Feature Detection in Triangle Meshes

Bachelorarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Lukas Seeholzer

2021

Leiter der Arbeit:
Prof. Dr. David Bommes
Betreut durch:
Martin Heistermann
Institut für Informatik

Abstract

We propose an algorithm to quickly detect features on triangular meshes. Our implementation of this algorithm in the OpenFlipper environment is also part of the result of this thesis. The plugin offers an automatic detection method, which works by extending seed edges to cyclic feature paths, where the seed edges are found by rating the edges according to their likelihood to be part of features based on properties in a neighbourhood such as the dihedral angle while the extension of these edges follows a greedy approach. We also offer some basic tools to manually refine the result of the automatic tool. We set up a dual graph on which we perform all necessary calculations for both the automatic part as well as the manual tools.

Acknowledgement

I would like to thank Prof. David Bommes for providing the initial idea for this thesis, as well as for his continuous support including valuable inputs throughout the development of this work. Furthermore, I would like to express my sincere gratefulness to his assistant Martin Heistermann, who provided very useful inputs about my work at every stage and offered a good amount of motivation in our weekly meetings. Prof. Bommes and Mr. Heistermann both were there for me, whenever I could not progress on my own, for which I am very thankfull.

Contents

Chapter 1

Introduction

Feature lines

Features are regions of interest on a three-dimensional object that, from a human perception, mainly define the shape of the object. They may be characterized as being lines along the surface of the object that have minimal curvature in the direction of the line, have maximal curvature in the orthogonal direction and which are generally long. On a cube, for example, there are features running along the twelve edges, while there are no features inside the faces. In this case it is very easy to detect the features and a very simple algorithm (see algorithm 1) which just highlights all edges with a dihedral angle above a given threshold would suffice to reliably find all features on this object. In the aforementioned example, if we are given a clean mesh without any noise, where the dihedral angles along the feature edges are all exactly 90 degrees, this algorithm would deliver a perfect result for any angle threshold in the range of $[0°, 90°)$.

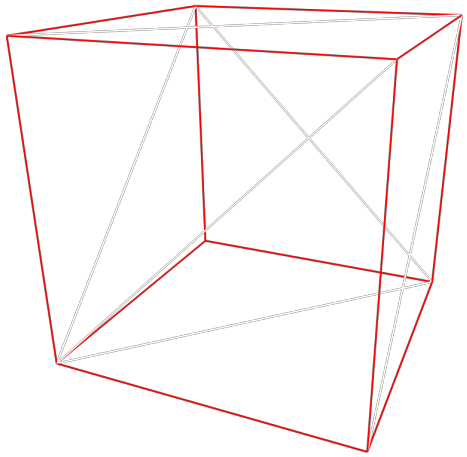


Figure 1.1: Simple cube consisting of 12 triangles with highlighted features

Algorithm 1 Simple Feature Detector(α, E)

```
for e ∈ E do
  if φ(e) > α then
    mark e as feature
  end if
end for
```

However, there are also objects with harder to detect features and there are regions of objects, where some dihedral angles are relatively high, but without features. Shallow features are one class of features that can be difficult to detect. Shallow features are lines on the surface of a mesh, where the dihedral angle is not very sharp but which still are features as they define the shape from a human perception. Consider an icosahedron, where the dihedral angles of the edges are around 42°. If we applied the algorithm from before with a threshold somewhere above 42°, then these features would not be found. However, in this case, we could just chose a smaller threshold and the algorithm would again find the features.

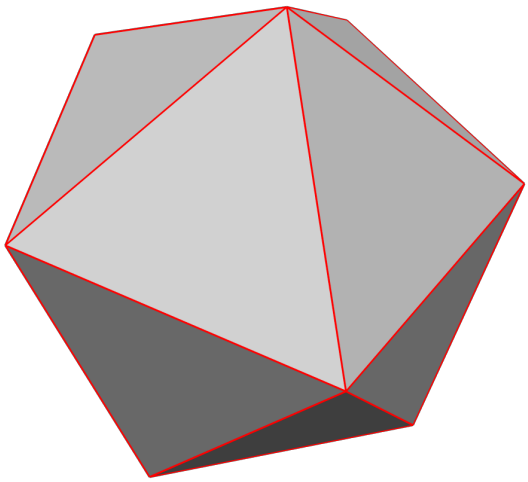


Figure 1.2: Icosahedron with highlighted features. We can see, that the dihedral angles are not very sharp, yet they are still all features.

On the other hand consider a region of an object with some smooth curvature, e.g. the side of a cylinder. Then there are some edges with dihedral angles strictly greater than 0°. Depending on the tessellation, these angles can be considerably greater than 0°. In this case we don't want these edges to be identified as features. Also, if there is some noise in the mesh, there might be some edges in noisy regions where the dihedral angle is considerably greater than the angle would be in a clean mesh. We don't want such edges to be identified as feature edges either. To prevent this from happening, we could just set the threshold in the algorithm described above great enough.

A problem arises, when we have both shallow features with relatively flat

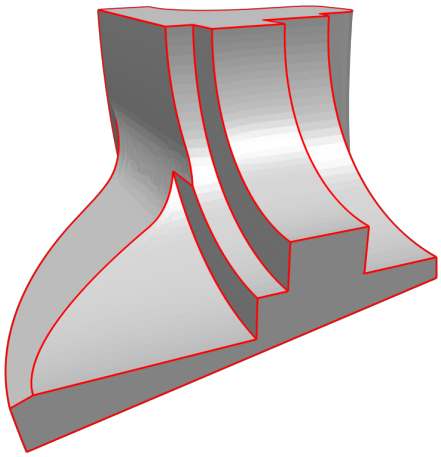


Figure 1.3: Fandisk with highlighted features. The feature in the front is a shallow feature and after a certain point, it isn't clear where the feature is exactly, as the feature is getting shallower. We can also see that the feature starting at the front and going to the left is shallowing and disappearing into the plane.

dihedral angles but which are indeed features and curved regions with relatively sharp dihedral angles but which are not considered features at the same time or if some edges have sharp dihedral angles due to noise, where in fact, there is no feature. In this case we can't just apply the mentioned algorithm as the threshold would need to be large enough to not consider edges in smooth curved regions and noisy regions but at the same time small enough to still find shallow features. In many cases this is not possible, and we need a more intelligent approach to still reliably find the features.

Apart from the mentioned problems, it is not even unambiguously clear in every case whether a certain edge is part of a feature or not. To illustrate this, consider a shallowing feature, i.e., a feature which starts as a clear feature but where the dihedral angle gets smaller as we proceed in one direction until it completely vanishes in a flat surface. In this case the edges at the beginning are clearly part of a feature but as the dihedral angle gets smaller it looks less and less like a feature. In this case the feature just stops at some point, but it is not clear at which point exactly.

Motivation

Many applications need to know the features of a mesh to work at all or to be able to deliver better results. Probably the most prominent of these applications is any form of remeshing. Transforming a triangle mesh into a quadrangle mesh is one such possibility of remeshing, refining the tessellation, decreasing the number of triangles to compress the file size, trying to obtain a more regular triangulation, various transformations, e.g. from a surface- to a volumemesh or vice-versa or smoothing can all offer advantages compared to

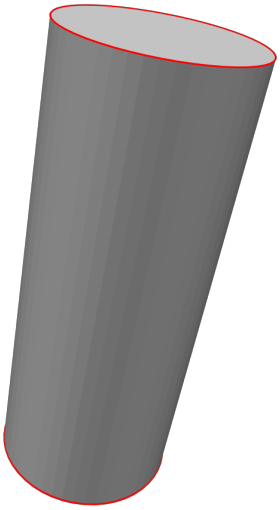


Figure 1.4: The Edges along the side of the cylinder have a dihedral angle $> 0°$, however, they aren't features. In this example, the model has quite a high tessellation, leading to dihedral angles close to $0°$. If the tessellation was smaller, the dihedral angles would be sharper and the edges might be considered features, which they are not.

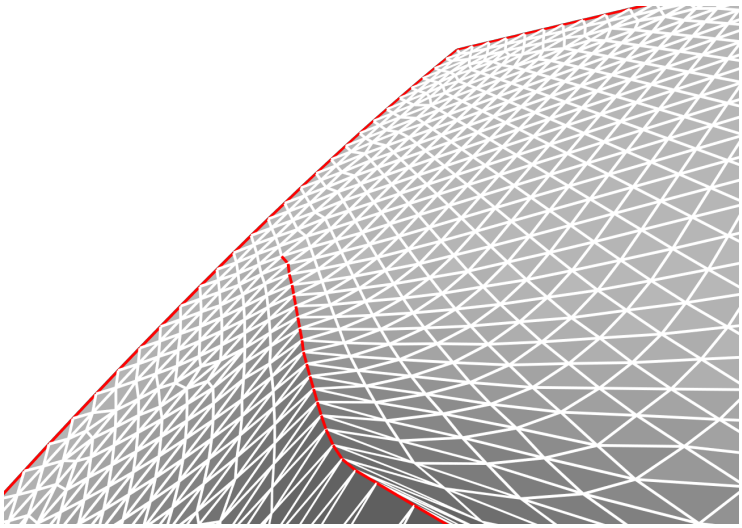


Figure 1.5: This is a closer look at the shallowing feature in the fandisk, also shown in Figure 1.3. We can clearly see that the feature is sharp in the front, but gets shallower towards the rear. It is not unambiguously clear, where it ends.

the original mesh. To prevent the general shape from being altered significantly by such a remeshing operation, one can set the feature lines as unchangeable.

This thesis

The goal of this thesis is to develop a feature detection algorithm and implement it as a plugin for the OpenFlipper [MK12] environment to prove its utility in practice. The plugin should provide the user with commands to automatically detect most of the obvious features and then, in a second step, to manually improve the selection of features.

The automatic part should be working with as little user input as possible and still be able to reliably find the most obvious features including shallow features but not including lines along curved regions. The time complexity of this action is also critical, it should essentially run instantaneous to enable the user to try different parameters to find the best result as quickly as possible even on large meshes.

The manual part should be easy to use and provide the user with a few tools to quickly remove features that were found by the automatic part but are not considered actual features by the user (false positives) and add features that were not found by the automatic part (false negatives).

We assume that the meshes, this algorithm is supposed to work on, contain clear features, are not too noisy and are sampled rather regularly. If these assumptions are not violated, the algorithm should deliver good results.

Chapter 2

Related Work

2.1 Related Work

[KCL09] introduce a method of feature detection on n-dimensional triangle meshes taking into account not only the curvature but also additional attributes, such as color. They calculate a normal voting tensor for each vertex and conduct an eigen-analysis of this tensor. The amount of the resulting eigenvalues not close to 0 determines the classification of this vertex as part of a face, an edge or a corner. After this analysis, they grow regions from seed triangles and extend these regions in all directions until they are bordered by sharp edges.

[LL02] introduce a method based on snakes which are already used in image analysis. This snake takes an initial position defined by the user and then slithers towards the closest edge and follows along this feature. It does this by optimizing an energy function dependent on the snakes smoothness and the sharpness of the feature. The smoothness energy is calculated as the weighted sum of the first and second derivatives of the feature line, the sharpness energy is computed based on the normal variation of the neighbouring faces of a vertex and is then linearly interpolated for all internal points. This method finds feature lines that do not necessarily follow along edges and do not even have to be on the mesh surface. In the end, all of these feature lines are then projected onto the mesh surface.

In [MTAM+19] several different approaches to detect features on triangular meshes are discussed. The goal was to find a subset of vertices on the mesh that form the feature lines. The SHREC'19 track was a challenge, to which four groups handed in their outcome. These four groups followed the following strategies:

- Spectral based saliency estimation for the identification of features (SBSE) by G. Arvanitis and K. Moustakas:

  The proposed method consists of two steps. In the first step, vertices are rated based on the curvature in their local neighbourhood. Based on this rating, the vertices are grouped into five classes by a k-means algorithm. In the second step, these vertices are clustered based on their mean curvature to form features.

- Point aggregation based on angle and curvature saliency(PCs) by Nhat

Hoang-Xuan, E-Ro Nguyen, Minh-TrietTran:

This group proposes two approaches. The first approach finds candidate vertices by detecting edges with a relatively sharp angle between the two adjacent triangles and then considering the two vertices of this edge. Then, pairs of candidates connected by very long edges are removed and the candidate vertices are connected to form components. Finally, very small components are removed. The second approach computes the curvature of the edges based on the normals of its adjacent vertices. The curvature of a vertex is then computed as the average of the curvatures of adjacent edges. These vertex curvatures are then refined and vertices with high curvatures are considered as feature candidates. The vertices are then connected to form components and finally, very small components are removed.

- Point-based multi-scale curve voting (PMCV) by T.Lejemble, L. Barthe and N. Mellado:

  This group samples the Mesh into a point cloud and then computes the local curvatures. A voting value is assigned to each point based on the curvatures in a local neighbourhood. Finally, a region growing process finds the feature lines along points with similar voting values.

- Feature curve characterization via mean curvature and algebraic curve recognition via Hough transforms (MHT)by C. Romanengo, S. Biasotti and B. Falcidieno:

  This method finds feature candidate vertices where the mean curvature is significant. These candidates are clustered to feature lines. The curves are then classified by Hough transforms.

[PC20] This approach finds umbilic points (points, where the curvatures in all directions are the same) on the mesh by evaluating the principal curvatures and then looking for local minima of $\kappa_1 - \kappa_2$. Apart from that, it finds elliptic and hyperbolic ridges by evaluating the curvatures.

2.2 Differences to our Approach

The main difference between the mentioned related work and our approach is that we do not consider the meshes we work on as smooth surfaces but as discrete meshes. This prevents us from taking first and second derivatives into account, which most of the mentioned approaches do. This gives us less mathematical tools to work with, but it allows us to work more closely on the input mesh. A problem that arises when interpreting a mesh as a smooth object is that a typical object is not completely smooth. It normally consists of some smooth patches, which are connected in non-smooth features. As it is exactly these features we want to find, such an approach might not be optimal. If one works directly on the discrete mesh, however, these problems do not arise.

Apart from that, there was no tool that allowed the feature detection in the OpenFlipper environment prior to this work. So, a main goal of this work is to provide such a tool to the users of OpenFlipper.

Chapter 3

Mathematical Definition of Features

3.1 Definition of Feature Lines

We define a feature line as a path along edges on a triangle mesh that defines the shape of a three-dimensional object from a human perception. Feature lines are characterized by relatively sharp dihedral angles and relatively smooth angles in the direction of the feature. One could also define feature lines just as lines on the surface of a discrete mesh or even on a continuous surface. For this thesis, however, we only consider edge paths as feature lines. We divide feature lines in different classes:

- Long Sharp Features

  These features are the easiest to detect for humans and probably also for machines. They have sharp dihedral angles everywhere and are rather long.

- Long Shallow Features

  These features are long and their dihedral angles are relatively sharp compared to a local neighbourhood which is rather flat but the dihedral angles might be relatively flat compared to sharp features. Another characteristic of such features is that, generally, the patches on either side are relatively flat and large. One of the difficulties in detecting them is to distinguish them from regions with smooth curvature such as the side of a cylinder.

- Short Sharp Features

  These features have relatively sharp dihedral angles but are shorter than the long sharp features. They often connect other sharp features and end in sharp corners. One of the difficulties in detecting such features is to distinguish them from high frequency noise.

- Shallowing Features

  These features start off as relatively sharp features and get shallower as they progress in one direction. The detection of such features is relatively

easy as the sharp part can be found by looking for edges with sharp dihedral angles. The difficulty with these features is to decide whether they continue in a flat region or end, and if they end, at which point exactly they end.

- Short Shallow Features

  These are the most difficult to detect, as they combine the difficulties of short features with those of shallow features. Often, it is also difficult to distinguish such features from noise, in some cases even for humans.

3.2 Mathematical Definitions

- Mesh M(V,E,F) consisting of a set of vertices V with $|V|$ elements and $v_i \in V : \forall i \leq |V|$, a set of edges E with $|E|$ elements and $e_i \in E : \forall i \leq |E|$ and a set of faces F with $|F|$ elements and $f_i \in F : \forall i \leq |F|$.

  An edge $e_i \in E$ connects two vertices $v_j, v_k \in V, v_j \neq v_k$, thus an edge can also be defined as $e(v_j, v_k)$. Of course, $e(v_j, v_k) = e(v_k, v_j)$, as we work on undirected meshes. We write $v_j \in e_i$ iff $e_i = e(v_j, v_k)$ for some $v_k \in V$.

  The Euclidean length of an edge e is notated as $|e|_2$

  A face $f_i \in F$ is defined by three vertices $v_j, v_k, v_l \in F, v_j \neq v_k \neq v_l$, which have to be connected by three edges, so $e(v_j, v_k), e(v_j, v_l), e(v_k, v_l) \in E$ or by these three edges. Thus a face can also be defined as $f(v_j, v_k, v_l)$ or $f(e(v_j, v_k), e(v_j, v_l), e(v_k, v_l))$. We write $v_j \in f_i$ iff $f_i = f(v_j, v_k, v_l)$ for some $v_k, v_l \in V$ and $e_j \in f_i$ iff $f_i = f(e_j, e_k, e_l)$ for some $e_k, e_l \in E$.

- Euclidean length of an indexed edge $|e_i|_2$

  The Euclidean length of an edge e_i: $|e_i|_2$ connecting vertices v_j and v_k in mesh m is the Euclidean length of the vector from v_j to v_k. Let p_j and p_k be the three dimensional points representing the position of v_j and v_k respectively, then $l(e) = |p_k - p_j|_2$.

- Path

  A path $P(E_P)$ is a sequence of consecutive edges in a mesh M with $|E_P|$ edges and $\forall i \leq |E_P| : e_{p,i} \in E_P$. All edges in this path have to be in the set E of edges of the Mesh M: $\forall e_{p,i} \in E_P : e_{p,i} \in E$. The edges have to be consecutive, i.e. $\forall i < |E_P| : e_{p,i}$ and $e_{p,i+1}$ are adjacent, i.e. $\exists v \in V : v \in e_{p,i} \wedge v \in e_{p,i+1}$. We can also define such a path with the consecutive vertices it passes through, where $\forall i \leq |E_P| + 1 : v_{p,i} \in E_P$. They have to be consecutive, i.e. $\forall i \leq |E_P| : e(v_{p,i}, v_{p,i+1}) \in E_P$. Additionally, such a path may not self-intersect or contain coinciding parts, i.e. $\forall i, j \leq |E_P|, i \neq j : e_{p,i} \neq e_{p,j}$ and $\forall i, j \leq |E_P|+1, i \neq j : v_{p,i} \neq v_{p,j}$. $|E_P|$ and $|P|$ describe the same and correspond to the amount of edges in E_P and $|P|_2$ corresponds to the Euclidean length of the path, i.e. $|P|_2 = \sum_{i=1}^{|E_P|} |e_{p,i}|$, the sum of the Euclidean lengths of all edges in the path.

- Angle between two Edges θ

  We define the angle between two edges θ to be the angle between the two vectors pointing along the edges in the "same" direction. This means that

one vector points to the vertex connecting the two edges and one vector points away from this vector. It doesn't matter which vector points to the center vertex and which one points away from it, the resulting angle will always be the same. With this definition, two edges have an angle θ of 0°iff they are parallel, 90°, iff they form a 90°corner and 180°iff they are on top of each other. In a triangle mesh which is not degenerate, the last case cannot happen, thus θ is always in the range of [0°, 180°).

θ_i is the angle between the edges e_i and e_{i+1} in a path p.

- Dihedral Angle φ

  We define the dihedral angle φ of an edge to be the angle between the normals of the two adjacent faces, such that the angle is 0 iff the two faces are coplanar but not coinciding and is approaching π for sharper angles. It can never become π, as this would mean that the two faces are coinciding, which is not allowed in a non degenerate mesh. It can also never be greater than π, as we always consider the smallest angle between the two faces. See figure 3.1 for two illustrations of this.

  Let n_1 and n_2 be the normals of the two adjacent triangles. Then the two following equations hold:

  - $\sin\varphi = \frac{|n_1 \times n_2|}{|n_1| \cdot |n_2|}$
  - $\cos\varphi = \frac{n_1 \cdot n_2}{|n_1| \cdot |n_2|}$

  The angle φ is now calculated the following way:

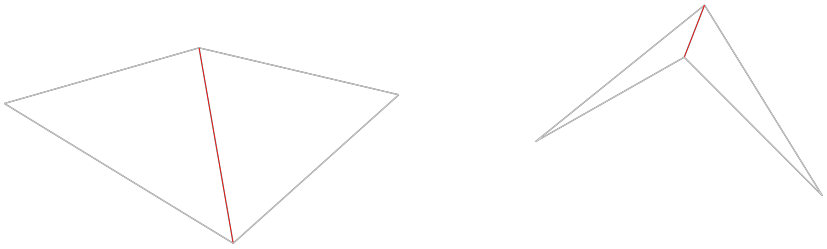
$$\varphi = \begin{cases} \arcsin\sin\varphi, & \text{for } \cos\varphi > 0 \\ \pi - \arcsin\sin\varphi, & \text{for } \cos\varphi < 0 \\ \frac{\pi}{2}, & \text{for } \cos\varphi = 0 \end{cases} \tag{3.1}$$

  We assume that the normals of the triangles are either all pointing outwards or inwards wrt. the mesh. This way, φ is 0, iff the two triangles are coplanar but not coinciding, $\frac{\pi}{2}$, iff they form a right angle regardless of the orientation (concave or convex) and π, iff they are coinciding, which cannot happen in a mesh that is not degenerate. This means that $\varphi \in [0, \pi)$

- Cost $c(p)$

  The cost of a path is a function of the path taking into account the dihedral angle φ of all edges in this feature, the angle θ between each pair of consecutive edges in the feature, the smoothness of the local neighbourhood, i.e. the deviation of the normal vectors of the trianglefan adjacent to the vertex in the middle and the total length of the feature. The general idea is that the cost goes up the worse a proposed feature is, i.e. the less likely it is that this can actually be considered a feature and the cost goes down the better a proposed feature is, i.e. the more likely it is that this can be considered a feature.

  The definition of such a function is thoroughly developed and discussed in the next section. We also refer to this cost function as the rating of a feature.



(a) The dihedral angle φ of the highlighted edge is completely flat, so $\varphi = 0°$.

(b) The dihedral angle φ of the highlighted edge is exactly 90°.

Figure 3.1: Comparison of dihedral angles

A summary of these definitions as well as of all other relevant definitions can be found in chapter 6 at the end of this thesis. We decided to have the glossary at the end of the thesis, as different items of it get defined in different places throughout the thesis.

3.3 Rating of Feature Lines

We would like to rate feature lines with mathematical means based on their likelihood to be a good feature line. This rating should take into account the dihedral angle φ of all edges in this feature, the angle θ between each pair of consecutive edges in the feature and the total length of the feature. We can divide the different properties that define the quality of a feature in three classes:

- Length

  We are talking about the Euclidean length of a feature here. The longer this Euclidean length is, the better is the feature.

- Sharpness

  The sharper the dihedral angles φ are, the better is the feature line.

- Smoothness

  The smoother the feature line along the direction of the feature is, the better is the feature line. This means that the angles θ are all relatively small.

Desired behaviour of the function $c(P)$ for a path P depending on feature characteristics:

- Length

  The longer the Euclidean length $|P|_2 = \sum_{i=1}^{|P|} |e_i|_2$ of the feature P gets, the smaller the cost $c(P)$ should be.

- Dihedral Angle φ (Sharpness)

  To rate a feature regarding the dihedral angles, several options can be considered:

  - Minimum: $P\_min\_dihedralAngles = min_{i=1}^{|P|}\varphi_i$
    This option considers only the flattest angle in the path, leading to good ratings only for paths that have only sharp dihedral angles. In many cases, this would be a good option and distinguishing sharp features from moderate noise is rather easy. Also, we can be sure, that all features found with this option are quite sharp. However, just a few or even a single outlier with a very flat dihedral angle lead to a very bad rating, which can make it difficult to detect shallowing features.
  - Average: $P\_avg\_dihedralAngles = average_{i=1}^{|P|}\varphi_i$
    If we rate the paths based on the average dihedral angles, then a few outliers are tolerated and a path with mainly sharp angles can still get a good rating even if there are some bad edges in between.
  - Quantiles: $P\_q\_quantile = max\{\alpha : |\{\varphi_i \in P : \varphi_i < \alpha\}| \leq q * |P|\}$
    This approach is in some sense in between the other two approaches, as one or, depending on the choice of q and the length of the path, several bad outliers are tolerated and such a path might still be rated quite well. But if the amount of bad angles gets too high, then the rating will be very bad. If we choose $q = 0.5$, then we get the median of the angles, which might rate paths with a lot of very bad angles still relatively good. This is not a favourable behaviour, so, if we consider the q-quantile approach, then q should be quite small.

- Angles between two consecutive edges θ (Smoothness)

  To rate a feature regarding the smoothness, also several options need to be considered:

  - Sum
    $P\_sum\_\theta = \sum_{i=1}^{|P|-1} \varphi_i$
  - Average
    $P\_avg\_\theta = average_{i=1}^{|P|-1}\varphi_i$
  - Maximum
    $P\_max\_\theta = max_{i=1}^{|P|-1}\varphi_i$

  As for the dihedral angles, there are also several options how to rate a feature path based on its smoothness. The advantages and disadvantages of the average and maximum option are similar to the dihedral angles.

  In this case, however, calculating the sum of the angles is another option as the angles should all be as close to 0 as possible. If the rating is based on the sum of these angles, then a few sharp angles can already greatly decrease the rating similar to the maximum option. In contrast to the maximum approach, however, the sum method might return still some acceptable rating if there are only very few sharp angles. In this sense, the Sum approach is somewhere in between the average and the

maximum approach. However, if a simple sum is applied, then long paths are penalized, which is definitely not a desirable behaviour. So, if we use the sum, then we should also take the total length of the path into account. Depending on how we take the length into account, this might effectively result in an average.

- Mixed Situations
  - Short Sharp Features

    These features have relatively sharp dihedral angles: $\forall i \leq |E_P| : \varphi > \alpha$ for some large α but are relatively short: $|P|_2 < a$ for some small a. In this case we want $c(P)$ be smaller than for paths with flatter dihedral angles but greater than for longer paths.
  - Long Shallow Features

    These features are rather long and their dihedral angles are sharp compared to a local neighbourhood which is relatively flat but the dihedral angles might be relatively flat compared to sharp features. We can express this with the following three inequalities:
    - $|E_P|_2 > a$ for some large a
    - $\forall i < |P| : \varphi_i < \alpha$ for some small α
    - $average_{i=1}^{|P|} \varphi_i > average_{i \in E'} \varphi_i$ for some Edges $E' \subseteq E$ in a local neighbourhood of P.

    Shallow features are often still perceived as features by humans, so we still want them to be detected as feature lines, so $c(P)$ should not be too high in such cases.
  - Shallowing Features

    The dihedral angles of the edges in some parts of the feature are relatively sharp: $\varphi_i > \alpha$ for some large α but are relatively shallow in other parts. In this case we want $c(P)$ be relatively small and getting smaller for longer paths as longer paths should be cheaper than short ones but also getting bigger,if φ gets smaller, as shallow features should be more expensive.

In the final rating of a feature path all of these properties should be considered in a way that as well short sharp features as long shallow features get a low cost rating while noise, which are typically short paths with sharp dihedral angles and probably also sharp angles between two consecutive edges should get a high cost rating. Also, paths in curved regions, which are typically rather long, might have smooth angles between two consecutive edges and also probably flat dihedral angles should get a high cost rating.

The calculation of such a rating actually poses some problems. For example, as described above, we would like to have a better rating for longer paths, but also have a worse rating for paths that violate some of the feature properties. Often, we could make a path longer by just adding some edges to it, which at the same time might worsen the path's feature properties. The main difficulty is now to compare the two versions of this path and to decide which one is better. We did not manage to come up with a function that can calculate a rating that satisfies these requirements, so the description of this rating function in this chapter is a purely theoretical description of a function we have not found

and may not exist. Nevertheless, we included it in this chapter for pedagogical reasons because the behaviour of this function is helpful for identifying the properties of features. The (purely local) modifiers, which will be discussed in the next chapter, are based on the general ideas of this function with the difference that they are actually implemented in the final algorithm.

Chapter 4

Proposed Method

4.1 General Idea

The fundamental idea is to build a dual graph which has the mesh edges as its nodes and where two nodes are connected with a link, iff the corresponding edges in the mesh are adjacent to the same vertex. For reasons of legibility and to avoid confusion, the terms mesh, vertex and edge are used when we talk about the original 3D mesh, while the terms graph, node and link are used for the dual graph, based on the mesh, henceforth. For a complete list of the terms and abbreviations we use here, please refer to chapter 6. We then calculate modifiers on the nodes and links, which indicate the likeliness of an edge being part of a feature line respectively the likeliness of two connected edges to be in the same feature line. These modifiers are then multiplied with the Euclidean lengths of the edges to obtain a modified virtual length for each link. This length is now dependent on the likelihood of the two corresponding edges to be in the same feature line, the likelihood of them being in a feature line at all and of the Euclidean lengths of these edges. This allows us to run Dijkstra's algorithm [D$^+$59] on the graph to determine the optimal path between two edges, following features but also minimizing the Euclidean length at the same time.

This method immediately allows the implementation of a simple manual feature detector, where a user can easily and effectively define a feature line of arbitrary complexity by selecting only a few control edges. As the construction of the graph and the calculations of the modifiers and the virtual lengths has to be done only once in a preprocessing step and as Dijkstra's algorithm is very efficient, this manual selection can be calculated in real time.

The automatic detection is divided into three steps:

1. Detection of a few seed edges

   Dijkstra's algorithm is run from every node until a specified modified distance is reached. This allows us to find the shortest path from every touched node to the original node wrt. the modified virtual length of the links. Of these shortest paths, the longest path wrt. the Euclidean length is selected and for each node on this path we add the Euclidean length of the path to a field on this node. The sum of these lengths is then used to grade the nodes for the likelihood of being part of a feature line. The idea is that the shortest paths wrt. the modified virtual length

tend to follow along features, so edges on features will be visited more often than non-feature edges. Also, we don't just count the amount of paths running through an edge but add the Euclidean lengths of these paths. The idea behind this decision is that the longer a path is wrt. the Euclidean length, the more likely it is that it follows a feature and very short paths just connecting two nodes don't necessarily run along features. This way, feature edges will get a higher rating because it is more likely that many such shortest paths as described above run through them and because the value added to their rating will probably be higher for each path that runs through them as these paths are more likely to run along features. So, such edges profit twice from the fact that they are in features. This allows us to extract the nodes that are most likely in feature lines such that we then can extend them along feature lines. The abort condition of the Dijkstra algorithm depends, of course, on the maximal modified virtual length. This is a parameter called extension depth that may be altered by the user. See 4.9 for more information on the meaning of this parameter and the effects of changing it.

2. Extending these seed edges along feature lines to form cyclic feature paths

   From a seed edge, we extend the feature greedily in both directions until they connect to another already found feature or to itself. This idea follows the observation that features generally are not isolated but are part of a feature network or at least closed cycles. However, it is possible that a feature is neither connected to other features nor forming a cycle. This case is not well covered by this approach and needs to be manually improved in a later stage.

3. Repeating the two first steps until all features are found

   From all the edges that are not yet in a feature line, we select the one with the highest rating as the next seed edge and extend it. We repeat these steps until we have found the specified amount of features as indicated by the user. This is the main user input required to find as many of the actual features as possible while having as few false positives as possible.

The important steps of this approach are more thoroughly described in the following sections.

Following is a pseudocode illustrating the general idea described above:

Algorithm 2 Feature Detection

```
Create dual graph
Calculate node modifiers
Calculate link modifiers
Rate nodes based on modifiers
while not all features found yet do
    Choose node with best rating not yet in any feature
    Extend seed node to cyclic feature
end while
```

4.2 Creation and Use of Dual Graph

We define the following three objects:

- Node n

  A node n of our dual graph g represents an edge e of the original mesh m. As each node represents exactly one edge, we have of course $|N| = |E|$, where N is the set of all nodes in g and E is the set of all edges in m.

- Link l

  A link l of our dual graph g connects two nodes iff the two edges e_i and e_j represented by these nodes are neighbours in the mesh, i.e. they are connected to the same vertex v in m. In a reasonably fine tessellated mesh, the mesh surface locally approaches a plain on average, leading to an average vertex valency of around 6. A vertex with a valency of 6 leads to 15 connected pairs of edges at this vertex. So, we can expect that $|L| \approx 7.5|V|$, where L is the set of links in g and V is the set of vertices in m.

- Graph g

  The dual graph g consists of a list N of nodes with $n_i \in N$ for $i \in [1, |N|]$, where $|N|$ is the amount of nodes in N and $|N| = |E|$ for the list E of edges in the mesh m, and a list L of links with $l_i \in L$ for $i \in [1, |L|]$, where $|L|$ is the amount of links in L.

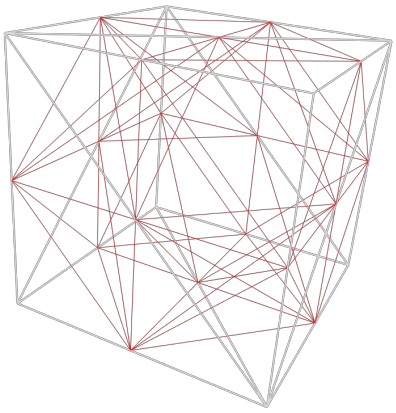


Figure 4.1: Triangulated Cube with Dual Graph in red. Although the underlying object is quite simple (8 vertices, 18 edges, 12 faces), the dual graph is already quite complicated (18 nodes, 66 links). This is the reason why in most illustrations, the dual graph is not shown, as this would unnecessarily complicate the illustrations.

This graph can now be used to calculate a path connecting two edges in the mesh by traversing g along the links.

To do this, however, we need to assign some cost value to each link as a function of the dihedral angles of the two edges, the angle between the two edges, the Euclidean lengths of the two edges and possibly of other properties. The idea is t calculate modifiers that get smaller for sharp dihedral angles, smooth angles between the two edges and for flat regions on either side of the two edges and larger for the opposite properties. We then multiply these modifiers with the Euclidean lengths of the edges. This makes edges with good feature properties to be shorter in the modified metric. With this property, we can then find shortest paths between two nodes using Dijkstra's algorithm, where the shortest paths tend to follow features.

To calculate this cost, we first calculate a modifier for each node n as a function of its dihedral angle φ. These modifiers get smaller for sharp angles and bigger for flat angles.

Notation

- $|n|_2$ = the Euclidean length of the edge represented by node n.
- n_1, n_2 the two nodes connected by a link l.

4.3 Calculation of Node Modifiers

Given the dihedral angle φ, the node modifier is now calculated the following way:

$$m(n) = \frac{\pi}{2 * \varphi} - \frac{1}{2}$$

To prevent a division by 0, the value of φ is clamped to a value range of $[\frac{\pi}{200}, \pi)$ by assigning all values of $\varphi < \frac{\pi}{200}$ to $\frac{\pi}{200}$. We could also allow division by 0 and define the result to be ∞. However, in this case, the modifiers of the links could also become ∞, which would pose a problem, as we want to allow the user to add features manually later, which relies on the Dijkstra algorithm to work on the whole mesh. If we had links with an infinite modifier, then Dijkstra's algorithm would not be able to find any path through this link.

Thus, the value range of this function is in $(\frac{1}{2}, 100]$. Now we subtract 0.5 to move the minimal value to 0. This offset will later be added again to the total modifier. This way, however, it is possible to set this offset manually as a parameter, as will also be described in section 4.9. The standard value for this offset is $\frac{1}{2}$, so, with this standard offset value, we can regard the value range of this function as mentioned above, but strictly speaking, it is actually in $(0, 99.5]$.

The upper limit of this function's range is chosen very high with the intention that the cost of a path following nodes, that are most probably not part of features as they have very flat dihedral angles, should be very high. Of course, the value 100 is chosen arbitrarily and there might be better choices for this limit.

4.4 Calculation of Link Modifiers

The rating of the links is slightly more difficult than the rating of the nodes, as this rating should depend on the following properties:

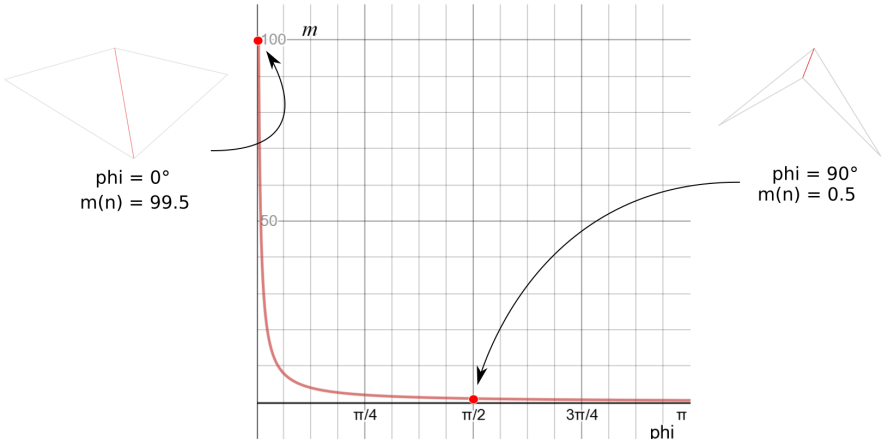


Figure 4.2: Node modifier: $m(n) = \frac{\pi}{2*\varphi} - \frac{1}{2}$

- The dihedral angles φ of both edges
- The angle θ between the two edges
- The flatness of the two triangle fans on either side of the common vertex
- The deviation of the dihedral angle

All of these four properties are first rated independently and a modifier for each of them is calculated. These four modifiers are then combined by a weighted sum to compute a general link modifier. Let's first look at the computation of each of the four basic modifiers:

- Sharpness modifier $m_{sh}(l)$ depending on the dihedral angles φ_1 and φ_2 of the two edges represented by the the two nodes n_1 and n_2.

  To calculate this modifier, we can directly access the already computed node modifiers $m(n_1)$ and $m(n_2)$, which only depend on the dihedral angles φ_1 and φ_2.

  The sharpness modifier of a link l is now just the weighted average of $m(n_1)$ and $m(n_2)$ with the Euclidean lengths of the according edges as weights. Thus, the formula for calculating $m_{sh}(l)$ is:

$$m_{sh}(l) = \frac{m(n_1) * |n_1|_2 + m(n_2) * |n_2|_2}{|n_1|_2 + |n_2|_2} \tag{4.1}$$

  Please refer to sections 4.3 and 3.2 for more information on the calculation of the sharpness modifier and the dihedral angle φ it is based on.

- Smoothness modifier $m_{sm}(l)$ depending on the angle θ between the two edges

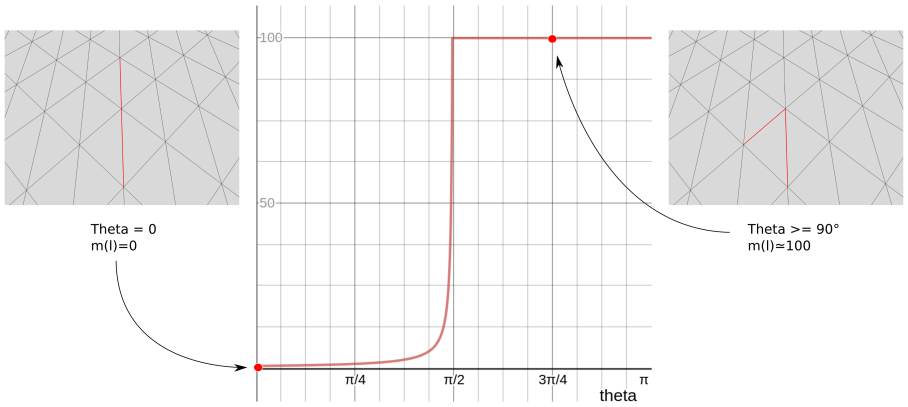


Figure 4.3: Smoothness modifier: $m_{sm}(l) = \frac{1}{\frac{\pi}{2}-\theta} - \frac{2}{\pi}$

Let θ be the angle between the two edges, where $\theta = 0$, iff the two edges are colinear but not coinciding, and $\theta = \pi$, iff the two edges are coinciding, which can never happen in a non-degenerate mesh. Note that a larger angle is not possible, as θ is defined as the smallest angle between the two edges, so $\theta \in [0, \pi)$. We then clamp all angles $\theta > \frac{\pi}{2} - \frac{1}{100}$ to $\frac{\pi}{2} - \frac{1}{100}$, such that $\theta \in [0, \frac{\pi}{2} - \frac{1}{100}]$. This allows us now to calculate the smoothness modifier $m_{sm}(l)$ as:

$$m_{sm}(l) = \frac{1}{\frac{\pi}{2} - \theta} - \frac{2}{\pi} \tag{4.2}$$

, where $m_{sm}(l) \in [0, 100 - \frac{2}{\pi}]$.

This function tends to ∞ as α tends to $\frac{\pi}{2}$. The intention is, that paths that take a 90° turn or sharper are very expensive. Note, that all $\theta \geq \frac{\pi}{2}$ get the same modifier. The intention behind this decision is that a feature path should never make a 90° turn or sharper. A 90° turn is already so bad, that it can hardly get worse, so the modifier does not get any worse either, even if the angle is sharper.

- Neighbourhood modifier $m_{ne}(l)$ depending on the smoothness of the triangle fans on either side of the two edges.

  The principle idea behind this modifier is that a feature should pass along the feature lines of a mesh, where this feature divides a local neighbourhood into two relatively flat patches. Here, we only look at the immediate neighbourhood, namely the one-ring neighbourhood of the vertex in between the two edges. In this neighbourhood, we look at the normals of each triangle connected to this central vertex and evaluate the deviation of these normals. Let $[n_1, n_2, ..., n_m]$ be the sequence of the normals of the triangles on one side of the two edges, ordered clockwise (or counterclockwise) with n_{n1} adjacent to one of the two edges and n_{nm} be adjacent to the other edge represented by n_1 and n_2 and with m normals of m faces

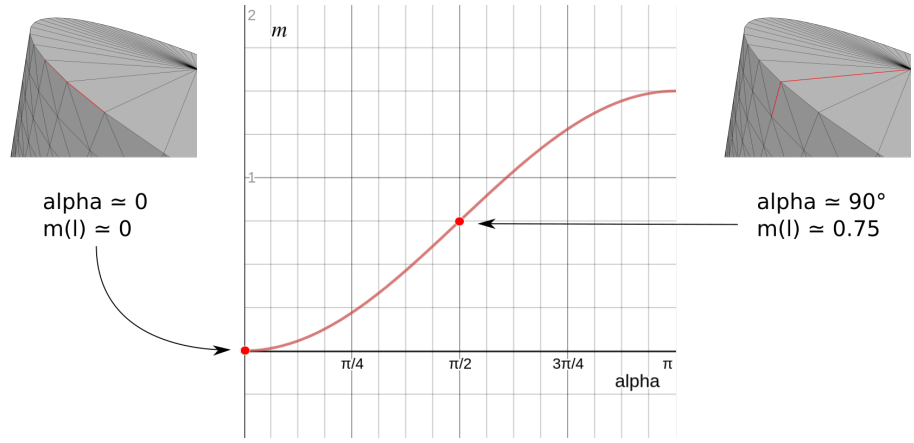


Figure 4.4: Neighbourhood modifier: $\frac{3}{4} * (1 - \cos\alpha)$

on this side of the two edges. We just consider the calculations leading to $m_{ne}(l)$ for the triangles on one side, the calculations for the other side are analogous.

Note that there might be any number m $\geq$ 1 of faces on either side of the two edges. If there is just one face on one side, then this patch is, in this local neighbourhood, absolutely flat, so the modifier will become 0, the smallest possible value. If there are several faces on this side, then we take a look at each pair of consecutive triangles, and the dihedral angle, they form. Let $\alpha_1, \alpha_2, ..., \alpha_{m-1}$ be the dihedral angles between these pairs of triangles. Then the neighbourhood modifier $m_{ne}(l)$ is calculated the following way:

$$m_{ne}(l) = \frac{\sum_{i=1}^{m-1} (\frac{3}{4} * (1 - \cos\alpha_i))}{m - 1} \tag{4.3}$$

Note: $\alpha \in [0, \pi) \rightarrow 1 - \cos\alpha \in [0, 2) \rightarrow m_{ne}(l) \in [0, \frac{3}{2})$

The choice of this function was made in order to make sure that the minimal value of m_{ne} is 0 as for the other modifiers.

This is now calculated for both sides, resulting in the two modifiers $m_{ne1}(l)$ and $m_{ne2}(l)$, which we then can average to the final neighbourhood modifier $m_{ne}(l)$.

We could also have chosen to consider a bigger neighbourhood, e.g. the 2-ring, which might lead to better results at the cost of longer computing times. We did not try this out, but this might well be considered in future improvements.

- Angle Deviation modifier $m_{ad}(l)$ depending on the similarity of the dihedral angles of the two connected edges.

  In a relatively smooth and sharp feature, the dihedral angles φ_1 and φ_2 of two consecutive edges e_1 and e_2 are probably very similar. If they weren't, this would mean that at least on one side of these two edges, the normals of

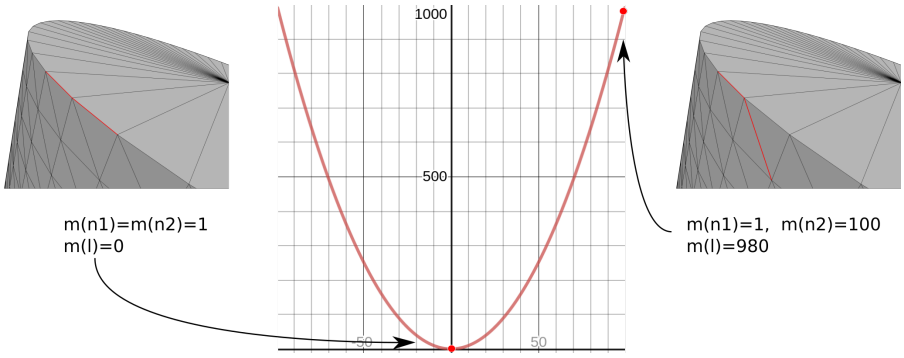


Figure 4.5: Angle deviation modifier: $\frac{1}{10} * (m(n_i) - m(n_j))^2$

the two adjacent triangles strongly vary. This is a hint that these two edges are not in the same feature. This is already covered by the neighbourhood modifier to some extent, however, there are situations, where two edges clearly do not belong to the same feature that are not well covered by the neighbourhood modifier, e.g. when the dihedral angles of the two edges deviate strongly but the dihedral angles on the one ring triangle fan on both sides only deviate gradually. On the other hand, there are situations, where there are large dihedral angles in the neighbourhood but the dihedral angles of the two linked edges are similar. So both modifiers have a legitimate reason to be included, even if there is some overlap in many situations.

We calculate the angle deviation modifier the following way:

$$m_{ad}(l) = \frac{1}{10} * (m(n_i) - m(n_j))^2 \tag{4.4}$$

,where l is the link connecting the nodes n_i and n_j and $m(n_i)$ is the modifier of node n_i. As $m(n_i) \in (0, 100 - \frac{1}{2}]$, we have that $m_{ad}(l) \in [0, 990.025)$.

We now have these four modifiers for each link, so that we can just take the weighted sum of them to calculate the final link modifier $m(l)$. The standard weights to calculate the weighted sum are the following:

- Neighbourhood modifier: $w_{ne} = 0.2$
- Sharpness modifier: $w_{sh} = 0.4$
- Smoothness modifier: $w_{sm} = 0.4$
- Angle Deviation modifier: $w_{ad} = 1$

However, these weights can be adjusted by the user. Thus, the formula to calculate the final link modifier is:

$$m(l) = w_{ne} * m_{ne}(l) + w_{sh} * m_{sh}(l) + w_{sm} * m_{sm}(l) + w_{ad} * m_{ad}(l) \tag{4.5}$$

The choice of the modifiers has a direct influence on which nodes will be detected as seed edges and how these seed edges will be extended. For a more

complete explanation on the effects of changing these parameters please refer to section 4.9.

Calculation of the shortest path between two edges

An integral part of the whole algorithm is to be able to find paths running along features. To do this, we calculate a modified virtual length $|l|_{mv}$ for each link in the graph. This modified virtual length is just the sum of the Euclidean lengths of the two edges multiplied by the link modifier:

$$|l|_{mv} = (|n_1|_2 + |n_2|_2) * m_l \tag{4.6}$$

This allows us now to run Dijkstra's algorithm to find the best path between two edges, which minimizes the Euclidean length of this path and tries to follow features at the same time. An attentive reader might remark that with this formula, the Euclidean lengths of all internal edges appear twice while those of the two edges at the ends appear only once. This is correct and while there is an irregularity, it does not matter, as, in practice, we use this value only to compare paths, where the same irregularity applies to all paths. The absolute value has no meaning and is never used, just the relative values are used for comparison with other links.

4.5 Detection of Seed Edges

The general idea behind our detection method for good seed edges is that the shortest path between two edges, wrt. the modified virtual lengths $|l|_{mv}$ of the links it passes through, generally passes through a lot of edges that are probably good feature edges.

For each node in the graph, we run Dijkstra's algorithm to find the shortest paths, wrt. the modified link lengths starting from this node up to a given maximal modified length. Of all the paths that were found, we choose the one with the longest Euclidean length as the best path from this node. The better a path follows along feature lines, the smaller get the modifiers and thus, the further this path can continue until it reaches the maximal modified length. So the longest path wrt. the Euclidean length is probably a path following along features.

We find such a path for each node. Let such a path for node n_i be $p_i = (l_{i1}, l_{i2}, ..., l_{in}) \mathrel{\widehat{=}} (n_{i1}, n_{i2}, n_{i3}, ..., n_{in}, n_{in+1})$, where $l_{i1} = (n_{i1}, n_{i2}), l_{i2} = (n_{i2}, n_{i3}), ...,$ $l_{in} = (n_{in}, n_{in+1})$ and let $|p_i|_2 = \sum_{j=1}^{n+1} |n_{ij}|_2$ be the total Euclidean length of the path. We then rate each node by adding up the Euclidean lengths $|p_i|_2$ of all such paths passing through this node. We do this with the idea that good feature edges are more likely to be in a lot of such paths, and that paths passing through features tend to be longer wrt. the Euclidean length. This way, edges in features get a lot higher rating than edges that are not in features, first because they tend to be in more such paths and second because those paths tend to be longer wrt. the Euclidean length.

We now have all nodes rated and can select the one with the best rating to be the first seed edge.

Algorithm 3 Rate Nodes

```
for n ∈ N do
  n.rating ← 0
end for
for n ∈ N do
  run Dijkstra until maximal modified virtual length reached
  path_max ← longest path in visited set w.r.t. the Euclidean length
  for n_p ∈ path_max do
    n_p.rating+ = l_max
  end for
end for
```

4.5.1 A remark about Dijkstra's Algorithm

When running Dijkstra's algorithm on a graph with a maximal path length as an abort condition, it is possible to program the algorithm in a way, such that the algorithm stops *after* having exceeded the maximal length or *before* exceeding the maximal length. We chose to implement the Dijkstra in the latter way as this is generally the faster option and as we noticed that for relatively small maximal lengths, the results are better when the Dijkstra stops before exceeding the maximal length. The bigger the maximal length gets, the smaller gets the difference between the two options. With this choice, however, it is possible that some nodes get a rating of 0, which might not be desirable. This happens if all links connected to this node have a modified virtual length greater than the maximal path length (extension depth), in which case no path with length shorter than the maximal path length can ever pass through this node. For this reason, we added an option to run the Dijkstra in the former way. In this case, at least the longest path from the node itself will have a length strictly greater than 0 and there will be no nodes with a 0 rating. In general however, nodes that would get a 0 rating with the latter option are very unlikely to be part of any feature, so it might even be desirable for them to have a 0 rating. Of course, if the maximal path length is set to a great enough value, then also nodes cannot have a rating of 0, but with the Dijkstra run in the former way, this is guaranteed. In all the examples in the 'Results' section and also all other results, we used the latter mode, which is also the standard setting.

4.6 Extension of Seed Edges

Now that we have a seed edge, we extend this edge in both directions. For the actual extension, there are a few possible approaches on how to extend them exactly. We chose a greedy approach, as we have seen during the development of this procedure, that the results of a greedy approach are almost as good as any other approach we tried and it is much faster than any other approach. The greedy version is very simple and just extends the path by the best rated link wrt. the link modifier. We perform this node by node in two directions, one node in each direction at a time. However, we take the current direction of the path into account, i.e., we don't allow the path to take a 180° turn.

With this approach, however, we need an abort condition, otherwise the path

would grow indefinitely. This abort condition is discussed in the next section.

Of course a non greedy approach would also be possible and might even perform better under certain circumstances. Another possibility that we tried to follow before settling on the greedy approach is to find a feature path that passes through the seed edge and that follows the feature until it either vanishes or meets another feature. This approach, however turned out to be very difficult to implement, as we would need to rate complete paths in order to decide which of all the possible paths running through a given seed edge is the best. Such a rating, however, is very difficult to implement, as it would need to take into account several different properties such as the total length of the path as well as the sharpness and other properties, some of which we described in the sections 4.3 and 4.4. For a good feature path, we would want it to be as long and straight as possible and as sharp as possible at the same time. If we tried to calculate such a rating with the modifier approach we have introduced in this chapter, then we would need to find paths that maximize the total length and minimize the total or average modifiers at the same time. This is difficult as longer paths, wrt. the amount of edges in it, in general would have a greater total length but also a greater average of the modifiers. So if we put too much weight on the total length of the path, the algorithm might come up with very long paths that possibly go around the whole mesh, but contain some really bad edges, and if we put too much weight on the feature properties, such as the sharpness, the algorithm might come up with a lot of very short paths with just one or two edges. While we were not able to come up with such a rating for feature paths, having such a rating would be very nice to have, as it would allow us to improve many aspects of this project, as we also describe in chapter 6. Apart from that, it is also difficult to find possible feature paths running through a node in a non greedy way. While it is easy to find a shortest path from one node to another w.r.t. the virtual modified length, we did not find a way to come up with a good feature path passing through one specific node in a non greedy way. In conclusion, there are some very difficult problems we would have to solve to follow this approach, if we want it to deliver results that are considerably better than with our greedy approach. Apart from that, the time needed to find such paths with our greedy approach is very short, while a non greedy approach would probably take a lot more time. This aspect was quite an important one, when we decided to follow the greedy approach. In the course of the development, however, we managed to make large parts of the algorithm considerably faster, so now, such an approach would be feasible again. Because of these difficulties and because we saw that the greedy approach delivered quite good results, we went on with the greedy approach.

4.7 Creating Cyclic Features

As discussed in the previous section, we are now able to extend a given path on both ends by one additional edge. We repeat this, until we get a cyclic graph. This means, that the path, we're currently working on, forms a closed cycle, or is connected at both ends to an already found feature from a previous step. That way, each end edge connects to a vertex that is already in a feature. Of course, the very first feature we find, needs to build a closed cycle, as there are no other features yet.

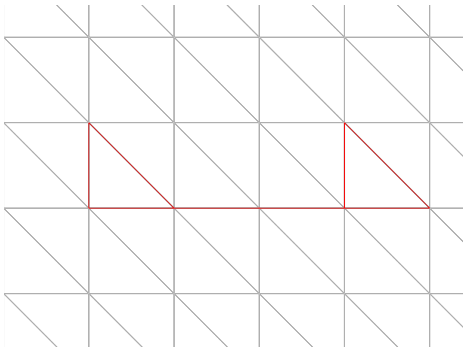


Figure 4.6: This is also regarded as a cyclic path

This idea comes from the observation that features often (almost always) form a closed cyclic graph. The only exception to this are shallowing features, which will be discussed later in this section. In all other cases, features connect to other features in feature vertices or form a closed cycle.

It is also possible that the path, we're working on, doesn't strictly speaking form a closed circle nor connect to another feature on both ends. This may happen, if in one direction, the path forms a relatively small cycle and connects to itself, but not to the other end of the cycle. This might happen, if there actually is such a feature, where the path is following along. Another case, however, in which we observed such a behaviour, are shallowing features. In such cases, a seed edge might be found in a region where the feature is still a strong feature and gets extended in both directions. In the shallowing direction, the rating of the links gets worse and worse as the dihedral angles get flatter and flatter. In such cases, it can happen that after some point, the rating of the links in the most obvious direction gets so bad due to the really flat dihedral angles, that the rating of other links is better. In this case, it often happens, that the path forms some sort of small cycle reconnecting to a previous part of the path.

In this case, however, the shallowing feature should actually stop at some point. It is unclear, at which point exactly it should end, as this exact point might be ambiguous. A possible solution for that problem would be to detect such small self-connecting cycles, not consider them as features, and let the feature end in the vertex, where the self-connection happened. However, this would violate the idea of having only cyclic features, so, in such cases, we let the feature path continue and let the user later decide, what to do with these cases with the manual refinement tools.

4.8 Repeating the Previous Steps Until All Features Are Found

With the previous three steps, we are now able to extend our feature graph by closed cycles. We can repeat this until we have found all the relevant features. To do this, however, we need an abort condition to decide, when we have found all the relevant features. If this abort condition is too strict, there might be

some features that have not yet been found but are clearly features. If the abort condition is too loose, the algorithm might consider some paths, that are not actually features.

One possible approach for this abort condition, is for the user to define some percentage of edges that need to be in features. With this approach, the algorithm would continue to add cyclic features, until at least the specified amount of edges are in features. One drawback of this approach is that some clear features might be only added after some less clear features or noise, just because the seed edge of the former feature has a worse rating than the one of the latter feature. In this case, it would be impossible to add the better feature while not adding the worse one. Also, it might not be a very intuitive user input.

A similar approach is to let the user define the amount of features that are considered rather than the percentage of edges in features. This way, it is easier for the user to find the optimal solution by trying some values and settling at a value, for which increasing and decreasing it by one both return worse results. On the other hand, however, it might be even more difficult to get an intuition for the optimal value than with the previous approach.

Another approach would be to rate the path to be added to the feature network before adding it and only adding it, if it gets a better rating than a given threshold. One drawback of this approach is, that we need another threshold, which might be hard to set. Also, the rating of feature path candidates is not straight forward, as several properties would have to be considered, as described in 4.6. An advantage might be that we might be able to skip some badly rated candidates and still add some candidates that get found later.

We decided to go with the third approach (user chooses amount of features), as it does not depend on the ability to rate feature paths, which turned out to be very difficult, but at the same time is relatively user-friendly. While it might be hard to find a good value for the amount of features initially, it is not hard to try out some values until a satisfying result has been found. For this, of course, the program needs to be able to find new feature paths as fast as possible, to prevent waiting times between trying out different values. Our program satisfies this requirement, at least for relatively small meshes. On meshes with up to a few 100'000 edges, finding a new feature path only takes a fraction of a second. On bigger meshes, we often also need to find more features, so the time needed to find all features might take a few seconds. On significantly bigger meshes, this time might increase to time spans that don't qualify anymore to be called an instantaneous calculation. However, all features that have been found are stored separately and don't get deleted if the user decreases the amount of features parameter. So, all feature paths have to be found only once, after which increasing or decreasing the amount of features only changes the appearance of the mesh but not the underlying data. This way, once all relevant features have been found, trying out different values for the amount of features parameter becomes an instantaneous operation again. Please refer to section 5.6 for a more thorough analysis of the time complexity.

4.9 Parameters

The initial goal for this project was to have as few parameters as possible, the optimal case would be an application with just one button to initiate the feature detection, which would then find all features and find no false positives. In the real world, however, meshes can vary in a lot of ways and finding an algorithm with no parameters to be set by the user, is very difficult or might even be impossible.

In our final application, there are seven parameters, which can be grouped into two classes: (1) Modifier weights, extension depth and modifier offset to prioritize different aspects of features over others and (2) amount of features to find a reasonable abort condition for adding new features. Apart from these parameters there is also the option to run the Dijkstra in a different mode when finding seed edges, called Dijkstra abort after. For more information about the effects of this option, please refer to section 4.5.1. We will now shortly discuss these parameters.

4.9.1 Modifier Weights, Extension Depth and Modifier Offset

The four modifier weights for the sharpness, smoothness, neighbourhood and angle deviation directly have an influence on the calculation of the link modifiers as described in equation 4.5 in section 4.4. The standard values of 0.4, 0.4, 0.2 and 1 for the sharpness, smoothness, neighbourhood and angle deviation modifier weights respectively have provided relative good results, as can be seen in chapter 5.

These parameters can be changed by the user to put more emphasis on some of these aspects. If changing these values, however, the user should keep in mind that these weights are absolute weights in the calculation of the link modifiers and not relative weights, as can be seen in the aforementioned equation 4.5. Because of this, if these weights are changed in a way that their total sum is altered, the average modified virtual links will be different as well. If, at the same time, the extension depth is not adapted, the behaviour of the Dijkstra algorithm while detecting seed edges, can be very different than expected, which might lead to a bad result. For example, if the modifiers are all set to very high values, while the extension depth has a rather low value, then the modified virtual lengths of the links become very long and the Dijkstra algorithm is aborted much earlier than usually. This might lead to a lot of nodes (in the most extreme case all of them) to have a rating of 0. In this case the seed edges are chosen arbitrarily which leads to an apparently very bad choice of seed edges. On the other hand, however, if the extension depth is much larger than the modifier weights, the modified virtual lengths become very short and the Dijkstra algorithm is allowed to go very far, which can lead to much longer computing times. Also, short features are more difficult to find in this case. We found that an extension depth of about 2-3 times the total sum of the modifiers works best. This means that, on average, the Dijkstra finds paths containing approximately 2-3 links.

In most cases, the standard values should deliver acceptable results, so these parameters need only be changed on rare occasions, so a deep understanding

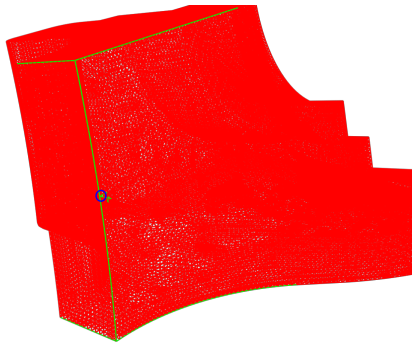


Figure 4.7: Green: all edges that were visited by Dijkstra's algorithm with an extension depth of 100. The origin is approximately at the center of the blue circle

of the workings of this algorithm, especially of the underlying dual graph data structure, is normally not needed to operate this program.

The extension depth parameter defines the maximal distance wrt. the modified virtual length that Dijkstra's algorithm covers during the node rating process. The actual maximal distance wrt. the modified virtual length that Dijkstra's algorithm covers is calculated as this parameter multiplied with the average edge length of the mesh. If we expect the average modifier to be around 1, then this means that during the node rating process, Dijkstra's algorithm covers approximately the area that can be reached within n edges, where n is the extension depth parameter. Note that in practice, this distance can be highly variable, as the modifiers can become very large or small, even 0.

With the modifier offset, a minimal value for all modifiers can be set. Originally, this was directly included in the calculation of all parts of the modifier. However, we decided to externalize this value, so that it can be chosen by the user. Now, the minimal value for all modifiers is 0, so the total link modifier can also become 0. However, this might lead to unexpected results, as it is then possible for a feature path to have a total length of 0, regardless of the Euclidean distance it spans. Also, this would mean that the Euclidean lengths of the edges have no influence on the virtual modified length of links with a modifier of 0. So, the standard value for this offset is set to 0.5. Increasing this value puts more weight on the Euclidean lengths of the edges and less on the feature properties. If changing this parameter, keep in mind, that this also alters the average modified virtual lengths of the links, so the extension depth parameter should be adapted accordingly.

4.9.2 Amount of Features

This parameter is used as an abort condition for adding new features. A better solution would be to rate features based on their feature qualities and then abort adding new features as soon as there are no more features with a good enough rating. However, we did not find such a rating, as explained in the conclusion of section 4.4, but it would be a very nice addition to this approach, as explained in section 6. Due to the lack of such a rating, we need some alternative abort condition, which is provided by this parameter. The value of this parameter

determines the amount of cyclic feature paths that are added one after another to the feature network. The order in which these features are added cannot be altered, as it is determined by the rating of the seed edges as described in section 4.5.

4.10 Internal Data Structure to Store Features and Work Flow

Internally, features are stored as paths passing through nodes in the dual graph. Each of these paths consists of a list of node indices and a list of link indices referring to the nodes and links in the dual graph. These paths are stored in two parallel lists: (1) a list containing all features found by the automatic detection and (2) a list containing all features that are currently displayed. By increasing or decreasing the amount of features parameter to n, the first n paths in list 1 are copied to list 2 and list 1 is updated, i.e. a new feature has to be found, iff more features are to be displayed than have been found so far. This means that a feature has to be found only once, even if the user decreases and increases the amount of features parameter over and over again. This significantly increases the usability even on large meshes and even if a lot of features have to be detected, as trying out different values for this parameter is possible to do in real time. Only when starting the automatic detection, some delay might occur on large meshes. The features in the first list, containing all feature paths found so far, are organized in a way, such that no path passes through a feature corner with valence ≥ 3. This means, that wherever three features meet, they get split up in three different features and are stored in this list as three different paths. This makes the deletion of features with the according manual tool more intuitive. See sections 4.11.3 and 4.11.4 for clarification. In principle, this means, that there are no feature corners with valence 2 that split up features. However, in a circle that does not connect to any other feature, there is a valence 2 feature corner where this path starts and ends. Unless this feature is split up manually afterwards, however, this is not noticeable. The manual refinement exclusively works on the second list, containing the currently displayed feature paths. This way, even after some manual alterations have been performed, changing the amount of features parameter should not result in long delays. On the other hand, however, this also means, that changing the amount of features parameter discards all manual alterations performed so far. With this in mind, the optimal work flow is to first fine tune the parameters of the automatic detection, until a relatively satisfying result has been achieved and then improving this first draft with the manual tools and not changing any parameters anymore, as this would result in a loss of the so far performed manual improvements.

However, list 2 has a history, meaning that it is possible to return it to a previous state with the undo and redo buttons. So, even if by accident the user changes the amount of features parameter leading to a loss of the manual improvements, this can be undone and the manual improvements recovered.

If any of the other parameters are changed, all modifiers have to be recalculated. Because of this, the two lists described above as well as the working history are cleared.

4.11 Manual Refinement

The main focus of this work has been to create an automatic feature detection that delivers as good a result as possible with as few clicks as possible in as little time as possible. However, such a first draft might still include some false positives, miss some false negatives, or some of the found features might not exactly meet the users expectations. For these cases we included a few tools to enable the user to correct these errors as quickly as possible and with as few clicks as possible. Some of these tools use some of the data calculated in the automatic part, in particular, the dual graph and the modifiers. So this part of the program might seem independent of the automatic part, but it is not, and it requires the precalculations of the automatic part to be computed before it can be used. We tried to limit the amount of tools as much as possible, to make it as easy to use as possible. Following is a description of the four tools we provide with a short description of the functionality, followed by a short description of the underlying mechanics to better understand their working.

4.11.1 Draw new Feature

With this tool, the user can add a new feature by selecting a few control edges. Clicking on an edge starts a new path and enables the preview of the feature path to the edge the mouse is hovering over. Clicking on another edge fixes the path to that edge and allows to continue the path from this edge onward. Clicking on the same edge twice or creating a closed cycle ends this path, which is then treated as a path found by the automatic part. Pressing 'u' on the keyboard undoes the last click and allows the user to continue defining the feature from the previous edge. Pressing 'u' when only one edge has been selected, allows the user to select a new edge to start a feature path. This mode is especially useful to find longer features, running along parts that might not be very clear features or defining the exact end of a shallowing feature.

The shortest path between two edges is found using the Dijkstra algorithm on the modified virtual length of the links in the dual graph, see section 4.4 for more information about this. So, the path between two edges should follow along features while being as short as possible. In this case, the Dijkstra has to be calculated only once, when clicking on a new edge. This might take some time, which might result in a delay right after the click, but, since the Dijkstra algorithm is very fast, this delay only gets noticeable for very large meshes with more than a few 100'000 edges. After the Dijkstra algorithm has been performed, finding the shortest path is very fast and should not lead to noticeable delays, even on very large meshes.

4.11.2 Add new Feature

This tool is similar to the previously described one, but it allows adding short and clear features with fewer clicks than the other tool, while it is less useful for adding complex features. Only one edge has to be selected, and a feature path passing through this edge is then proposed to the user. This proposal cannot be altered and is displayed in blue. After a click by the user, this proposal is fixed and thereafter treated as a path found by the automatic part.

To find such a path running through a specific edge, the same algorithm as for the extension of seed edges of the automatic part is used. See section 4.6 for a more thorough explanation of that procedure.

4.11.3 Delete Feature

This tool allows the user to quickly delete any unnecessary features with only one click. By hovering over a detected feature, this feature gets colored blue, by clicking on it, it gets deleted. A feature path starts and ends in a feature node with valence ≥ 3 or 1 or in any point defined by the split feature tool. See section 4.10 for an explanation of the underlying data structure.

4.11.4 Split Feature

This tool allows the user to split a feature path at a certain vertex into two separate paths. This modification is not immediately visible on the mesh, however, it allows the user to delete only a part of a feature. Other than for deleting features, this alteration has no consequences. No tool is provided to merge two paths, as the only reason to do this, would be to be able to delete both paths at the same time. In this case, however, providing such a tool would not make the process of deleting both paths any faster than just deleting one path after the other, so it would essentially be superfluous.

Chapter 5

Results

5.1 Introduction to Analysis of Results

In this chapter, we compare the results of our algorithm to the features as they are perceived by humans as the desired result, and, at the same time to the results of a trivial feature detection algorithm. The trivial feature detection algorithm we consider here, is an algorithm that just marks all edges of a mesh as feature edges, if their dihedral angle is sharper than a given threshold φ. This algorithm was already described in the introduction as algorithm 1. We take a look into different aspects of the resulting program of this thesis and into some special cases that pose difficulties in finding features. We define two cases of possible errors: false positives and false negatives. These two classes of errors are quite straight forward in the case of feature detection: false positives are edges that were detected by the algorithm to be features, where, from a human perspective, there is actually no feature. False negatives are features, that are indeed features from a human perspective but that the program has failed to detect.

We will also compare the results of our algorithm to the simple algorithm and will use the amount of clicks it takes a user to improve the result to achieve the optimal result as a metric to compare the usability of the two algorithms. When we count clicks, we only count the necessary clicks directly on the mesh. Clicks, dragging the mouse or pressing buttons on the keyboard to adjust the view or select tools are not counted. This, however, gives an unfair advantage to our plugin, as a user would probably need to change tools several times during the manual improvements, where changing a tool also needs one click on the corresponding button. The alternative without our plugin would be to select single edges. This requires far more clicks in general, however, as there is only one tool, there will be fewer clicks for changing the tools. So, to fairly compare the two alternatives and get an impression of which alternative is faster, one should keep this in mind and multiply the necessary clicks for our plugin by 2. However, in all examples, our plugin needs by far fewer clicks than the alternative, so even when we perform this adjustment, our plugin is still the far better option. We will not comment on this again, henceforth.

5.2 Automatic Detection on some Meshes with Default Parameters and Comparison to Trivial Algorithm

5.2.1 Fandisk

As the first example, we take a look at the fandisk model. The mesh we used here contains 25'994 Vertices, 77'976 Edges and 51'984 Faces. Although this is already a lot of data, our algorithm still works instantaneous. Given that the simple dihedral angle algorithm is much simpler than our algorithm, it also works instantaneous.

Let us first have a closer look at the fandisk and identify the features that should be found, those that might be found and regions where no features should be found. 5.1 shows some of these regions.

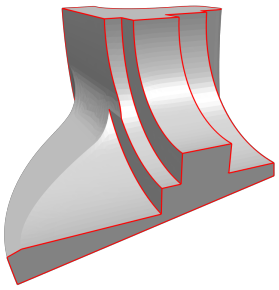
In figure 5.1a, we see all the features on the fandisk that are obvious features and that should definitely be found by any feature detector algorithm. 5.1b shows a shallow feature, that might be more difficult to find, but that still definitely can be classified as a feature. 5.1c highlights a shallowing feature. This feature should be detected, although it is not unambiguously clear, where it stops or if it stops at all or just continues, until it meets another feature. We decided to let it stop at a position where the dihedral angle has become quite flat. 5.1d shows a short and shallow feature. This is especially difficult to find, as it combines two difficult characteristics. 5.1e shows all the above mentioned features and is the desired result. Finally, 5.1f shows some false positives that might occur at the back. These should not be found even though they have dihedral angles considerably sharper than most edges.

Now, let us have a look at the result of our algorithm and compare it to the result of the simple dihedral angle algorithm.

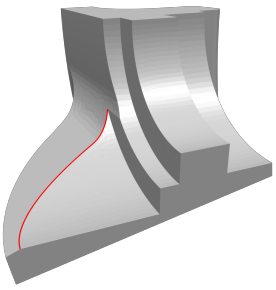
The simple algorithm with a threshold angle of 23°, as we can see in figure 5.2b, delivers an almost perfect result with no false positives. But a part of the shallow feature in the front is missing, consisting of 10 edges. Also, the shallowing feature on the left side suddenly stops. In this case, it is not unambiguously clear, where exactly the feature stops, or if it even stops at all. If we assume that the feature should continue until it meets the sharp feature, 10 edges would be missing. It also misses the short and shallow feature shown in 5.1d, consisting of 12 edges.

Correcting the two false negatives with the tools OpenFlipper provides would mean that the user would need to select all missing edges individually, taking 22 to 32 additional clicks.

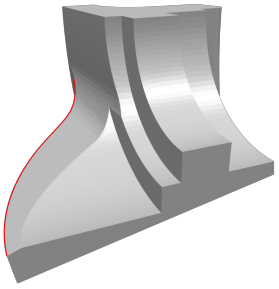
The result of our automatic detection with the standard parameters is, as we can see in figure 5.2a, identical to the result of the simple algorithm apart from three locations: The aforementioned shallowing feature, as well as the shallow feature are continued until they meet another feature here and there is an additional false positive passing through the bent area on the left side consisting of 32 edges. The short and shallow feature is still missing. To obtain a perfect result, we need to get rid of the new false positive, which can be done in one click with our manual tools, add the short shallow feature with one click and possibly shorten the shallowing feature, which takes one click to split up the



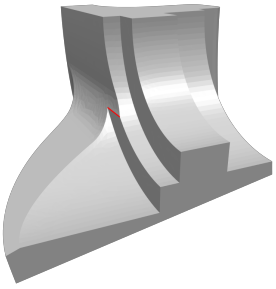
(a) Obvious Features



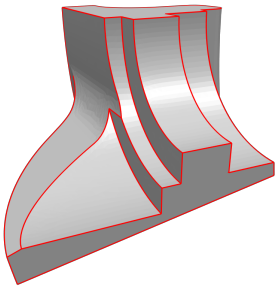
(b) Shallow Feature



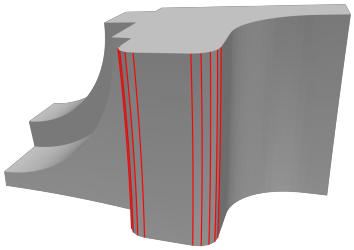
(c) Shallowing Feature



(d) Short and Shallow Feature

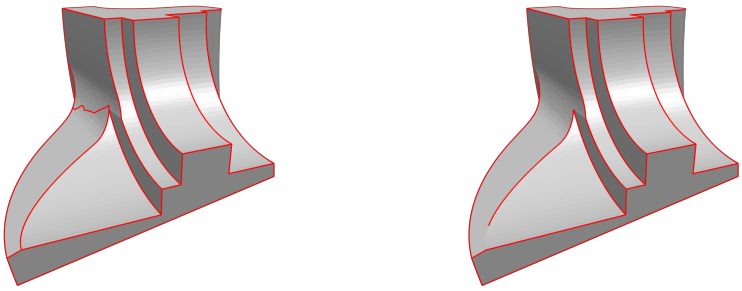


(e) Desired Result



(f) False Positives

Figure 5.1: Here we see different categories of features and possible regions where false positives might occur on the fandisk.



(a) Automatic Detection with 13 displayed features

(b) Simple Detection with a threshold angle of 23°.

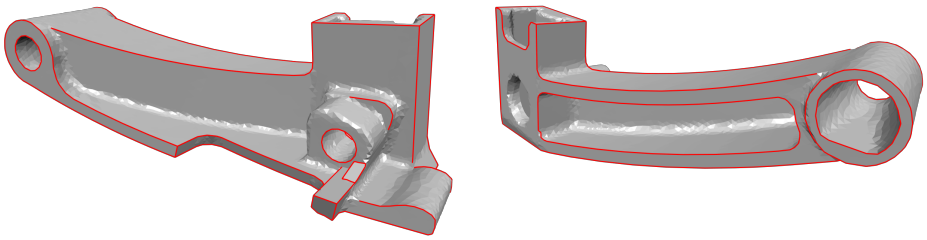
Figure 5.2: Comparison of the two algorithms on the fandisk

feature at the desired position and another click to delete part of it. Altogether, we need between two and four clicks to obtain a perfect result.

Compared to the 22 to 32 clicks needed with the simple algorithm, our 2 to 4 clicks are of course a lot better, however in large parts due to our simple manual refinement tools.

5.2.2 AlphaRem

The second example, we look at is the AlphaRem model. It consists of 10'002 vertices, 30'024 edges and 20'016 faces. Again, let us first have a closer look at this mesh to identify the features.



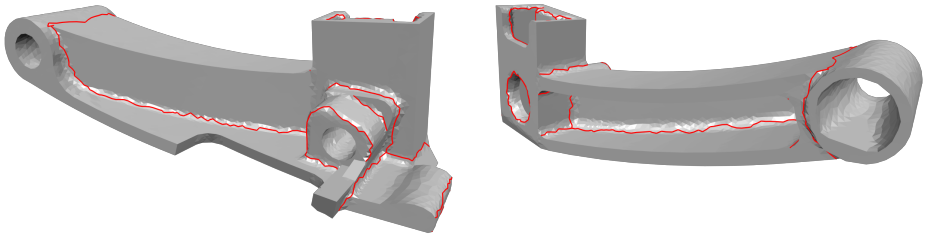
(a) Front

(b) Back

Figure 5.3: AlphaRem model with obvious features

In figure 5.3 we can see the AlphaRem model with the obvious features highlighted. There are still some features that are not so sharp as they are rounded off, but from a human perception, they would still be considered features.

In figure 5.4 we can see some of the less obvious features on the AlphaRem model. These are features that are rounded off, but probably still count as features. If one takes a closer look at the models, one can see that the features

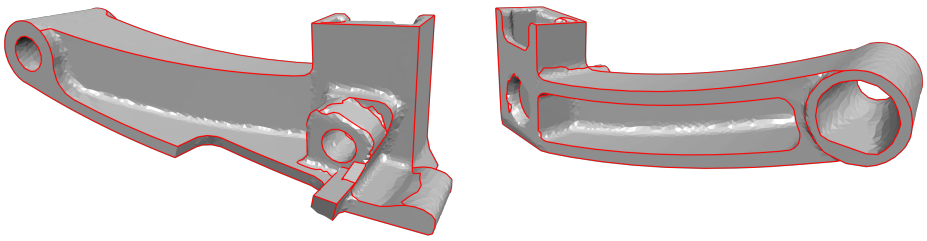


(a) Front (b) Back

Figure 5.4: Less obvious features on AlphaRem

are not so straight as the previous features we have seen. The exact course of the feature is also quite ambiguous, there are many possibilities how we can define the feature in these cases. What we show here in these figures is just one of these many possibilities. However, we would like a feature detector program to come up with a result that approximates this optimal solution.

Let us now compare the results of our automatic algorithm and the simple dihedral feature detector.



(a) Front (b) Back

Figure 5.5: Result of our automatic algorithm on the alphaRem model with 28 displayed features

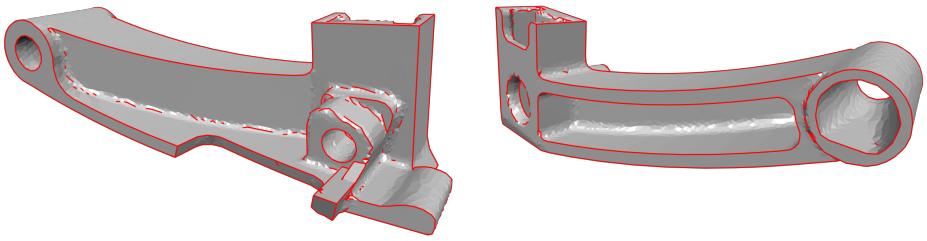
If we take a look at the result of our algorithm, as seen in figure 5.5, where we chose to display 28 features, we see a lot more differences to the optimal solution, than in the previous example. Our algorithm has found all features we categorized as "obvious", apart from one short feature seen in the front view in the lower center. Of course, the dead ends we had in the optimal solution were continued by our algorithm such that they connect to another feature, as dead ends are not allowed in our algorithm. Regarding the features we categorized as "less obvious", our algorithm has failed to find most of these. Some, however, have been found. It would take too long to describe all the similarities or

differences here, so we go without it here. However, we counted the differences and came up with the following statistics: We found 6 paths of false positives, consisting of 52 edges in total and 17 paths of false negatives, consisting of 293 edges in total. Of course, counting these errors, especially the false negatives, is extremely difficult, as the course of the feature is not clear in many cases and sometimes it is even not clear, whether there is actually a feature.

If we wanted to correct this result of the automatic feature detection, we would need to delete the 6 false positives, which would take 6 clicks with our delete tool, or rather 8 clicks, as we need to split up a feature on two occasions to prevent deleting too much. Adding the missing features takes about 11 clicks with the "add feature tool". It takes fewer clicks than we found false negatives, because there are cases, where we can add several features with one click, if for example, this feature that gets added, crosses a feature vertex where no other feature has been found so far.

In total, it takes 19 clicks to improve the result of the automatic part to a perfect result.

Now, let us compare this to the result of the simple algorithm.



(a) Front (b) Back

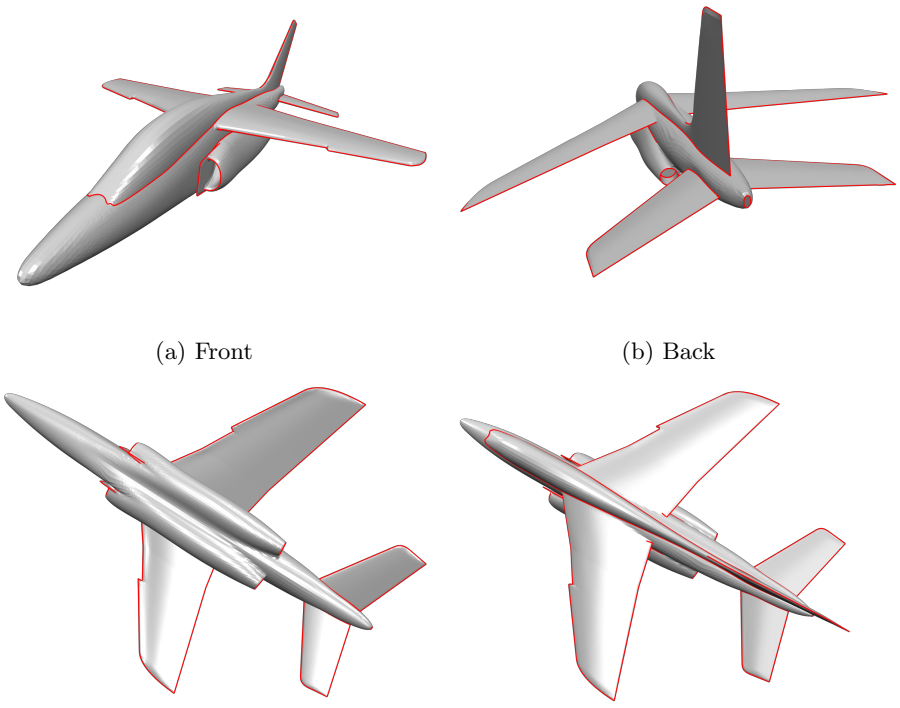
Figure 5.6: Result of the simple Algorithm on the AlphaRem model with a threshold angle of 35°

When we take a look at the result of the simple algorithm in figure 5.6, we can immediately see, that, while the clear features have been found pretty well, the less obvious features are very much less clear. We get a rough idea where these features approximately are, but the result is just a bunch of mostly unconnected single edges or short paths of edges, and we can also see many false positives. Again, counting false positives and false negatives is very difficult, so we only counted the most obvious false positives and found 87 such cases. We did not count the false negatives, there are just too many of them. We can spot immediately several features, where many edges are missing, so we can safely assume that there are by far more than 100 false negatives, probably several hundred. Again, without any other possibility to manually select all feature edges, in this case it would take far more than 100 clicks to achieve a perfect result. In this case, our plugin definitely allows the detection of all features much faster than the simple algorithm. In this example we also see that the result of our automatic detection is a lot better than the result of the simple

algorithm, so in this case, the improvement is not only due to the manual tools.

5.2.3 Alpha Jet

Thirdly we look at the alpha jet model, consisting of 36'473 vertices, 109'413 edges and 72'942 faces.



(a) Front (b) Back

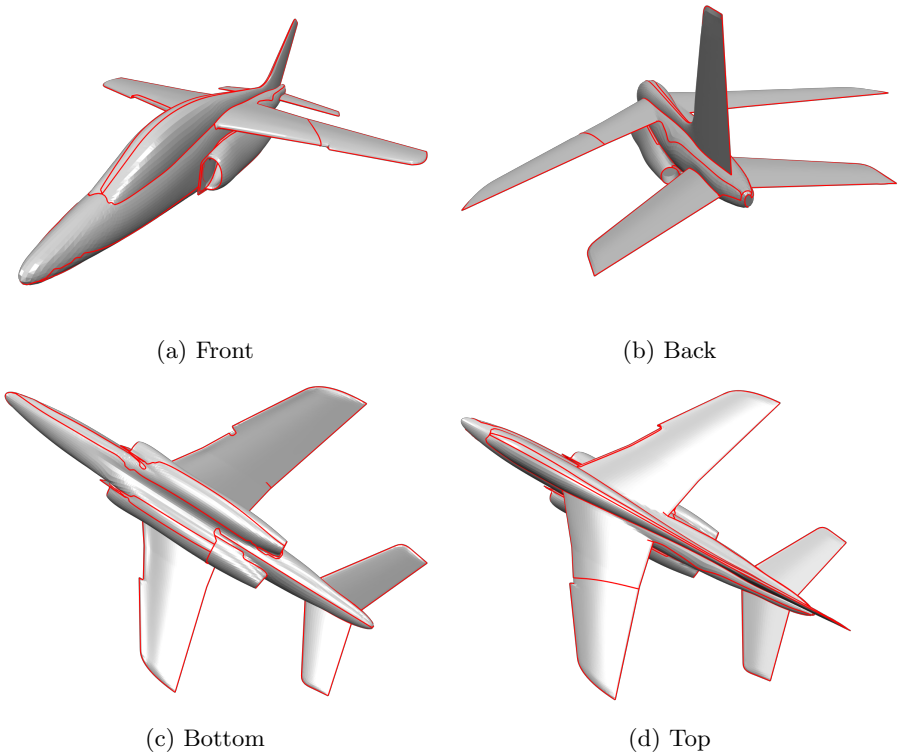
(c) Bottom (d) Top

Figure 5.7: Desired result on the alpha jet model

This model seems to be a lot more difficult to handle for our algorithm than the previous two models. We need to add 30 features in order to make sure that the most obvious features get detected. However, by that time, also a lot of false positives get detected. Counting all false positive edges would take too long, so we just note here, that there are about 25 paths of false positives, each containing many edges. However, thanks to our manual tools, the removal of these false positives should not take too long, at most 3 clicks per path (splitting the feature at both ends and deleting the middle part). So, even in this case, it should be possible to obtain the desired result with a few dozen clicks.

Let us now compare this result to the result of the simple algorithm:

Unsurprisingly, the simple algorithm did a good job in detecting the sharp features, e.g. on the wings, but failed to find the shallower features, e.g. around the cockpit. There are already some false positives, e.g. inside the engine, as can be seen in figure 5.9a. Again, most false positives are single edges or very short paths, making it very time-consuming to correct them. We found around 100 false positive edges without searching very thoroughly, so there might be



(a) Front (b) Back

(c) Bottom (d) Top

Figure 5.8: Result of our automatic detection on the alpha jet model

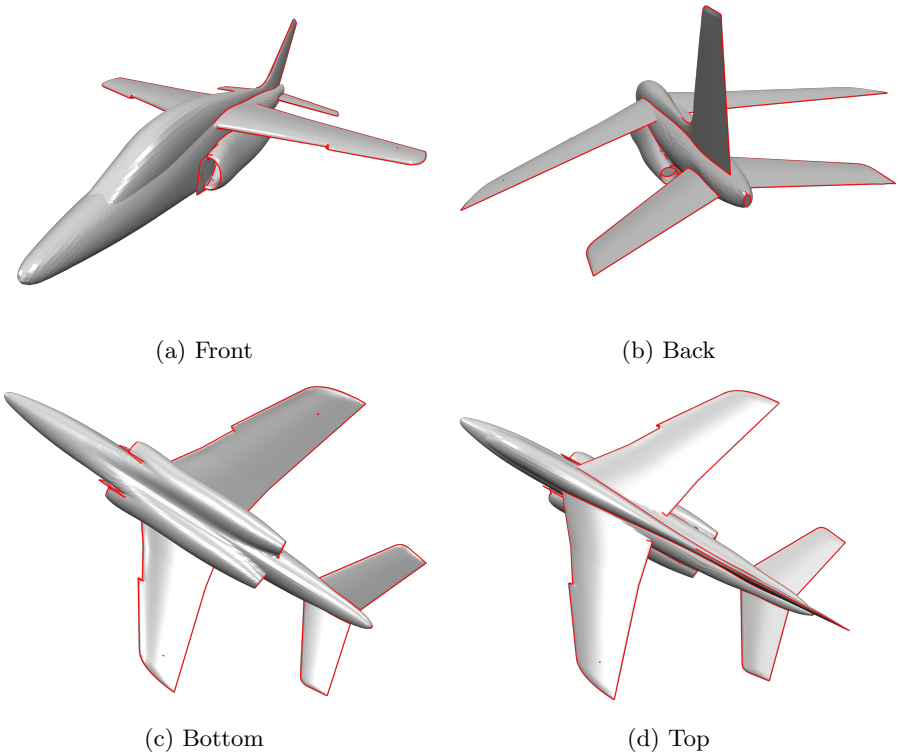
more. We also found a few false negatives, however not many. The prominent, though shallow, feature around the cockpit alone consists of about 160 edges and is missing in this result. So, even if there are not many false negative paths, there are still many false negative edges. To manually improve this result, at least 260 clicks are needed.

In conclusion, our algorithm still offers a better alternative for this model, even though the result of our automatic detection is not very good.

5.3 Best Achievable Automatic Result by Fine Tuning Parameters

5.3.1 Fandisk

If we finetune the parameters in order to have more weight on the sharpness and angle deviation modifiers than in the standard settings, the false positive we had in the previous example does not occur anymore. The parameters set in this case were 2, 0.5, 0.5, 1 and 8 for sharpness-, smoothness-, neighbourhood-, angle deviation modifiers and the extension depth respectively, as opposed to 0.4, 0.4, 0.2, 1 and 4 in the standard settings. This result is perfect apart from the short shallow feature shown in figure 5.1d.



(a) Front (b) Back

(c) Bottom (d) Top

Figure 5.9: Result of the simple algorithm on the alpha jet model

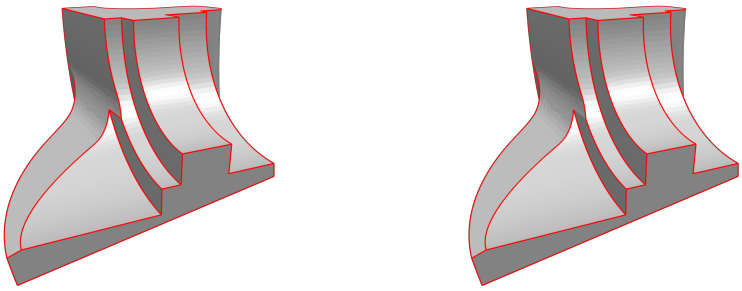
5.3.2 Box minus Sphere

In this example we used a mesh that consists of a perfectly regular cube, where on one face, a part of a sphere was removed. It consists of 26'161 vertices, 78'477 edges and 52'318 faces. The resulting feature along the intersection of the cube and the sphere is quite shallow with an angle of about 8°. As we can see in figure 5.11, both algorithms have found all the features with the right parameters. Our algorithm only found the shallow feature, if the extension depth parameter is set large enough. We set the parameters to 4, 4, 2, 1, 50 for the sharpness, smoothness, neighbourhood and angle deviation modifier weights and the extension depth, respectively. At the same time, for the simple algorithm, the threshold angle has to be somewhere between 1° and 7°.

5.4 Results in Special Cases

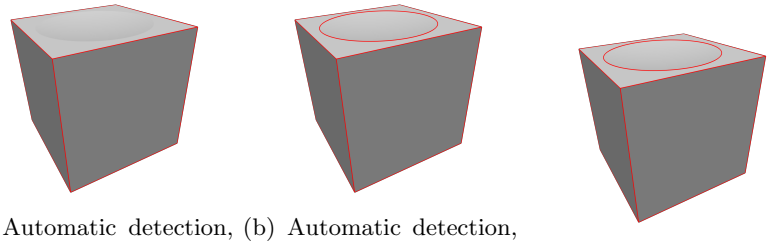
5.4.1 Shallow Features and Shallowing Features

As we have seen in 5.10a and 5.10b above, the two shallow features have been detected quite well by our algorithm. For most parts, the simple dihedral-angle algorithm discovers them just as well, however, in the case of the shallowing features, the discovered features from the simple algorithm stop at some point, whereas in our algorithm, the features continue until they meet with another



(a) Desired Result (b) Best achievable Result

Figure 5.10: Best achievable result on Fandisk



(a) Automatic detection, standard parameters (b) Automatic detection, better parameters (c) Simple detection

Figure 5.11: Automatic algorithm with different parameters and simple algorithm on Box minus Sphere

feature. It is difficult to say, which behaviour is better in this case, as it is not unambiguously clear, whether the feature stops at some point and if yes, at which point exactly it stops. This decision possibly also depends on the the application for which the features are needed. So, it is not possible to say, which algorithm does a better job in this case.

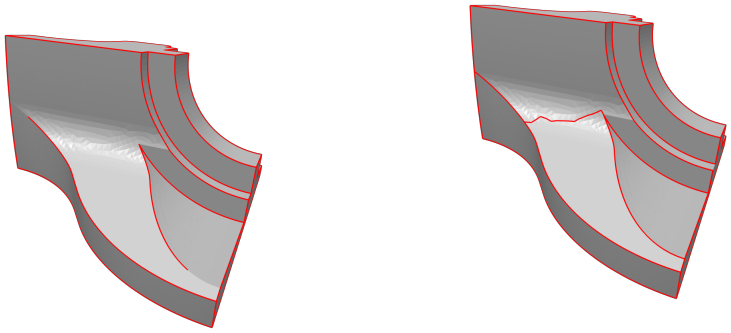
In the case of the box minus sphere model, shown in figure 5.11, both algorithms were able to detect the shallow feature given the right parameters.

Also, in regard of the user-friendliness, there is not a big difference, as in both cases, the user needs to provide some choices for parameters.

5.4.2 Short Features

If we take a look at the fandisk, we discover several relatively short features. Finding these features with the simple algorithm is no problem at all, as the length of a feature has absolutely no influence on finding the features. With our algorithm, however, this poses a difficulty, as the algorithm needs a seed edge in each of these features to find them, or they need to be detected from another feature next to the short feature. In 5.2a, we see, that all of these features were found.

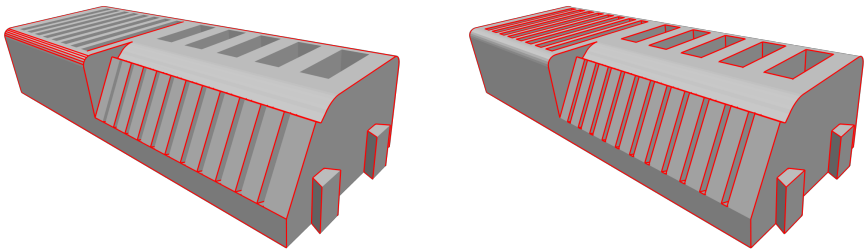
In this example, however, we see that our algorithm has some problems detecting short features. In the mesh micro usb holder, consisting of 484 vertices, 1'446 edges and 964 faces, there are many sharp features that only consist of one



(a) Simple Detection, 23 degrees, shallow features stop at some point

(b) Automatic detection, 13 Features, shallow features continue until they meet other feature

Figure 5.12: Comparison of the behaviour on shallowing features

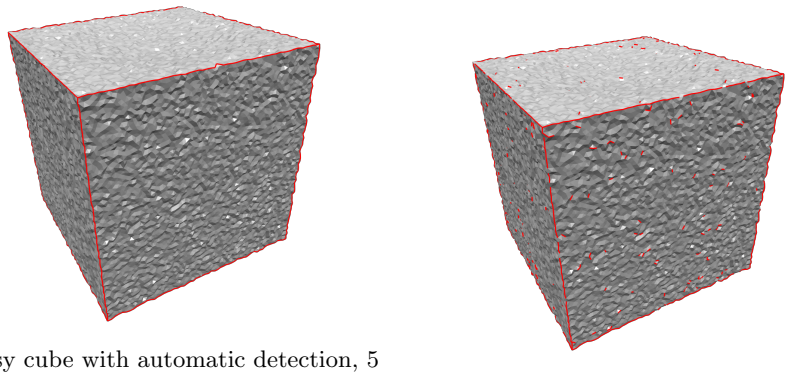


(a) Automatic, 73 features

(b) Simple algorithm, 45°

Figure 5.13: This example shows some limits of our algorithm, as the result of the simple algorithm is a lot better. This is mainly due to the irregular triangulation of the mesh.

edge. Our algorithm has not found all of them, even after 73 displayed features, while it has already found many false positive in the bent areas. The simple algorithm finds all of these features without problems, as it is not dependent on the feature length. This example shows some of the limits of our algorithm, there are cases, where it is definitely worse than the simple algorithm. The triangulation of this mesh is not at all uniform with many extremely long and sharp triangles. Such a triangulation is useful in such cases to decrease the amount of triangles to limit the amount of data needed to store the mesh. However, our algorithm works best on uniformely triangulated and finely tessellated meshes. If these requirements are not met, as it is the case here, our algorithm has a very poor performance.



(a) noisy cube with automatic detection, 5 features

(b) noisy cube with simple detection, 60°

Figure 5.14: Noisy cube with results of automatic and simple algorithms

5.4.3 Bent Surfaces

In 5.1f, the highlighted edges are not actually features, but just edges with a dihedral angle sharper than most other edges, because they are part of a curved region. There are quite a lot of such regions on the fandisk, where no features should be found. In 5.2a, we see that only one such possible false positive occurs. Of course, if we increased the amount of features to be found, at some point, also such edges would be detected as features. The result of the simple algorithm does not include any of these false positives, as long as the threshold angle is large enough. If we compare the result of our algorithm to the result of the simple algorithm, we can remark that both algorithms include such false positives, if the corresponding parameters are set accordingly. However, if a mesh contains a shallow feature with edges with angle $< \varphi$ and a bent region with edges with dihedral angles $> \varphi$ for any φ, then it is not possible to find no false positives and miss no false negatives with the simple algorithm, while our algorithm is able to achieve this, under certain circumstances. For example on the fandisk, as we see in 5.2, our algorithm finds the shallow features, while finding only one false positive in a bent area, while the simple algorithm finds no false positives either, but misses some parts of the shallow features.

5.4.4 Noise

In this example, we have a cube with some noise, created with the OpenFlipper Noise Plugin with a maximal distance of 0.005, which corresponds to about one third of the average edge length. It consists of 26'161 vertices, 78'477 edges and 52'318 faces. Both algorithms are having some trouble with the noise, while both algorithms still find most of the feature edges along the actual features and only find few false positives. However, the result of the simple algorithm includes a great number of single edges that are unconnected and are definitely no features. At the same time there are many gaps in the found features along actual features. Both issues do not occur in our algorithm, as such features are not allowed. This leads to longer paths of false positives and longer stretches along actual features of false negatives. But the total amount of false positives and false negatives is smaller. The result of our algorithm is also a lot better

suited to be manually improved, as only a few features have to be removed or added to achieve the optimal solution, while manually improving the result of the simple algorithm would be very time consuming.

If a mesh gets too noisy, however, our plugin cannot achieve good results anymore. After a certain degree of noisiness the result of the automatic detection is not much more than some arbitrary paths on the surface. However, the simple algorithm is not able to handle too much noise either and provides very bad results as well for too noisy meshes. The example we provided here has a level of noise such that the result of the simple algorithm is not useful at all while the result of our algorithm is not perfect, but still a lot better than the one of the simple algorithm.

5.5 Manual Refinement

In the noisy cube example used in 5.4.4, the result of the simple algorithm includes 550 false positives, most of them unconnected to any other detected edges and 64 false negatives, again most unconnected to other false negatives. In order to correct these errors manually, one would need to manually click on each of those edges. The result of our algorithm, on the other hand, includes 41 false positives, forming 4 paths and 17 false negatives, forming 3 paths. With the manual tools of our program, correcting these errors works the following way: splitting up features, such that the correct false positives can be deleted: 5 clicks, deleting the 4 false positives: 4 clicks, adding the 3 false negatives: 3 clicks with the add feature tool. Altogether, this takes 19 clicks, not including clicks needed to select and change the according tools and to adapt the view. If we compare this to the result of the simple algorithm and assume that the fastest way to change the state of an edge to or from a feature edge is to just select or unselect the according edge, then the time needed to obtain a perfect result is by far shorter with our program than without it. Without it, it would take about 614 clicks to just change the states of all the false positives and false negatives. The described method to select features manually by clicking on each single edge is indeed the fastest way to achieve this goal in OpenFlipper. So, regarding this, our plugin indeed provides an essential improvement.

We also made this observation in the three examples at the beginning of this chapter.

5.6 Time and Space complexity

Let us analyse the time and space complexity of the different parts of our algorithm:

5.6.1 Setting up Dual Graph

A polyhedron in general is a sparse graph, which means that $O(|V|) = O(|E|)$, due to Euler's formula, as well as its dual graph, so $O(|N|) = O(|L|)$. As each node in the dual graph represents an edge in the mesh, altogether this gives us $O(|V|) = O(|E|) = O(|N|) = O(|L|)$. The graph consists of a list of nodes and links, so the whole graph as a data structure as well as the time to set it up is in $O(|E|)$. After setting up the graph, the modifiers are calculated. These

calculations take place in a small neighbourhood of the according nodes and links, so it is also in $O(|E|)$. In conclusion, the whole process of setting up the dual graph is in $O(|E|)$.

5.6.2 Finding Seed Edges

To find the seed edges, we run Dijkstra's algorithm from each node until a specified maximal depth to obtain a rating for each node. See section 4.5 for a detailed description of this process. Let d be this maximal depth. Then, the amount of visited nodes is in $O(d^2)$, or, if d is sufficiently large, such that a considerable part of the total mesh is covered, in $O(|E|)$. As described in section 4.5, d is calculated as a user specified parameter multiplied with the average edge length of the mesh. The visited subgraph of the dual graph is also sparse, so the amount of visited links is also in $O(d^2)$. The Dijkstra algorithm has a time complexity of $O(|E| * log(|V|))$, so in this setting, the time complexity is in $O(d^2 * log(d^2))$. As we perform this process for each node, the total time complexity to calculate the rating of the nodes is in $O(|E| * d^2 * log(d^2))$. This process has to be done once per mesh, and has to be redone if any parameters have changed, not however, if we just want to find more features with the same parameters on the same mesh. Once the rating of the nodes has been performed, we only need to find the node that has the highest rating and is still available. This process can be done in $O(|E|)$. If we consider d as a constant and if $d << |E|$, then this whole process is in $O(|E|)$.

5.6.3 Extending Seed Edges to Cyclic Feature Paths

This part of the algorithm is greedy, meaning that we extend the path one edge by another in both directions until it forms a cycle. In the decision which edge we take for the extension, only the direct neighbours are considered and have an influence on our choice. In general, vertices in meshes have an average valence of about 6, so the amount of possible extensions can be considered a constant. The feature path has to be extended until it forms a cycle. This is guaranteed to happen after $O(|E|)$ extensions, as such a path cannot include more than all available edges. However, in practice, a cycle will be obtained much faster, especially, if other features have already be found, as the chance increases of meeting one of them. After each extension, we need to check, whether we have met another feature or if we have to continue. This is just a local computation and can be considered to run in constant time. So, finding a new feature path is also definitely in $O(|E|)$.

5.6.4 Automatic Part in General

The automatic part mainly consists of the three aforementioned parts, all of which run in linear time, as we have seen. So the complete algorithm should also run in linear time. All data that is calculated is stored directly in the dual graph data structure, apart from the found features, which are stored as lists of indices of nodes in the dual graph. These lists are all definitely in $O(|E|)$, as an edge can never be in more than one feature path and we certainly cannot have more feature paths than edges in the mesh. With the space complexity of the dual graph being in $O(|E|9$, as we have seen in section 5.6.1, we can conclude

that this algorithm has linear time and space complexity for reasonably small values of d ($d << |E|$).

5.6.5 Experimental Evidence of Linear Time Complexity

In the previous sections, we have conjectured, that our automatic algorithm runs in linear time. Following are two experiments to show that this also holds in practice.

A note on the experiment setup:

We executed this experiment on two meshes: (1) The fandisk, we already used previously, consisting of 25'994 vertices, 77'976 edges and 51'984 faces and a simple cube, as seen in 1.1, consisting of 8 vertices, 18 edges and 12 faces. Starting with these meshes, we repeatedly subdivided the meshes to get finer tessellations. In each subdividing step, each triangle gets split up to four smaller triangles. With each step, the amount of edges and the amount of faces are multiplied by four exactly and the amount of vertices is multiplied by approximately four (To be exact, the formula is $|V|' = 4 * |V| - 6$). For each of the resulting meshes, we let our algorithm run 10 times and measure the time it took for the three main stages:

- Setting up dual graph and calculating all modifiers, and the modified virtual lengths
- Rating the nodes, finding the first seed node and extending it to the first feature path
- Finding further feature paths.

The second part also includes finding the first feature path because, given the structure of our algorithm, it was a lot easier to measure the time for both steps rather than for each step independently. However, as we have seen, both steps should run in linear time, so the combination should also run in linear time.

For the third part we decided to add ten more features to the fandisk and only three more features to the cube, as after four features, all features on the cube are found. Anyway, this number does not really matter, as we are mainly interested in the asymptotical growth of computing time, not the actual time it takes for this complexity analysis.

We repeat this until a further refinement of the mesh is not possible anymore due to a memory shortage, which was the case after about 4'000'000 edges. The times in the resulting analysis are just the average of these 10 independent experiments.

Let us start with the fandisk:

Edges	77'976	311'904	1'247'616	4'990'464
Dual Graph Setup	392	1'719	7'150	30'030
Node Rating	75	365	1'525	6'384
Features 2 to 11	117	625	2'377	9'412
Total	585	2'710	11'054	45'826

Table 5.1: Time in ms for automatic detection on Fandisk in different sizes

To get a better understanding of the asymptotical growth, we also provide the dual table 5.2 where we can see the factor from one size to the next for all four metrics.

Edges	4.000	4.000	4.000
Dual Graph Setup	4.385	4.159	4.200
Node Rating	4.867	4.178	4.186
Features 2 to 11	5.341	3.803	3.959
Total	4.632	4.078	4.146

Table 5.2: Dual Table to 5.1

In this table we can clearly see the linearity of our algorithm. For an even better impression on the complexity, we also provide the graph of this table at the end of this section. Note, that this is a linear graph, while the mesh size grows exponentially, so the smaller samples are barely visible. However, the linearity is still apparent.

The second example is the cube. Here, we have a lot more data, because the base mesh is a lot smaller, so we were able to refine it a lot more often until the hardware limits were reached. However, the smaller samples are also more prone to errors in measurement and other constant factors can influence the runtime more significantly than in larger samples. The results for the cube with less than 4'608 edges were left out, because they are not very significant as time measurement is very inexact for small periods of time.

Edges	4'608	18'432	73'728	294'912	1'179'648	4'718'592
Dual Graph Setup	25.2	94.7	437.5	1'833.6	7'481.3	29'966.2
Node Rating	4.8	19.3	96.9	417.8	1'651.2	6'527.2
Features 2 to 4	2.8	10.6	60.7	249.9	986.1	3'916.1
Total	32.7	124.6	595.0	2'501.2	10'118.5	40'409.5

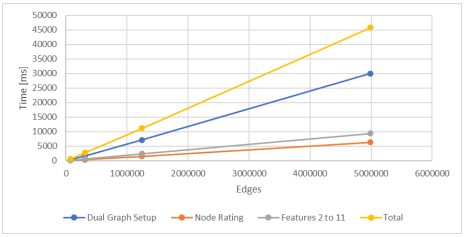
Table 5.3: Time in ms for automatic detection on a cube in different sizes

Again, let us look at the dual table, that shows, as in the previous example, the factors from one size to the next

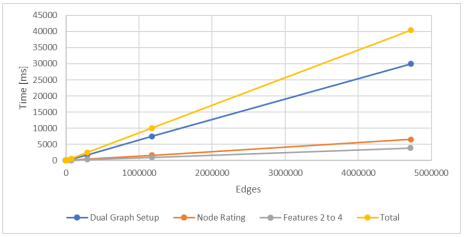
Edges	4.000	4.000	4.000	4.000	4.000
Dual Graph Setup	3.757	4.619	4.191	4.080	4.005
Node Rating	4.021	5.020	4.312	3.952	3.953
Features 2 to 4	3.786	5.726	4.117	3.946	3.971
Total	3.810	4.775	4.204	4.045	3.994

Table 5.4: Dual table to 5.3

We can see again that the algorithm clearly works in linear time. For an even better impression of this, let us have a look at the graph. As for the previous graph, keep in mind, that the graph is linear, while the size growth of the samples is exponential, so the smaller samples are not really visible in the graph. This is why we don't see the irregularities mentioned before in this graph.



(a) Graph of table 5.1



(b) Graph of table 5.3

These two examples show very clearly, that the algorithm indeed runs in linear time, supporting our conjecture in the previous sections.

Chapter 6

Continuing Work

The result of our work allows to efficiently detect the features on a triangle mesh and also provides the user with some simple but powerful manual tools to improve the result of the automatic detection. However, during the development, many ideas came up to improve the performance and usability of this program. The most important are listed here:

- Rating of feature curves

  Rate possible features in a way that satisfies the properties defined in chapter 3. If we had such a rating, we could easily add some very nice functionality to the program, some of which are described below. The difficulty lies in calculating such a rating that satisfies the desired properties.

- Non greedy feature extension

  Finding the best possible path through a given seed edge, rather than extending it greedily in both directions, would probably lead to better results, especially some false positives that cross flat regions would be less likely to occur. This would take significantly more time than the current approach, which is, regarding the current time complexity, no big issue. It would also require a good rating of features such as defined in section 6 to compare possible features. One of the reasons why we decided to implement a greedy approach was the computing time it took for a non greedy version in early development, so we decided quite early to follow this approach. However, during the development, we implemented several improvements regarding the computation time and with the resulting time complexity being rather fast, this option now has become feasible again.

- Superset

  This option is not planned to be actually implemented, it is just a different goal that could alternatively be set. While we tried to find an optimal result as fast as possible, the defined goal could also be to find a superset of the features, being a set, that guarantees to contain all features, but might also contain some false positives. This would make it easier in some way to manually refine the result to obtain an optimal result, as we could concentrate on removing false positives and would not have to bother to be able to add additional features. It would also enable us to implement

a second step in the automatic part which would only need to detect false positives and remove them. It is not clear, which of the two options would be easier or faster to implement or more useful.

- Automatic abort condition for adding new features

  In our program, the user must provide the program with a parameter called "Amount of Features", which is used to know when to abort finding new features. If we were able to rate features, we could abort adding new features as soon as there are no more features with a good enough rating. However, this would lead to another parameter, namely the rating threshold that may not be surpassed, but this threshold would probably be applicable to many different meshes and would not depend on geometry, size and tessellation, as the current parameter does.

- Extending the plugin to work on arbitrary polygonal meshes

  Right now, our program only works on triangle meshes. Extending it in a way that it also works on other polygonal meshes would greatly improve the usability as it would enable the plugin to be used on many more meshes. The resulting program does not rely on the fact that we are working exclusively on triangular meshes, so adding this functionality should not be too difficult.

- Considering more properties for the calculations of the modifiers

  In this paper, we considered the dihedral angle φ (sharpness), the angle between the two edges ϑ (smoothness), the flatness of the one-ring neighbourhood and the deviation of the dihedral angle φ for the calculations of the modifiers. Other properties might be used as well or the properties we considered might be extended, e.g. the region considered for the flatness of the neighbourhood so far is only the one-ring of the central vertex, the two-ring or a dynamical local neighbourhood could be used instead.

- Debugging graphic representation

  Unfortunately, there have been some bugs regarding the representation of the mesh in OpenFlipper, leading to strangely colored edges. Whether these bugs are caused by this plugin or some other OpenFlipper functionality is not yet known.

Glossary

- Mesh m

  A mesh is the fundamental object we are working on. It consists of vertices v_i, edges e_i and faces f_i in a way that each edge connects two vertices and each face is bordered by some edges and vertices. Our algorithm only works on triangle meshes, meaning that all faces are triangles.

- Vertex v

  A Vertex v in a mesh is defined by its three dimensional position in space.

- Edge e

  An edge e always connects exactly two vertices v_1 and v_2 and is defined by them. Two vertices are connected by at most one edge.

- Face f and normal

  A face f in a triangular mesh is defined by the three vertices it connects or by the three edges that form its borders. We can define a face normal vector as the perpendicular vector through the plane this face lies in.

- Node n

  A node n of our dual graph g represents an edge e of the original mesh m.

- Link l

  A link l of our dual graph g connects two nodes n_1 and n_2 iff the two edges e_1 and e_2 in m represented by these nodes are neighbours in the mesh m.

- Dual graph g

  The dual graph g consists of a list N of nodes with $n_i \in N$ for $i \in [1, |N|]$, where $|N|$ is the amount of nodes in N and $|N| = |E|$ for the list E of edges in the mesh M, and a list L of links with $l_i \in L$ for $i \in [1, |L|]$, where $|L|$ is the amount of links in L.

- Modifier of node n: $m(n)$

  Each node n is assigned a modifier based on dihedral angle of the edge it represents, i.e., the likelihood of this edge to be in a feature line. See section 4.3 for the complete calculation of this modifier.

- Modifier of link l: $m(l)$

  Each link l is assigned a modifier based on the sharpness modifier, smoothness modifier, neighbourhood modifier and . See section 4.4 for the details of the computation of this modifier.

- Modified virtual length $mvl(l)$

  The modified virtual length of a link is the result of the "Rating of Links" part of our algorithm, described in section 4.4. It depends on the modifiers, the modifier weights and the Euclidean lengths of the two adjacent edges e in the mesh m. For an exact definition of the calculation, please refer to equation 4.5.

- Euclidean length of edge e: $l(e)$

  The Euclidean length of an edge e: $l(e)$ connecting vertices v_1 and v_2 in mesh m is the Euclidean length of the vector from v_1 to v_2. Let p_1 and p_2 be the three dimensional points representing the position of v_1 and v_2 respectively, then $l(e) = |v_2 - v_1|_2$.

- Dihedral angle φ

  The dihedral angle φ is the angle that is formed by the two adjacent faces of an edge in the mesh. We always consider the smaller angle, so it is always in $[0, 180]^\circ$. As we do not consider degenerate meshes as valid inputs, we can even exclude the 180° case and actually get $\varphi \in [0, 180)^\circ$. ϕ is used to calculate the node modifier m_n.

  The definition of the dihedral angle we used in this paper defines $\varphi = 0^\circ$ for two faces that are coplanar but not coinciding and $\varphi = 180^\circ$ for two faces that are coinciding. This definition differs from the general definition of the dihedral angle, which is defined exactly the other way round. However, we found it more convenient to work with this definition and one definition can be easily converted into the other one by subtracting φ from 180°.

- Angle between two edges ϑ

  The angle between two adjacent edges ϑ is the smallest angle between two edges. Again, as we only consider non degenerate meshes, we get $\vartheta \in [0, 180)^\circ$.

- False positives

  When we compare the result of the automatic feature detection with the optimal feature network, we might find some edges that have been detected as features by the algorithm, but that are not actually features. Such edges are called false positives.

- False negatives

  When we compare the result of the automatic feature detection with the optimal feature network, we might find some edges that are actually features but that have not been detected by the algorithm. These edges are called false negatives.

Chapter 7

Conclusion

The goal of this thesis was to identify feature lines along mesh edges in mathematical terms and to implement a plugin for the OpenFlipper environment to enable a user to quickly find all features on this mesh in as little time as possible. This plugin should be an addition to the OpenFlipper environment that actually improves the workflow of an average user trying to identify the features to use them for further modifications or calculations on the mesh. Before this thesis, OpenFlipper provided no such functionality, so our resulting program certainly is a useful addition for its functionality, even though it is far from perfect.

We defined feature lines in mathematical terms in chapter 3, where we focused on a function that evaluates a possible feature path. We described the desired behaviour of such a function, which helped us to identify the relevant properties of features. However, we were not able to actually construct such a function. The lack of such a function prevented us from implementing some desired features in the final algorithm, such as automatically finding all features without any user input. The development of such a function would enable us to improve the algorithm in several different ways, as described in chapter 6.

However, even though we had no such function, we were still able to implement a plugin that generally provides relatively good results. Our algorithm consists mainly of two steps, (1) finding seed edges and (2) extending these seed edges to feature paths. The first part is done by a rating algorithm that takes a relatively small neighbourhood of the edge into account. This part works relatively well, given well chosen parameters for the modifier weights. However, choosing the best modifier weights is up to the user and is not always an easy task. If the mesh contains a lot of difficulties, such as noise, shallow features, short features, bent surfaces or irregular tesselation, then it might be very difficult to find parameters that work well for the whole mesh. In many cases, however, it is possible to achieve relatively good results, often even without changing the standard parameters.

The manual tools we also provide in the plugin make manually postprocessing the result quite simple and fast and the combination of the automatic and manual part allow a user to identify all features in a short time with a rather small amount of work.

So, while there are still some aspects to be improved, mainly in the automatic detection part, the plugin altogether certainly provides the average user with a fast and powerful tool to quickly identify the features.

List of Figures

Bibliography

[D+59] Dijkstra, Edsger W. u. a.: A note on two problems in connexion with graphs. In: *Numerische mathematik* 1 (1959), Nr. 1, S. 269–271

[KCL09] Kim, Hyun S. ; Choi, Han K. ; Lee, Kwan H.: Feature detection of triangular meshes based on tensor voting theory. In: *Computer-Aided Design* 41 (2009), Nr. 1, 47 - 58. `http://dx.doi.org/https://doi.org/10.1016/j.cad.2008.12.003`. – DOI https://doi.org/10.1016/j.cad.2008.12.003. – ISSN 0010–4485

[LL02] Lee, Y. ; Lee, S.: Geometric Snakes for Triangular Meshes. In: *Computer Graphics Forum* 21 (2002), Nr. 3, 229-238. `http://dx.doi.org/10.1111/1467-8659.t01-1-00582`. – DOI 10.1111/1467–8659.t01–1–00582

[MK12] Möbius, Jan ; Kobbelt, Leif: OpenFlipper: An Open Source Geometry Processing and Rendering Framework. In: Boissonnat, Jean-Daniel (Hrsg.) ; Chenin, Patrick (Hrsg.) ; Cohen, Albert (Hrsg.) ; Gout, Christian (Hrsg.) ; Lyche, Tom (Hrsg.) ; Mazure, Marie-Laurence (Hrsg.) ; Schumaker, Larry (Hrsg.): *Curves and Surfaces*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–27413–8, S. 488–500

[MTAM+19] Moscoso Thompson, E ; Arvanitis, G ; Moustakas, K ; Hoang-Xuan, N ; Nguyen, E R. ; Tran, M ; Lejemble, Thibault ; Barthe, Loïc ; Mellado, Nicolas ; Romanengo, C ; Biasotti, S ; Falcidieno, B: SHREC'19 track: Feature Curve Extraction on Triangle Meshes. In: *12th EG Workshop 3D Object Retrieval 2019*. Gênes, Italy, Mai 2019, 1 - 8

[PC20] Pouget, Marc ; Cazals, Frédéric: Approximation of Ridges and Umbilics on Triangulated Surface Meshes. Version: 5.0.2, 2020. `https://doc.cgal.org/5.0.2/Manual/packages.html#PkgRidges3`. In: *CGAL User and Reference Manual*. 5.0.2. CGAL Editorial Board, 2020