vertex clockwise or counterclockwise. Obtaining the predecessor and the successor of a boundary vertex is then also trivial. Subsequently, contours of holes are cleaned as suggested by Pernot et al. [2012]. To be more specific, we recursively remove boundary vertices with a single adjacent triangle (Fig. 5.18).

Once a hole is identified and cleaned, we create a triangulation of the hole, especially of the three-dimensional polygon defined by the hole boundary. We use

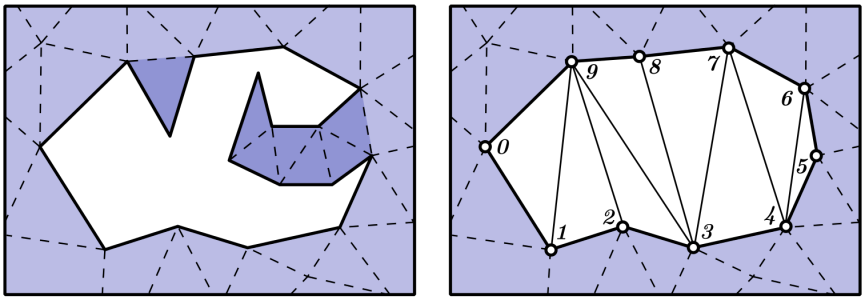

Figure 5.18: Left: *A hole with a jagged contour. The contour cleaning step removes the faces coloured with a dark colour.* Right: *An example of a naive triangulation of the same hole.*

dynamic programming to do this. Let $V = (v_0, v_1, \ldots, v_{n-1})$ be the boundary polygon and let $w_{i,j}$ denote the minimum-weight triangulation of the polygonal chain $(v_i, \ldots, v_j)$. A weighting function which assigns a certain weight to a triangle is denoted by Ω. We apply the following procedure to get an optimal triangulation with respect to the weight function:

```
n = hole.size()
for i = 0, ..., n - 1:
   w[i] = list(length = n-i-1, values = identity_element())
   t[i] = list(length = n-i-1, values = -1)

for j = 2, ..., n:
   for i = 0, ..., n - j:
      k = i + j
      m = -1
      min = minimal_value()
      for l = i+1, ..., k:
         val = w[i][l-i-1] + w[l][k-l-1] + Ω(i, l, k)
         if (val < min): (min, m) = (val, l)
      w[i][k-i-1] = min
      t[i][k-i-1] = m

return extract_triangles_recursively(0, n-1, t)
```

The weight function remains to be defined. Liepa [2003] enhanced the original weight function by Barequet and Sharir [1995] and defined it as a function $\Omega_l : V^3 \to L$, $\Omega_L(v_i, v_j, v_k) := (\mu(v_i, v_j, v_k), a(v_i, v_j, v_k))$ where μ is the largest dihedral angle between the triangle (v_i, v_j, v_k) and an existing neighbouring triangle, a denotes the size of the area of the triangle (v_i, v_j, v_k) and $L = [0, \pi] \times [0, \infty)$ with an identity element: $0_L := (0, 0)$, an ordering: $(a, b) < (c, d) \Leftrightarrow (a < c \vee (a = c \wedge b < d)$ and an operation of addition: $(a, b) + (c, d) := (\max(a, c), b + d)$.

The original weighting prefers triangulations which form a kind of convex hull to minimum-area triangulations. However, we modified it because it tends to produce triangulations with ill-shaped triangles, especially while filling holes with a complex shape. We replaced the function a with a triangle aspect ratio measure $q : V^3 \to \mathbb{R}$ mentioned by Pernot et al. [2012]. Suppose a triangle t with the area α, the semiperimeter σ and the longest edge h. Then $q(t) := 2\sqrt{3}\,\alpha/(\sigma h)$.

Note that α/σ is equal to the radius of the incircle of the triangle t. This measure implies values from the interval $[0, 1]$, is maximal for equilateral triangles, minimal for zero-volume triangles and it is scale independent. After modification, the algorithm prefers triangulations with regular triangles. Moreover, we changed the ordering to be less strict with angles: $(a, b) < (c, d) \Leftrightarrow (\mathrm{w}(a) < \mathrm{w}(c) \vee (\mathrm{w}(a) = \mathrm{w}(c) \wedge b < d)$ where $\mathrm{w}(x) := \mathrm{round}((x/2)^4)$. Due to this change, the algorithm is less sensitive to angles and produces better triangulations (Fig. 5.19).

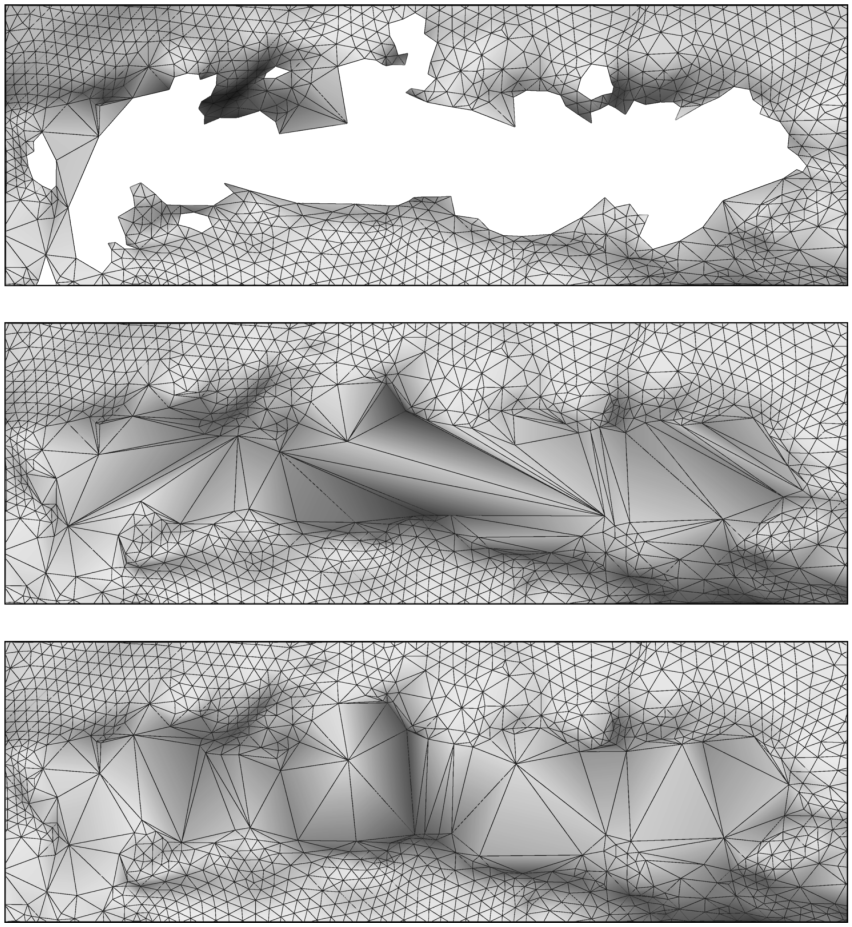

Figure 5.19: Top: *A huge hole in the eyebrow.* Middle: *The initial topological fill produced by the original Liepa's algorithm. There is a lot of ill-shaped triangles.* Bottom: *The naive triangulation made by our algorithm with enhanced weighting.*

We can now span a hole with a naive triangulation. However, the triangulation might not fit its local neighbourhood well. That is why we have to perform a refinement. The refined triangulation should be similarly dense as the surrounding mesh. We use the algorithm given by Liepa [2003] which is an adaptation of the original algorithm proposed by Pfeifle and Seidel [1996]. The idea is to subdivide triangles of the spanning triangulation to reduce edge lengths and to relax edges somehow to maintain a triangulation with similar density as surrounding triangles. The relaxation of an edge is a procedure which swaps the edge if at least one of two adjacent triangles are inside the circum-sphere of the other triangle (Fig. 5.20).

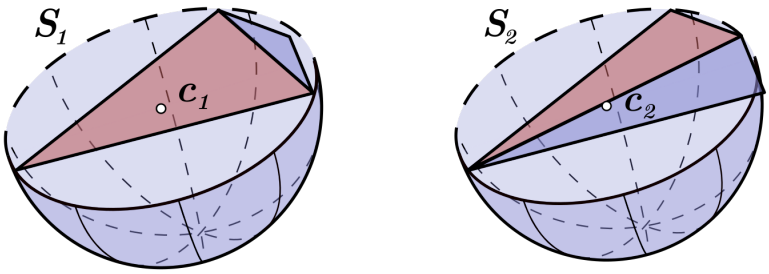

Figure 5.20: Left: *A half of the circum-sphere S_1 of the red triangle. The other triangle is inside the sphere, so the edge should be swapped.* Right: *The same situation after edge swap. Now, the new triangle is not inside the circum-sphere.*

The centre of a circum-sphere is the same point as the centre of a circum-circle in the plane of a particular triangle. We use the following code to check whether a point is in the circum-sphere of a triangle:

```
edge_1 = triange_3 - triange_1
edge_2 = triange_2 - triange_1
normal = e2 × e1
centre = (normal × edge_2) * edge_1.length_squared() +
         (edge_1 × normal) * edge_2.length_squared()) /
         (2 * normal.length_squared())
return (point - triange_1 - centre).length() <= centre.length()
```

Now, the whole refinement algorithm can be described as follows:

```
// compute scale attributes
foreach vertex in vertices: sum = 0
foreach neighbour  in vertex.neighbours():
   sum += (neighbour - vertex).length()
   vertex.scale =  sum / vertex.neighbours().size()

repeat until no change:
   foreach (v_1, v_2, v_3) in hole.triangles:
      centroid = (v_1 + v_2 + v_3) / 3
      scale = (v_1.scale + v_2.scale + v_3.scale) / 3
      verified = true
      foreach v  in (v_1, v_2, v_3):
         // condition proposed by Pfeifle and Seidel
         d = α * (centroid - v).length()
         if d > scale and d > v.scale:  continue
         verified = false
      if not verified:  continue
      // new vertex can be added into triangulation
      add_vertex(centroid)
      centroid.scale = scale
      remove_triangle(v_1, v_2, v_3)
      add_triangle(centroid, v_2, v_3)
      add_triangle(v_1, centroid, v_3)
      add_triangle(v_1, v_2, centroid)
      relax_edges((v_1, v_2), (v_2, v_3), (v_1, v_3))

   repeat until no change: relax_edges(hole.edges)
```

where α is a density control factor. We use the value $\alpha = \sqrt{2}$ according to the advice in the original article. The repeat-until-no-change loop ends as soon as no other changes occur to the hole vertices and edges, so that they stay the same during a loop iteration. It is not an endless loop, because we subdivide triangles on the one hand, and we relax each edge at most once per iteration on the other.

The last step in the hole filling algorithm is fairing. Fairing is performed in order to smooth and further improve the resulting patch. We implemented the technique introduced again by Liepa [2003]. Firstly, let $\mathbf{u}_i$ be a weighted arithmetic mean of positions of neighbours of a vertex i weighted by reciprocals of the distance between a particular neighbouring vertex and the vertex i. It means that $\mathbf{u}_i$ measures the deviation of the vertex i from a taut surface given by its neighbours. To measure the deviation of $\mathbf{u}_i$, we define:

$$\mathbf{v} := -\mathbf{u}_i + \frac{1}{\sum\limits_{n \in N_i} \frac{1}{\|v_i - v_n\|}} \sum_{n \in N_i} \frac{1}{\|v_i - v_n\|} \mathbf{u}_n$$

where N_i is a set of neighbours of the vertex i. Restricting $\mathbf{v}_i$ value to 0 implies an average tautness at a vertex i (with respect to the tautness of all its neighbours). To fair a hole patch, we would like to achieve $\mathbf{v}$ equal to 0 for each interior vertex i. This gives us a set of linear equations which can be at least approximately solved. The following pseudocode explains how to get the set of equations (more precisely a matrix and an appropriate vector) for a hole in terms of our data structure:

```
// we will write the code without updating all coordinates for brevity
n = hole.vertices.size() * 3   // we want entries for three coordinates
A = matrix(n, n)               // the coefficient matrix
x, b = vector(n)               // vectors for solution, eq. right-hand side

function weight(u, v): return 1 / (u - v).length()
function initialize(vertex, continuation, fixed):
   foreach neighbour in vertex.neighbours:
      total_weight += weight(vertex, neighbour)
   foreach neighbour in vertex.neighbours:
      effect = -2 * weight(vertex, neighbour) / total_weight
      if neigbour is in hole.vertices:
         A[hole.idx_of(neighbour)][hole.idx_of(vertex)] += effect
         ... update of all coordinates
      else:
         fixed += effect * neighbour ... update of all coordinates
      if continuation: initialize(neighbour, false, fixed)

for vertex in hole.vertices:
   idx = hole.idx_of(vertex)
   A[idx][idx] = 1 ... update of all coordinates
   // update the matrix A
   fixed = (0,0,0)
   initialize(vertex, true, fixed)
   // update vectors b, x
   b[idx] = -fixed ... update of all coordinates
   x[idx] = vertex ... update of all coordinates
```

We use the *Conjugate Gradient* iterative solver to solve the set of equations. This method enables parallelisation, that is why it can be fast. The maximum number

of steps done by the solver can be set in the configuration file of our algorithm (see Appx. A.6). Users are informed in case a hole could not have been faired enough after the maximal number of iterations.

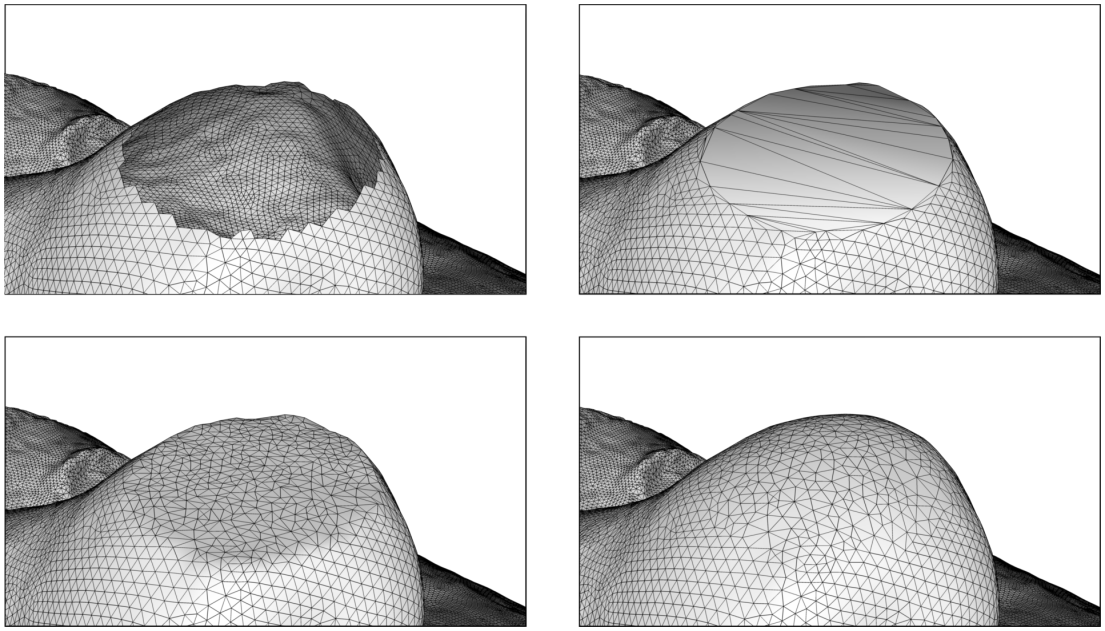

Figure 5.21: *This figure shows individual steps of our hole-filling algorithm.* Top left: *An artificial hole around the nose tip.* Right top: *An initial hole span.* Bottom left: *The hole after the refinement of the naive triangulation.* Bottom right: *The filled hole after all steps including the fairing.*

5.6.3 Results and future work

Now, we are going to compare our algorithm with the hole filling features offered by MeshLab and Geomagic. We are also going to mention a problem of our implementation and its possible solutions.

The quality of the hole filling must be assessed visually. The triangulation of a filled hole should have a similar density to the surrounding mesh, should not contain ill-shaped or degenerated triangles and the connection between the new patch and existing mesh should be smooth.

MeshLab had been offering the hole filling feature till the version v1.3.3, but it was removed in the version 2016.12. Marco Callieri, a MeshLab developer, proclaimed that it was done because of numerous bugs and instability. The version 2016.12 can create just a naive triangulation, but its quality is poor. So MeshLab currently does not provide a sufficient hole filling tool.

Geomagic Warp 2017 presents two slightly different hole filling features. These tools differ in the fairing step. It means that a user can choose whether the patch must match curvatures or tangents (which results in more tapering infill) of the surrounding mesh. However, both techniques give good results and produce a triangulation which consists of regular triangles and naturally follows the surrounding mesh. Compared to our algorithm (Fig. 5.22), the Geomagic's hole filling produces holes of similar shapes, but it can accommodate density and shape of the surrounding triangles in the better way.

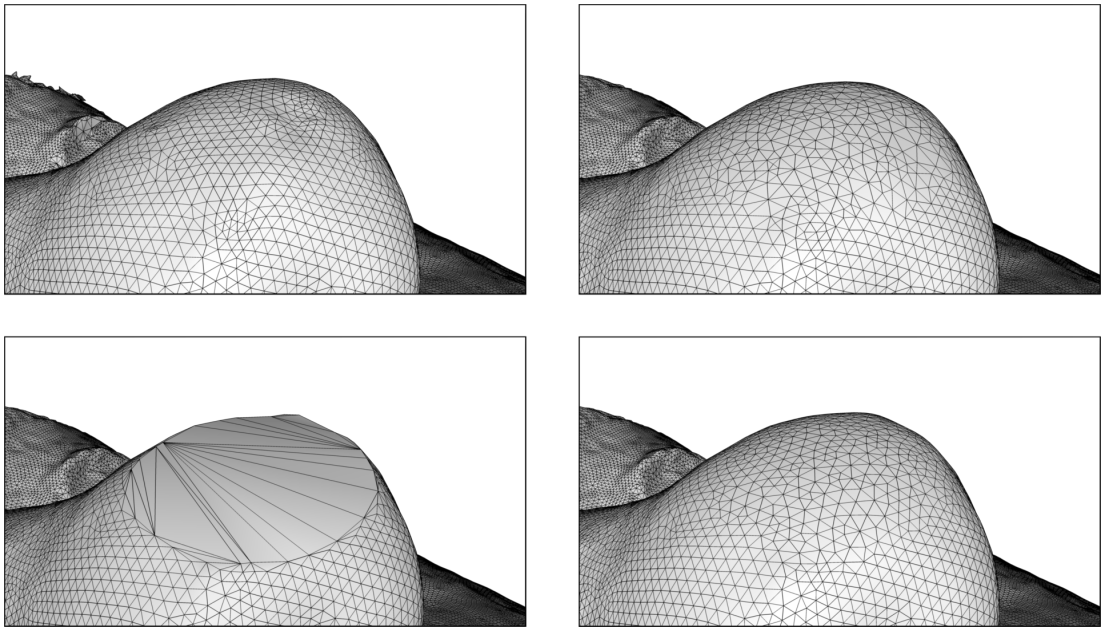

Figure 5.22: *The comparison of our algorithm and the algorithm of Geomagic and MeshLab. We use the hole from the Fig. 5.21.* Top left: *The original filling obtained from a scanner.* Top right: *The result of our algorithm. An irregularity of triangle shapes can be noticed.* Bottom left: *The filling produced by MeshLab.* Bottom right: *Geomagic's result. All new triangles are almost congruent.*

The problem of our implementation is that it cannot handle texture coordinates very well. Textures are not important for anthropological research since it focuses on faces. That is why we have not dedicated ourselves to this problem and have left it for the possible future work. Nevertheless, we are now going to describe the mentioned issue in detail.

While adding new vertices during the refinement step, we interpolate texture coordinates. The interpolation gives good results for small holes, but it assumes that the mapping from a texture to texture coordinates is continuous. That assumption is often erroneous as textures usually consist of a pair of photos. Thus, it is hard to set texture coordinates of fillings with vertices having texture coordinates from various parts of the texture image.

One possible solution of this problem (which was used in an older version of Rapidform, now called Geomagic) is to replace the original texture with a new texture which consists of triangles filled by colour gradients (Fig. 3.1). Of course, the triangles have to be chosen to approximate the original texture accurately. Therefore while adding new vertices and faces during the hole filling, a new texture triangle can be added to the texture. However, this technique requires creating of new images, so it is not trivial to implement it. Moreover, it produces an approximation and throws the original texture away.

Geomagic Warp 2017 can resolve this problem partially. It does not modify the original texture and it works perfectly with the original scanned mesh and texture (which consists of coupled images). Still there is a visible texture seam in critical areas, but it is not so obvious. However, if we tried to fill a hole on a mesh with the texture described in the previous paragraph, it failed in a similar way to the interpolation we implemented.

5.7 Thin Triangles

We use the triangle aspect ratio measure (which has already been mentioned in the previous chapter 5.6.2) to recognise thin triangles. The aspect ratio of a triangle t is defined as $q(t) := 2\sqrt{3}\,\alpha/(\sigma h)$ where α is the area of the triangle, σ is the semiperimeter of t and h denotes the longest edge of the triangle.

We resolve just triangles with the aspect ratio less than 0.25 (Fig. 5.23). The threshold depends on our needs, but the value 0.25 corresponds to really thin triangles, and thus there should be no need for changes. If a triangle to be resolved has one side noticeably shorter than the other two, we collapse that edge into a single vertex, therefore two triangles have to be removed in this scenario. The new vertex is placed in the middle of the two original vertices. Otherwise, the triangle has one long edge and two shorter, in this case, we try to swap the longest edge. We should remember that the edge might have only one adjacent triangle. In that case, we remove the thin triangle.

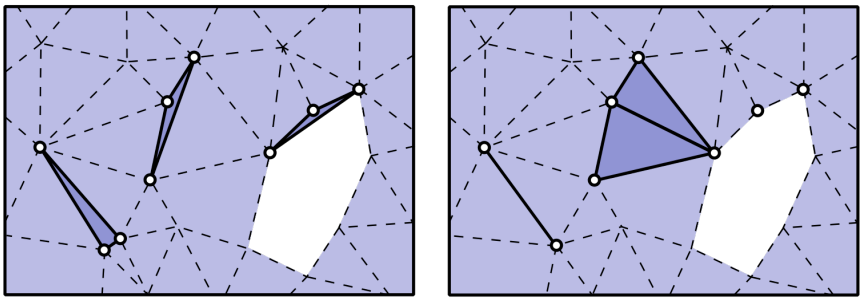

Figure 5.23: Left: *Triangulation with highlighted thin triangles. The three highlighted triangles have the aspect ratio equal to 0.25.* Right: *The same situation after applying our algorithm to resolution of thin triangles . All three cases which can occur are depicted.*

5.8 Rotation

The rotation is the very last step of our algorithm and, of course, depends on the face orientation. We have already explained how to determine face orientation in the chapter about cropping (Chapter 5.4.1). So we translate the whole mesh to align the face centre with the centre of the coordinate system and we also rotate the mesh to make the direction of sight and the vertical direction to be identical with z and y axes, respectively.

6. Technical Documentation

Now we are going to describe the internal structure of the program: individual classes and their usage and connections, some interesting implementation details and used libraries. We also give some advice on the modification of the program.

6.1 Structure

The program is written in C++ and has several classes. These classes are mostly just wrappers around methods which perform a particular step of the algorithm. Although some of them represent a triangle mesh, namely `Vertex`, `Face`, `Corner` and `Scene`. See Chapter 4 for details about connections between these classes. The following table describes the purpose of some other classes and namespaces:

Name	**Description**
`IModelHandler`	An interface for classes which can import and export meshes. There is implemented support for Wevefront .OBJ files (`WavefrontObjectFiles` class) and .ply files (`PolygonFileFormat` class). If you need to add support for another type, please implement the interface and change the appropriate part of the `main` function.
`Curvature`	A class holding information about curvatures (Chapter 5.2) for a single vertex. There are also grouped static methods for curvature computation.
`Cleaner`	The main class of our algorithm. It groups all cleaning methods which do not have own class (the face cropping, defects removal, thin-triangles resolution and rotation). The workflow of the whole program is defined in the `process` method.
`PointClassifier`	This class can perform the detection of landmarks. It loads and stores classifier models and manage their creating, feeding and disposing. It is dependent on TensorFlow sources.
`HoleFiller`	This class groups the whole hole filling algorithm and has a single public method `fill` which defines the hole filling workflow.
`MathSupport`	A namespace which groups mathematical and geometrical computations such as calculating angles, determining triangle intersections and solving linear equations.

6.2 Program flow

The program starts with `main` function execution which reads all command line options, the configuration file; the log file, which is connected to the `Logger` class is created. Afterwards, instances of `PointClassifier`, `HoleFiller`, and `Cleaner` classes are created. It also includes loading of classification models. In the next step, all input files are processed. They are successively opened, imported, processed and output files are created. The file format is derived from the file extension and, according to that extension, an appropriate mesh manipulator (import, export) is selected. The cleaning process starts by calling the `process` method of the instance of the `Cleaner` class. As a part of this method, there is a definition of the algorithm workflow which was described earlier (Chapter 5.1).

6.3 Implementation Details

6.3.1 Settings

To parse all command line options, load and parse the configuration file, the program uses `Program_options` library from the Boost which is distributed under the Boost Software License, version 1.0.

6.3.2 Curvatures

The computation of curvatures is not an expensive operation, but the smoothing of curvatures has a quadratic time complexity. We need to find a local neighbourhood for each vertex and then combine their tensors (explained in Chapter 5.2.2). Fortunately, the operation can be easily parallelised and we use OpenMP directives to do that.

6.3.3 Point detection

As well as curvature smoothing, obtaining the vertex neighbourhood is an expensive operation. Thus we also parallelise it. The point detector uses four neural network models to classify mesh vertices. Inference of feature vectors of all mesh vertices is also a time-consuming operation. However, we do not need to check all vertices. First, we ignore vertices whose neighbourhood is flat. Second, if we check a vertex, we postpone checking all neighbours of the vertex, and in case the vertex has a non-zero probability of being a landmark, we also check its neighbours. This principle allows us to find landmarks within a minute, but we cannot expect that we detect landmark areas with a very small number of vertices. Thus we discard candidate components (see Chapter 5.3.5) with less than seven vertices.

The models should be frozen TensorFlow models with an input node named `input_node` accepting a `float32` tensor of shape `[?,98]` ($6 * 16 + 2$, see Chapter 5.3.3 for the feature vector design explanation) and an output node named `output_node` which returns a `float32` tensor of shape `[?,2]`. So models are required to be binary classifiers.

6.3.4 Defects removal

A part of removing defects is the intersecting triangles removal. In order to detect them in an acceptable time, we divide the triangles into groups as described in Chapter 5.5. However, quadratic checking of all triangles in a group can take a long time, so we process the groups in parallel.

6.3.5 Hole filling

One step of the hole filling algorithm – fairing – needs to find an approximate solution of a set of linear equations. We use a conjugate gradient solver for sparse matrices of the Eigen library which is licensed under the Mozilla Public License version 2.0 and supports parallelism.

For the texture handling enhancement (see Chapter 5.6.3), see the methods `refinement` and `try_swap_edge` of the `HoleFiller` class.

7. User Documentation

The whole program is a console application, so it is without a user interface. It has to be run on the command line and it accepts several command line options. The behaviour of the program can be controlled using these options together with parameters in the configuration file. All these means of program configuration are summarised in Appendix A. Now, we are going to describe how to setup and use our program. Please check the appendix before further reading because we will suppose familiarity with some basic options and configuration parameters.

7.1 Initial Setup

Before using the program, all required files should be checked. Let us suppose that the program executable – `FaceCleaner.exe` – is present in a directory `C:\\Path`. By default, there should be also two directories: `libs` and `models` with some files. Besides the application executable, there should be a configuration file `face-cleaner.cfg`. However, if the configuration file is located elsewhere, its location can be specified by the command line argument `config[c]`. The directory `models` should contain four files named `eyes`, `tip`, `root` and `lips`. However, the location of these files can be changed in the configuration file (see Appx. A.3).

7.2 Program Purpose

The first thing we should consider is what do we want the program to do. As we have already explained, the algorithm has several phases. However, some of these phases might be skipped as we wish. The following demands may occur:

- **Extract the facial area** – the parameter `crop` should be set to `ON` if we want to remove ears. If we also want to remove the neck area, the parameter `crop-by-normal` should be set to `ON`. If the customisation of the trimming criteria is needed, read Chapter 5.4. The customisation might be desired if trimmed faces still contain unwanted areas.

- **Get facial landmarks** – the parameter `save-landmarks` has to be set to `ON`. The name of the input file is extended by suffix specified by the parameter `landmarks-suffix`, and a new file with that new name is created. The new file contains vertex indices corresponding to detected landmarks – nose tip, nose root, inner eye and mouth corners respectively – each on a new line. If only the nose tip is detected, the file contains a single vertex index, and if the detection is not successful, the file is empty. See Chapter 5.3 for details about the landmark detection if the detection often fails.

- **Remove geometrical errors** – set the `remove-defects` parameter to `ON`. The cleaning requires certain thresholds which were discussed in Chapter 5.5. However, it should not be necessary to change their default values.

- **Fill mesh holes** – set the `fill-holes` parameter to `ON`. This step should be enabled whenever the defects removal is enabled because removing geometrical errors usually produces many holes. Quality of the hole filling can be controlled using the `max-iters` parameter. Setting a large number causes long-lasting filling of large holes and setting very small number results in inappropriate hole shapes.

- **Resolve zero volume and thin triangles** – set the `thin-triangles` parameter to `ON`. Change the `min-triangle-aspect-ratio` parameter to control how thin triangles the algorithm should resolve.

- **Rotate faces to a uniform orientation** – set the `rotate` parameter to `ON`. The rotation might come in handy if we process many faces with various orientation and the common orientation is required.

Note that by default all the mentioned features are enabled.

7.3 Input

The program accepts command line options as described in Appendix A. Other command line arguments are supposed to be paths to input files. It means that we can feed the algorithm with many input meshes by a single command. The program can load Wavefront .OBJ files and it can also handle Polygon File Format (.ply) files both in binary and ASCII formats. If an input file cannot be loaded, the file is ignored and an appropriate error message is produced. Note that the processing of the subsequent input files is not cancelled. Please read Chapter 4.2 if a problem with file loading occurs.

7.4 Output

Processed files are saved into the directory specified by the `destination-path` parameter (in the configuration file). If a path is provided, all processed files are saved into the directory. However, the default value of the parameter is `.` which has a special meaning. It means that output files are saved alongside the input files, so if the input files are saved in different directories, the output files are also in different directories. In order to distinguish between input and output files, a suffix provided by the `file-suffix` parameter is added to the original input file name to form a new output file name. The format of output files matches the format of input files. Thus if an input file has the binary .ply format and another file has the ASCII .OBJ format, the resulting output files have the binary .ply and the ASCII .OBJ formats, respectively. Output files containing detected facial landmarks are created in the same way. The file should contain indices of landmark vertices. These indices are saved in this order – each on a new line: nose tip, nose bridge, inner eye corners and mouth corners.

The program also produces logs so that we can watch the progress or spot problems during the algorithm execution. Log messages include three types – informative (INF), warnings (WRN) and errors (ERR). Informative messages are produced just to track the progress of the program. Warnings notify us that

something unusual is happening and we should check the result – warnings can be produced during the hole shape optimisation or while detecting facial landmarks. Errors report critical problems which forbid the algorithm to be completed. Use the `verbose` option to filter out some types of messages. The logs are printed to the standard output and also to the log file. The log file can be specified by `log-file` parameter which is `.\face-cleaner.log` by default.

7.5 Example Usage

Now, we are going to show a sample usage, and we suppose the usage of the Windows operating system. Let us suppose that the current working directory (`C:\\Path`) contains our program's executable `face-cleaner.exe` and the default configuration file. The folder `C:\\PathToMeshes\Meshes` contains .obj files. We want to perform the full cleaning on all mentioned .obj files, but we do not need to save the landmarks. All output files should be stored in the directory `C:\\PathToOutput\Output` with the suffix `_processed`. We also need to limit the maximal number of used threads to 4.

In order to complete that task, we need to change the configuration file – set `save-landmarks = OFF`. All other settings can be controlled using command line options. Now, we can run our program:

```
C:\\Path> FaceCleaner.exe -t 4 -d C:\\PathToOutput\Output -s _processed ^
          C:\\PathToMehses\Meshes\*.obj
```

An output similar to this one should be displayed:

```
================================================================================
||  Running Automated Face Cleaning v1.0.0, 2018 by Tomas Nekvinda
||  Number of input files:  88
||  Cleaned files' suffix:  _processed
||  Destination path:       C:\\PathToOutput\Output
||  Log file path:          .\face-cleaner.log
||  Number of threads used: 4
================================================================================
[INF 16:10:54] Loading: C:\\PathToMehses\Meshes\a-face-mesh.obj
...
```

Note that existing files can be overwritten while saving output files.

8. Conclusion

The goal of this work is to develop algorithm which could be used by the Department of Anthropology and Human Genetics at the Charles University for automated post-processing (especially cleaning and trimming) of three-dimensional facial scans produced by the scanner Vectra3D [Canfield Scientific, Inc., 2018].

In view of that, we developed an algorithm which consists of several independent parts. Some of these parts use means of discrete differential geometry, e.g. curvature. That is why we reimplemented and compared some algorithms for computation of curvatures (Chapter 5.2.2). We also implemented simple algorithms for removal of vertices which violate 2-manifoldness or for deletion of minor connectivity components. To enable facial area trimming, we developed a novel algorithm for detection of facial landmarks which makes use of machine learning methods together with a description of surface geometry (Chapter 5.3). The actual algorithm for facial area trimming is based on a set of thresholds and uses information about detected landmarks (Chapter 5.4). Moreover, we reimplemented an algorithm for the removal of self-intersections, and we developed a method of spikes removal which is also based on curvature thresholds (Chapter 5.5). The thresholds were discovered with regard to a statistical examination of our development data. We also enhanced an existing hole filling algorithm to suit our requirements better (Chapter 5.6).

To summarise our work, we have created a piece of software which can automatically fix basic topological errors, detect intersecting triangles, spikes, and blobs and trim the facial area of an arbitrary facial scan. It produces hole-free faces without ill-shaped triangles, and moreover, it can automatically detect six important facial landmarks – nose tip, nose root, and pairs of eye and mouth corners. However, the detection produces just rough positions of the landmarks and cannot replace human annotators if an exact placing is needed. The workflow and parameters of the program can be easily adjusted via program options or via the configuration file.

We compared our program with features of MeshLab v2016.12 and Geomagic Wrap 2017. It can replace all relevant features of these two programs, although Geomagic produces slightly better-looking triangulation while filling large holes. Our algorithm could have been designed to perform the complete post-processing automatically (unlike both compared GUI programs) since it aims at a very specific task.

For a future work, the program could be modified to support correct handling of textures while filling holes. Note that we did not dedicate ourselves to that problem because textures are not crucial for anthropological research since it focuses on the shape of faces instead.

Bibliography

E. Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010. ISBN 978-0262012430.

G. Barequet and M. Sharir. Filling gaps in the boundary of a polyhedron. *Computer Aided Geometric Design*, 12(2):207–229, 1995.

Canfield Scientific, Inc. Vectra m3 3d imaging system, 2018. URL `https://www.canfieldsci.com/imaging-systems/vectra-m3-3d-imaging-system/`.

K. Crane, de F. Goes, M. Desbrun, and P. Schröder. Digital geometry processing with discrete exterior calculus. In *ACM SIGGRAPH 2013 courses*, SIGGRAPH '13, New York, NY, USA, 2013. ACM. ISBN 978-1450323390.

D. L. Donoho. High-dimensional data analysis: The curses and blessings of dimensionality. In *AMS conference on Math Challenges of the 21st century*, 2000.

D. Eberly. Intersection of convex objects: The method of separating axes, @ONLINE, 2008. URL `https://www.geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf`. visited on 2018-04-15.

E. Eftelioglu. *Geometric Median*, pages 1–4. Springer International Publishing, Cham, 2015. ISBN 978-3319235196.

N. Farahani, A. Braun, D. Jutt, T. Huffman, N. Reder, Z. Liu, Y. Yagi, and L. Pantanowitz. Three-dimensional imaging and scanning: Current and future applications for pathology. *Journal of Pathology Informatics*, 8(1):36, 2017.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. ISBN 978-0262035613. `http://www.deeplearningbook.org`.

J. Guo, X. Mei, and K. Tang. Automatic landmark annotation and dense correspondence registration for 3d human facial images. *BMC Bioinformatics*, 14 (1):232, Jul 2013.

T. J. Hutton, B. R. Buxton, and P. Hammond. Dense surface point distribution models of the human face. In *Proceedings IEEE Workshop on Mathematical Methods in Biomedical Image Analysis (MMBIA 2001)*, pages 153–160, 2001. ISBN 0-769513360.

T. Ju. Fixing geometric errors on polygonal models: A survey. *Journal of Computer Science and Technology*, 24(1):19–29, 2009.

J. Koudelová, J. Brůžek, V. Moslerová, V. Krajíček, and J. Velemínská. Development of facial sexual dimorphism in children aged between 12 and 15 years: A three-dimensional longitudinal study. *Orthodontics & craniofacial research*, 18, 05 2015a.

J. Koudelová, J. Dupej, J. Brůžek, P. Sedlak, and J. Velemínská. Modelling of facial growth in czech children based on longitudinal data: Age progression from 12 to 15 years using 3d surface models. *Forensic Science International*, 248:33 – 40, 2015b.

J. Lee. *Introduction to Topological Manifolds*. Springer, 2011. ISBN 978-1441979391.

P. Liepa. Filling holes in meshes. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 200–205, Aire-la-Ville, Switzerland, 2003. Eurographics Association. ISBN 1-581136870.

N. Max. Weights for computing vertex normals from facet normals. *J. Graph. Tools*, 4(2):1–6, 1999.

M. Meyer, M. Desbrun, P. Schröder, and A. H. Barr. Discrete differential-geometry operators for triangulated 2-manifolds, 2003.

V. Moslerová, M. Dadáková, J. Dupej, E. Hoffmannova, J. Borský, M. Černý, P. Bejda, L. Kočandrlová, and J. Velemínská. Three-dimensional assessment of facial asymmetry in preschool patients with orofacial clefts after neonatal cheiloplasty. *International Journal of Pediatric Otorhinolaryngology*, 108, 2018.

J.-P. Pernot, G. Moraru, and P. Veron. Filling holes in meshes using a mechanical model to simulate the curvature variation minimization. working paper or preprint, Dec 2012. URL `https://hal.archives-ouvertes.fr/hal-00761482`.

R. Pfeifle and H.-P. Seidel. Triangular b-splines for blending and filling of polygonal holes. In *Proceedings of the Conference on Graphics Interface '96*, pages 186–193, Toronto, Ont., Canada, 1996. Canadian Information Processing Society. ISBN 0-969533853.

A. Pressley. *Elementary differential geometry*. Springer, 2nd edition, 2005. ISBN 978-1848828902.

J. Rossignac, A. Safonova, and A. Szymczak. Edgebreaker on a corner table: A simple technique for representing and compressing triangulated surfaces. In *Hierarchical and Geometrical Methods in Scientific Visualization*, pages 41–50, Berlin, Heidelberg, 01 2003. Springer Berlin Heidelberg. ISBN 978-3642557873.

S. Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. In *Symposium on 3D Data Processing, Visualization, and Transmission*, sep 2004. ISBN 0-769522238.

L. S. Tekumalla and E. Cohen. A hole-filling algorithm for triangular meshes. Technical Report UUCS-04-019, School of Computing, University of Utah, 2004.

G. Turk. The ply polygon file format, @ONLINE, 1994. URL `http://paulbourke.net/dataformats/ply/`. visited on 2018-04-15.

Y. Vardi and C. H. Zhang. The multivariate l1-median and associated data depth. *Proceedings of the National Academy of Sciences of the United States of America*, 97 4:1423–6, 2000.

Wavefront Technologies. Appendix b1. object files (.obj), advanced visualizer manual, @ONLINE, 1998. URL `http://www.martinreddy.net/gfx/3d/OBJ.spec`. visited on 2018-04-15.

A. Settings

A.1 Command line options

Option	Abbr.	Type	Default	Action
version	-	-	-	Print information about the current program version.
help	-	-	-	Produce a help message introducing available options.
manual	-	-	-	Print a description of the configuration file parameters.
threads	t	pos. int.	0	Specifies the maximal number of threads available. Use 0 to determine it automatically.
verbose	v	{0,1,2,3}	3	Set verbosity level – 0 quiet, 1 error messages, 2 warnings, 3 full info.
config	c	string	.\face-cleaner.cfg	A path (absolute or relative regarding the program executable) to the configuration file.
log-file	l	string	.\face-cleaner.log	A path (absolute or relative regarding current directory) to the log file.
suffix	s	string	_cleaned	A string which will be appended to all output files. Make sure it can really form a valid file name!
destination	d	string	-	A path to the directory where all output files will be stored. If . is specified, output files are saved alongside input files instead of saving into the current working directory.

A.2 General settings

Parameter	Type	Default	Chpt.	Action
verbose	pos. int.	3	-	The command line option can be used instead. Set verbosity level – 0 quiet, 1 error messages, 2 warnings, 3 full info
log-file	string	.\face-cleaner.log	-	The command line option can be used instead. A path (absolute or relative regarding the program executable) to the log file.

Parameter	Type	Default	Chpt.	Action
suffix	string	_cleaned	-	The command line option can be used instead. A string which will be appended to all output files. Make sure it can really form a valid file name!
destination	d	string	-	A path to the directory where all output files will be stored. If . is specified, output files are saved alongside input files instead of saving into the current working directory.
scale	pos. num.	1	3.2	The number of mesh units which corresponds to 1 millimetre in reality.
save-landmarks	ON/OFF	ON	5.3	Set ON if you want to detect and store landmarks into file specified by landmarks-suffix parameter. OFF if you do not want to save the landmarks.
landmarks-suffix	string	_landmarks	-	A string which will be appended to all output files containing indices of detected landmarks.
crop	ON/OFF	ON	5.4	Set ON if you want to trim the facial area, OFF otherwise. Usually preserves neck. See also crop-by-normal and the group of face cropping parameters.
crop-by-normal	ON/OFF	ON	5.4	Set ON if you want to trim the facial area and you want to remove also neck, OFF otherwise. To have an effect, the crop parameter must also be set to ON. See also the group of face cropping parameters.
remove-defects	ON/OFF	ON	5.5	Set ON if you want to remove geometrical errors, OFF otherwise. See also the group of defects removal parameters.
fill-holes	ON/OFF	ON	5.6	Set ON if you want to fill mesh holes, OFF otherwise. See also the max-refinement parameter.
thin-trianlges	ON/OFF	ON	5.7	Set ON if you want to resolve thin or zero-volume triangles, OFF otherwise. See also the min-triangle parameter.
rotate	ON/OFF	ON	5.8	Set ON if you want to rotate the face mesh to a uniform orientation based od the face landmarks, OFF otherwise.

A.3 Landmarks detection

Parameter	Type	Default	Chpt.	Action
nose-tip-model	string	.\models\tip	5.3.4	A path (absolute or relative regarding the program executable) to the nose tip classifier.
nose-root-model	string	.\models\root	5.3.4	A path (absolute or relative regarding the program executable) to the nose root classifier.
eye-corner-model	string	.\models\eyes	5.3.4	A path (absolute or relative regarding the program executable) to the eye corners classifier.
mouth-corner-model	string	.\models\lips	5.3.4	A path (absolute or relative regarding the program executable) to the mouth corners classifier.
e2e-mean	pos. num.	24.302	5.3.5	The expected distance between inner eye corners in millimetres.
e2e-dev	pos. num.	4.366	5.3.5	The expected standard deviation of the distance between inner eye corners in millimetres.
m2m-mean	pos. num.	47.350	5.3.5	The expected distance of mouth corners in millimetres.
m2m-dev	pos. num.	4.941	5.3.5	The expected standard deviation of the distance of mouth corners in millimeters.
n2r-mean	pos. num.	48.322	5.3.5	The expected distance between the nose tip and the nose root in millimeters.
n2r-dev	pos. num.	3.896	5.3.5	The expected standard deviation of the distance between the nose tip and the nose root in millimeters.
n2e-over-n2r-mean	pos. num.	1.150	5.3.5	The expected ratio of the nose tip-to-inner eye corner distance to nose tip-to-nose root distance.
n2e-over-n2r-dev	pos. num.	0.0677	5.3.5	The expected standard deviation of the ratio of the nose tip-to-inner eye corner distance to nose tip-to-nose root distance.
r2e-over-e2e-mean	pos. num.	0.786	5.3.5	The expected ratio of the nose root-to-inner eye corner distance to eye corners distance.
r2e-over-e2e-dev	pos. num.	0.080	5.3.5	The expected standard deviation of the ratio of the nose root-to-inner eye corner distance to eye corners distance.

Parameter	Type	Default	Chpt.	Action
r2m-over-n2r-mean	pos. num.	1.748	5.3.5	The expected ratio of the nose root-to-mouth corner distance to nose tip-to-nose root distance.
r2m-over-n2r-dev	pos. num.	0.101	5.3.5	The expected standard deviation of the ratio of the nose root-to-mouth corner distance to nose tip-to-nose root distance.
n2m-over-m2m-mean	pos. num.	1.1699	5.3.5	The expected ratio of the nose tip-to-mouth corner distance to mouth corners distance.
n2m-over-m2m-dev	pos. num.	0.0995	5.3.5	The expected standard deviation of the ratio of the nose tip-to-mouth corner distance to mouth corners distance.

A.4 Face cropping

Parameter	Type	Default	Chpt.	Action
far	pos. num.	130	5.4	The maximal allowed distance between a mesh vertex and the nose tip in millimetres.
near	pos. num.	100	5.4	A distance from the nose tip in millimetres. Farther vertices are checked, nearer are not.
n2ch-over-n2r-mean	pos. num.	1.3976	5.4.1	The expected ratio of the nose tip-to-chin distance to nose tip-to-nose root distance.
n2ch-over-n2r-dev	pos. num.	0.085	5.4.1	The expected standard deviation of the ratio of the nose tip-to-chin distance to nose tip-to-nose root distance.
neck-normal-threshold	pos. num.	1.309	5.4.1	The maximal allowed angle (radians) between the direction of sight and the normal of any vertex near chin. See also the near parameter.

Parameter	Type	Default	Chpt.	Action
ears-normal-threshold	pos. num.	2.1817	5.4.1	The maximal allowed angle (radians) between the direction of sight and the normal of any vertex near ears. See also the near parameter.
neck-angle-threshold	pos. num.	0.1745	5.4.1	The maximal angle (radians) between the plane defined by inner eye corners and mouth corners and the vector defined by a vertex near chin and the center of face.
ears-angle-threshold	pos. num.	0.6981	5.4.1	The maximal angle (radians) between the plane defined by inner eye corners and mouth corners and the vector defined by a vertex near ears and the center of face.
wide-protrusion-range	pos. num.	6	5.4.1	The size of boundary neighbourhood (millimetres) which is used to find boundary protrusions. See also wide-protrusion-angle and tiny-protrusion-angle parametres.
wide-protrusion-angle	pos. num.	3.4907	5.4.1	The maximal tolerated angle (radians) between a boundary vertex and the two boundary vertices which are distant twice the wide-protrusion-range from each other.
tiny-protrusion-angle	pos. num.	4.7124	5.4.1	The maximal tolerated angle (radians) between a boundary vertex and the two boundary vertices which are distant roughly twice the mean edge length from each other.
curvedness-tolerance	pos. num.	0.08	5.4.1	The maximal curvedness accepted in the area between near and far (mm^{-1}).

A.5 Defects removal

Parameter	Type	Default	Chpt.	Action
maximal-angle	pos. num.	4.36	5.5	Maximal tolerated dihedral angles in radians.
minimal-angle	pos. num.	1.92	5.5	Minimal tolerated dihedral angles in radians.
maximal-mean-curv	number	0.93	5.5	Maximal tolerated magnitude of the vertex mean curvature (mm^{-1}).
minimal-mean-curv	number	-1.15	5.5	Minimal tolerated magnitude of the vertex mean curvature (mm^{-1}).
maximal-gauss-curv	number	0.89	5.5	Maximal tolerated magnitude of the vertex Gauss curvature (mm^{-2}).
minimal-gauss-curv	number	-1.50	5.5	Minimal tolerated magnitude of the vertex Gauss curvature (mm^{-2}).
maximal-mean-curv-smooth	number	0.28	5.5	Maximal tolerated magnitude of the smoothed mean curvature of a vertex (mm^{-1}).
minimal-mean-curv-smooth	number	-0.38	5.5	Minimal tolerated magnitude of the smoothed mean curvature of a vertex (mm^{-1}).
maximal-gauss-curv-smooth	number	0.10	5.5	Maximal tolerated magnitude of the smoothed Gauss curvature of a vertex (mm^{-2}).
minimal-gauss-curv-smooth	number	-0.07	5.5	Minimal tolerated magnitude of the smoothed Gauss curvature of a vertex (mm^{-1}).

A.6 Other settings

Parameter	Type	Default	Chpt.	Action
min-triangle	number [0,1]	0.25	5.7	Specifies the minimal triangle aspect ratio which is still tolerated. Any other triangles are resolved if the thin-triangles parameter is set to ON.
max-refinement	pos. int.	200	5.6.2	The maximal number of refinement optimization steps, high number causes long lasting filling of large holes (warning can be produced), very small number results in inappropriate hole shapes.