UNIVERSITÀ DEGLI STUDI DI CAGLIARI

FACOLTÀ DI SCIENZE

Corso di Laurea Magistrale in Informatica

Simplifying the Shape of triangle meshes for Unfolding, Milling and Fabrication

An algorithm for the semplification of three-dimensional shapes that represent real objects, in order to facilitate the construction of wooden or paper models as educational tools or toys.

Supervisor
Prof. Riccardo Scateni

Student
Alessandro Muntoni
Matr. N. 48234

Anno Accademico 2013/2014

Contents

1 Introduction **1**

2 Background **3**
2.1 Variational Shape Approximation 3
2.2 Fabrication 3

3 Algorithm Overview **5**

4 Making Topology **7**
4.1 Goals 7
4.1.1 Orientations 7
4.1.2 Angles between Adjacent Faces 8
4.2 Trivial Labeling 11
4.3 Graph-Cut Labeling 12
4.3.1 Graph-Cut Algorithm 12
4.3.2 Graph-Cut in our Algorithm 12
4.4 Managing Segmentation 14

5 Making Geometry **19**
5.1 Goals 19
5.2 Deformation 19
5.2.1 Compute 19
5.2.2 Integrate 21
5.2.3 Iterated Deformation 23
5.2.4 Coarse Model Extraction 24
5.3 Edge Collapsing 24
5.3.1 Case 1: No Incident Triangles 25
5.3.2 Case 2: One Incident Triangle 25
5.3.3 Case 3: Two Incident Triangles 26
5.3.4 Results 30
5.4 Planarization 30

6 Results **35**

7 Conclusions and Future Works **42**

Chapter 1

Introduction

One of the main research fields that has gathered increasing attention from Computer Graphics researchers is Fabrication. Fabrication goal is to obtain a physical object, starting from a digital model stored in a computer. 3D printers are the most known machines that allow to fabricate a virtual shape (also quiet complex) using different materials and techniques, but other techniques are also available.

In this thesis, we consider the problem of fabricating a virtual shape by cutting rigid materials (e.g. wood or thick paper) that are glued together along their boundaries. The main challenge of this approach is twofold: on one side it is necessary to obtain a physical model that visually resemble to the original model; on the other side, we need a model that is easy to build using the above-mentioned fabrication technique: this model must have a low total number of pieces, and the complexity of each piece must be low, so as to make the assembly process easier and to reduce fabrication costs.

We propose an algorithm that, given a 3D model, simplifies the input model in such a way that is buildable with laser cut techniques on plywood. We decided to focus on the minimization of costs of fabrication and simplification of the construction process. To minimize fabrication costs, that depends especially on the cutting times (e.g. for energy, usury), we aim to delete, where is possible, frequent changes of cutting direction during the cut process. To allow an easy and guided construction process, we decided to facilitate the folding planes process by reducing the set of possible angles between two adjacent faces to a restricted set of well-known angles, that are perfectly suitable for the cut process. In this way, during the fabrication, see that two adjacent pieces fit perfectly is more simple, as shown in Figure 1.1. To do that, the input model is simplified in such a way that the orientation of every single face is well-defined and belongs to a restricted set of allowable orientations.

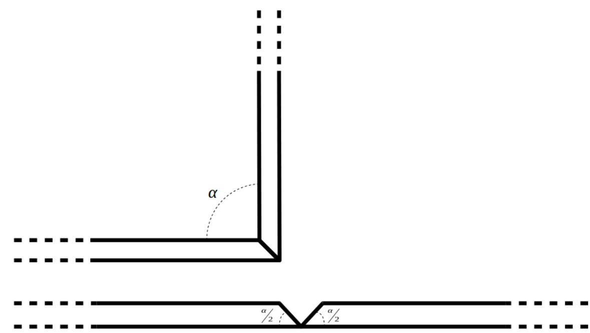


Figure 1.1: Example of an angle between two adjacent pieces during a cut process

The rest of this thesis is organized as follows: Chapter 2 reviews previous works on fabrication-oriented modeling; Chapter 3 shows an overview of our algorithm, Chapter 4 describes how the algorithm calculates the resulting topology of the model, and Chapter 5 shows how, using that topology,

is possible to obtain the resulting geometry of the model. In Chapter 6 we show some results, and finally, in Chapter 7 we conclude with a short summary, achievements and future developments.

Chapter 2

Background

The actual state of the art presents different 3D model simplification algorithms oriented at model construction by cutting and assemblage.

2.1 Variational Shape Approximation

Cohen-Steiner et al. [CSAD04] presented an algorithm for shape approximation, maintaining its geometric fidelity. Julius et al. [JKS05] proposed D-Charts, an algorithm for mesh segmentation that produces developable charts for making stuffed toys. These two approaches produce a simplification of the model where the charts flatness is not guaranteed, and therefore are not suitable for fabrication using rigid materials like plywood.

2.2 Fabrication

Elber [Elb95] proposed a fabrication scheme that automatically approximates a model creating developable surfaces that can be unrolled onto a plane.

Mitani and Suzuki [MS04] proposed an algorithm for papercraft construction, starting from a 3D model (Figure 2.1). Papercraft is a construction method for three-dimensional objects using paper, glue and scissors: paper strips are cutted, then are glued side by side, forming a representation of the real object. This algorithm, starting from a 3D model, creates paper strips that can subsequently printed, cutted and glued. This kind of papercraft are suitable for experts and fans, but are not recommended for people with little experience with papercrafts: bunny model construction toke 2 $^1/_4$ hours, and the rhinoceros model toke 3 $^1/_2$ hours. Furthermore, these models are not suitable for construction using other materials like wood, which is perfect for construction of toys with educational or simply entertainment purposes. Indeed, a wood-cut process on these model faces would be extremely slow (and therefore expensive), and error-prone because the angle between every pair of faces is arbitrary.

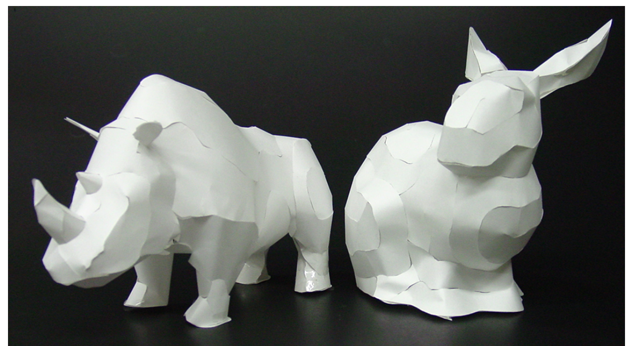
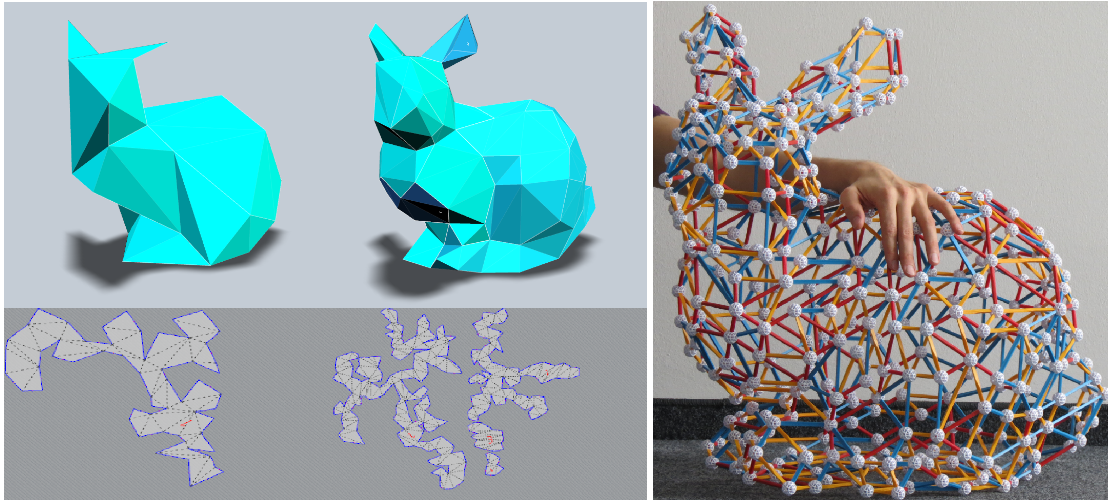


Figure 2.1: Mitani et al papercraft models



(a) 123DMake models with 30 and 130 vertexes (b) Zimmer et al. Zometool® model

Figure 2.2

Shatz et al. [STL06] presented an algorithm for segmenting a mesh into developable approximations for paper crafting, reducing the error between the given model and the paper model.

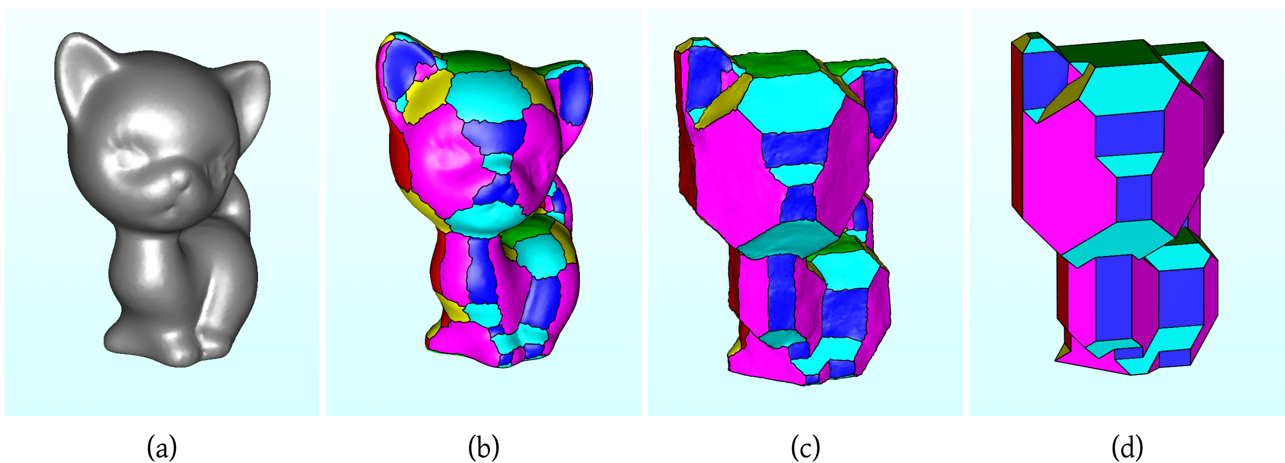
Autodesk's 123D Make [Aut14] is a software that allows 3D models construction using multiples techniques, including "folded panels" technique that gives an output similar to Mitani et al. and Shatz et al. ones, with the possibility of choose number of vertexes and the "joint" technique that user wants to use for joining the paper strips. Nevertheless, by selecting a big number of vertexes, we obtain a model that is very difficult to buid; instead, if we select a small number of vertexes, we obtain a model that has a simpler building process (not much), and that model is too simple to look like the original, as shown in Figure 2.2a.

Zimmer et al. [ZLAK14], starting from a 3D model stored in a computer, presented an algorithm that gives in output a simplified model buildable with Zometool®(Figure 2.2b), that is a building composed of nodes and edges that can be linked to each other. Every node allows 62 different entries for edges, tying the possible angles between adjacent faces to a small finite set. In any case, the model is suitable only for Zometool® constructions, that is complex and, at the moment, an algorithm for automatic generation of a guide for the build process is not known.

Chapter 3

Algorithm Overview

Our algorithm is divided in two principal steps: the goal on the first step is to create a good segmentation of the input model, where every chart corresponds to a planar polygon in the output model. In this step we first create a segmentation using an optimization algorithm, then we process the segmentation to remove noise and charts that aren't physically developable. In the second step we create the geometry of the model, first deforming the mesh in such a way that every chart is planar and well-oriented, then we take the corners of the deformed model (vertexes with three incident charts at least) and create the coarse model. We process the coarse model using edge collapsing techniques, and we do a final deformation (that we called planarization) to obtain the final model, ready for fabrication. Figure 3.1 shows the principal steps of our algorithm.



(a) (b) (c) (d)

Figure 3.1: The kitten model, model topology, model deformation and model planarization

Chapter 4

Making Topology

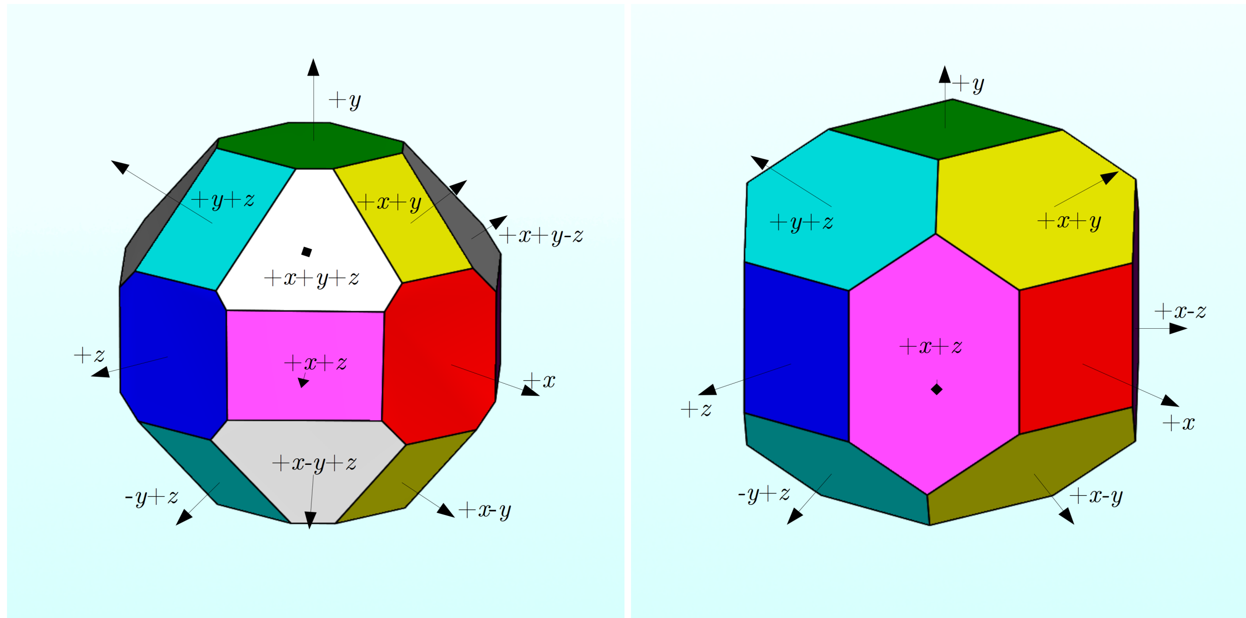
4.1 Goals

Given an input model, the first step is to obtain a good segmentation that gives a physically developable simplification that is simply to fabricate and construct through wood-cut process. In the next subsection, we analyse properties that must be met to reach our purpose.

4.1.1 Orientations

We initially decided to limit the possible orientations of charts to 26, where every orientation is the normal to the face, that is a 3D vector that belongs at the 3D vectors set composed by the 3-permutations with repetition of 3 values -1, 0 and $+1$, excluding the $\begin{bmatrix}0 & 0 & 0\end{bmatrix}$ null vector. As in Marching Cubes ([LC87], [MSS94]), we can imagine to obtain all orientations considering:

- the 6 principal orientations that are aligned to the 6 cube's faces, that are:
  - $+x$ ($\begin{bmatrix}+1 & 0 & 0\end{bmatrix}$);
  - $+y$ ($\begin{bmatrix}0 & +1 & 0\end{bmatrix}$);
  - $+z$ ($\begin{bmatrix}0 & 0 & +1\end{bmatrix}$);
  - $-x$ ($\begin{bmatrix}-1 & 0 & 0\end{bmatrix}$);
  - $-y$ ($\begin{bmatrix}0 & -1 & 0\end{bmatrix}$);
  - $-z$ ($\begin{bmatrix}0 & 0 & -1\end{bmatrix}$);
- the 12 orientations obtained by cutting everyone of the 12 cube's edges, that are:
  - $+x+y$ ($\begin{bmatrix}+1 & +1 & 0\end{bmatrix}$);
  - $-x+y$ ($\begin{bmatrix}-1 & +1 & 0\end{bmatrix}$);
  - $-x-y$ ($\begin{bmatrix}-1 & -1 & 0\end{bmatrix}$);
  - $+x-y$ ($\begin{bmatrix}+1 & -1 & 0\end{bmatrix}$);
  - $+x+z$ ($\begin{bmatrix}+1 & 0 & +1\end{bmatrix}$);
  - $-x+z$ ($\begin{bmatrix}-1 & 0 & +1\end{bmatrix}$);
  - $-x-z$ ($\begin{bmatrix}-1 & 0 & -1\end{bmatrix}$);
  - $+x-z$ ($\begin{bmatrix}+1 & 0 & -1\end{bmatrix}$);



(a) Deformed sphere using 26 orientations

(b) Deformed sphere using 18 orientations

Figure 4.1

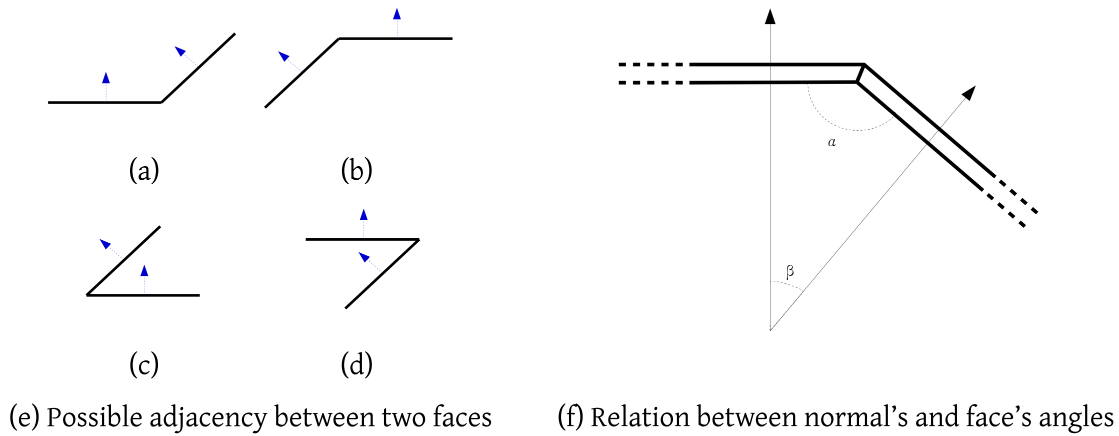
  - $+y+z$ ($\begin{bmatrix}0 & +1 & +1\end{bmatrix}$);
  - $-y+z$ ($\begin{bmatrix}0 & -1 & +1\end{bmatrix}$);
  - $-y-z$ ($\begin{bmatrix}0 & -1 & -1\end{bmatrix}$);
  - $+y-z$ ($\begin{bmatrix}0 & +1 & -1\end{bmatrix}$);

- the 8 orientations obtained by cutting everyone of the 8 cube's vertexes, that are:
  - $+x+y+z$ ($\begin{bmatrix}+1 & +1 & +1\end{bmatrix}$);
  - $-x-y-z$ ($\begin{bmatrix}-1 & -1 & -1\end{bmatrix}$);
  - $+x-y-z$ ($\begin{bmatrix}+1 & -1 & -1\end{bmatrix}$);
  - $-x+y+z$ ($\begin{bmatrix}-1 & +1 & +1\end{bmatrix}$);
  - $+x+y-z$ ($\begin{bmatrix}+1 & +1 & -1\end{bmatrix}$);
  - $-x-y+z$ ($\begin{bmatrix}-1 & -1 & +1\end{bmatrix}$);
  - $+x-y+z$ ($\begin{bmatrix}+1 & -1 & +1\end{bmatrix}$);
  - $-x+y-z$ ($\begin{bmatrix}-1 & +1 & -1\end{bmatrix}$).

However, as shown in Subsection 4.1.2, these orientations do not allow to obtain a easy buildable model. Therefore, we decided to use only the first 18 orientations, as shown in Figure 4.1b.

4.1.2 Angles between Adjacent Faces

Given two adjacent faces with different normal (we assume that two adjacent faces with the same normal are the same face), they can be adjacent to each other in four different ways, as shown in Figure 4.2e. We consider only the cases (a) and (b), because we assume that normals are facing outwards the model. The only difference between the cases (a) and (b) is that in case (a) we have a



(e) Possible adjacency between two faces (f) Relation between normal's and face's angles

Figure 4.2

concave internal angle, and in case (b) we have a convex internal angle. Given two normals $\overrightarrow{u}$ and $\overrightarrow{v}$ of two adjacent faces, the internal angle is calculated (as shown in the Figure 4.2f):

- $\alpha = \pi - \beta$ if the internal angle is convex;
- $\alpha = \pi + \beta$ if the internal angle is concave;

where β is the angle between the two normals $\overrightarrow{u}$ and $\overrightarrow{v}$:

$$\beta = \cos^{-1}(\overrightarrow{u} \cdot \overrightarrow{v})$$

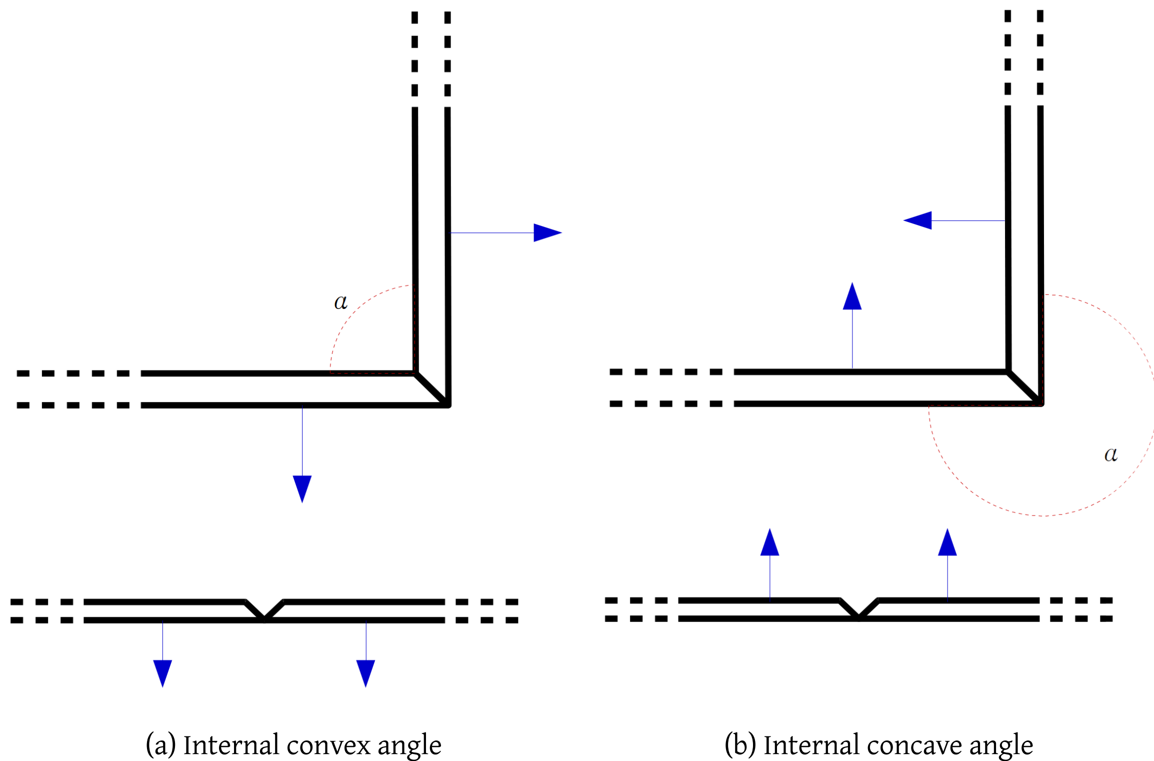
However, in terms of fabrication, process for cutting an internal concave angle is exactly the same of cutting an internal convex angle, by simply reversing the piece depending on the angle convexity, as shown in Figure 4.3. Therefore, given two normals $\overrightarrow{u}$ and $\overrightarrow{v}$, we can focus only on angles calculated in the following way:

$$\alpha = \pi - \beta$$
$$\beta = \cos^{-1}(\overrightarrow{u} \cdot \overrightarrow{v})$$

In a fabrication process, the cut angle in the edge will be exactly half of the convex angle between the two adjacent faces: therefore, if the internal angle is α, then the cut angle in the two wood pieces will be $\frac{\alpha}{2}$.

Now we can calculate all the possible convex angles between all possible combinations of adjacent faces. Will not be taken into account adjacent faces with the same orientation, because we can merge these faces, and adjacent faces with opposite orientations, because they are not physically developable. If we use all 26 possible orientations listed in Subsection 4.1.1:

- A face with generic normal $\pm u$ (vector with only one component different by 0):
  - adjacent with a face having normal with only one component different by 0:
    - generates always a $90°$ angle;
  - adjacent with a face having normal with two components different by 0 generates a:
    - $135°$ angle if the u component is different by 0 and has the same sign;
    - $45°$ angle if the u component is different by 0 and has opposite sign;



(a) Internal convex angle

(b) Internal concave angle

Figure 4.3: Possible adjacency between two faces

  - ♢ $90°$ angle if the u component is 0;
- ∘ adjacent with a face having normal with three components different by 0 generates a:
  - ♢ $\approx 126°$ angle if the u component has the same sign;
  - ♢ $\approx 54°$ angle if the u component has opposite sign;

- A face with generic normal $\pm u \pm v$ (vector with two components different by 0):
  - ∘ adjacent with a face having normal with two components different by 0 generates a:
    - ♢ $120°$ angle if has a component (between u and v) different by 0 and with same sign, and the other equal to 0;
    - ♢ $60°$ angle if has a component (between u and v) different by 0 and with opposite sign, and the other equal to 0;
    - ♢ $90°$ angle if both the u and v components are different by 0 and only one has the same sign;
  - ∘ adjacent with a face having normal with three components different by 0 generates a:
    - ♢ $\approx 145°$ angle if has both u and v components with same sign;
    - ♢ $\approx 35°$ angle if has both u and v components with opposite sign;
    - ♢ $90°$ angle if one component (between u and v) has the same sign and the other has opposite sign;

- A face with generic normal $\pm u \pm v \pm w$ (vector with three components different by 0):
  - ∘ adjacent with a face having normal with three components different by 0 generates a:

- ♢ $\approx 70°$ angle if the number of components with the same sign is even;
- ♢ $\approx 110°$ angle if the number of components with the same sign is odd;

We can count only 11 possible angles between every pair of faces having one of the 26 possible orientations defined above. However, using these orientations, we can't have a well-defined cut process: there are angles that are not well-defined, represented by numbers with unlimited decimals. A cut machine works fine with angles such as $30°$, $45°$, $60°$, $90°$, $120°$, $135°$ and $150°$, and we can limit our angles set to these values by simply not including the last 8 orientations (vectors with three components different by zero), that are the only that generates bad angles. In this way, we have a set of 18 possible orientations and only 7 possible angles: $45°$, $60°$, $90°$, $120°$, $135°$. A model that has only this kind of angles is perfect for a fabrication process by cutting and folding wood panels.

From this point onwards, for a better understanding of the segmentation, we associate different colors to the labels. The colors are calculated as following: using the hexadecimal color notation, and associating two ciphers for every vector component, if the component is 0, then the two ciphers will be 00, else the two ciphers will be FF. In this way, we have:

- Red color if label is $\pm x$ (#FF0000);
- Green color if label is $\pm y$ (#00FF00);
- Blue color if label is $\pm z$ (#0000FF);
- Yellow color if label is $\pm x \pm y$ (#FFFF00);
- Magenta color if label is $\pm x \pm z$ (#FF00FF);
- Cyan color if label is $\pm y \pm z$ (#00FFFF);

4.2 Trivial Labeling

Now that we have established what are the possible orientations that a face can have, we can compute a segmentation on a 3D triangle mesh. A first solution can be a "trivial labeling", that associates to each triangle one of the 18 possible orientations, assigning a label.

The trivial labeling is quite simple: for every triangle, we calculate the angular distance, using the dot product, between its normal and all the 18 possible directions. We are searching the orientation with minimum angular distance to the normal of the triangle, therefore we select the orientation with the maximum result of the dot product.

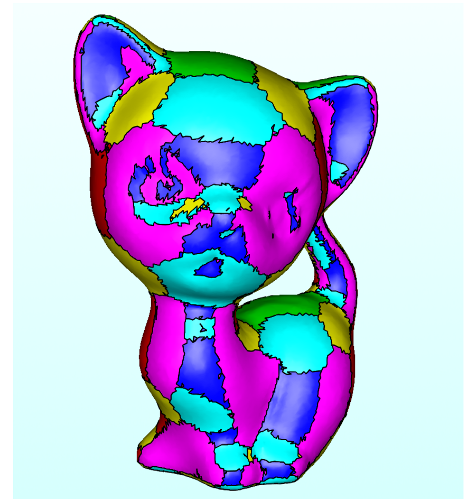


Figure 4.4: Trivial labeling

In Figure 4.4 we can see a segmentation of the bunny model using this method. This is not the result that we want: the charts (that are sets of adjacent triangles with the same label) have too much noise (e.g. single triangles with different labels at the center of a chart, triangle strips or chart not physically developable). We desire a segmentation composed of compact charts without holes (if they aren't a feature of the input model) and without noise. We can obtain a segmentation more closer to our ideal segmentation computing a Graph-Cut based labeling, shown in the Section 4.3.

4.3 Graph-Cut Labeling

To obtain a good segmentation with compact charts, like in [LVS*13], we decided to use an optimization algorithm that minimizes a cost described by the labels associated with triangles.

4.3.1 Graph-Cut Algorithm

Graph-Cut ([BVZ01],[KZ04],[BK04]) is a Max-Flow/Min-Cut algorithm on graphs, that, given a graph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ and a finite set of labels L, finds a labeling ℓ (a function that associates, for every site $p \in \mathcal{N}$ a label $l \in L$) that minimizes the following Energy:

$$E(\ell) = DataCosts(\ell) + SmoothCosts(\ell) + LabelCosts(\ell)$$

More specifically:

$$DataCosts(\ell) = \sum_p D_p(l_p)$$

where l_p is a potential label for node $p \in \mathcal{N}$, and D_p is the cost of assigning l_p;

$$SmoothCosts(\ell) = \sum_{pq} V_{pq}(l_p, l_q)$$

where $p, q \in \mathcal{N}$, $(p, q) \in \mathcal{A}$, and V_{pq} is the cost for assigning l_p, l_q to p and q;

$$LabelCosts(\ell) = \sum_{L'} h_{L'}(\ell)$$

where L' is some subset of labels L, and $h_{L'}$ adds a cost if and only if at least one label $l \in L'$ appears in the labeling ℓ.

The first term in the energy function $E(\ell)$ is typically called the data term, and it consists of the sum over all nodes p of the penalty (or cost) $D(p, l_p)$, what should be the cost of assigning label l_p to node p. The second term is a sum over all pairs of neighboring nodes $(p, q) \in \mathcal{A}$. That is there is a neighborhood relation on the set of pixels (this relationship is symmetric, that is if p is a neighbor of q then q is a neighbor of p). This second term is typically called the smoothness term. The third term is a sum over all labels (or, more generally, all subsets of labels) such that each label can have non-negative penalties associated with its use. This "label cost" feature is used to encourage labeling that use as fewer unique labels or, more generally, labels from as few unique subsets as possible.

4.3.2 Graph-Cut in our Algorithm

In our implementation, the graph is composed of a node for every triangle in the mesh, and two nodes are neighbors if and only if the associate triangles are adjacent. Therefore, with Graph-Cut we can do local operations like avoiding chart adjacent with another chart with opposite normal, by simply assign a greater cost for adjacent nodes with opposite labels. However, we can't do global operations, like make a chart has at least three neighbors. These kind of operations will have to be made after the execution of Graph-Cut algorithm.

Data Term

The Data Term function, for a given node and a given label, returns a cost for associating the label to the node (triangle). Intuitively, more the normal of the triangle is close to the orientation associated with the label, then lower will be the cost of associating the label to the triangle. Also in this case, the closeness is defined by the dot product of the two vectors. Given the triangle normal $\overrightarrow{t}$ and the orientation $\overrightarrow{l}$ of a label l, the cost function is:

$$DataTerm(\overrightarrow{t}, \overrightarrow{l}) = 1 - e^{-\frac{1}{2}\left(\frac{\left(\overrightarrow{t} \cdot \overrightarrow{l}\right) - 1}{\sigma}\right)^2}$$

The σ term determine the amplitude of the bell: in our implementation $\sigma = 0.1$. The graph of this function is shown in Figure 4.5. We used this function to obtain a gradual increase from the lower cost to the upper cost, that, in this case, are respectively ≈ 0 and ≈ 1. If the dot product between the two vectors is 1, the cost will be ≈ 0 because the angle between $\overrightarrow{t}$ and $\overrightarrow{l}$ is $0°$ and the triangle is already perfectly oriented to $\overrightarrow{l}$. We have a maximum cost from -1 to ≈ 0.7, i.e. a angular distance from $\approx 45°$ to $180°$, because if $\overrightarrow{t}$ and $\overrightarrow{l}$ have at least an angular distance of $45°$ it means that certainly there is another label l' that has an angular distance with $\overrightarrow{t}$ less than $45°$.

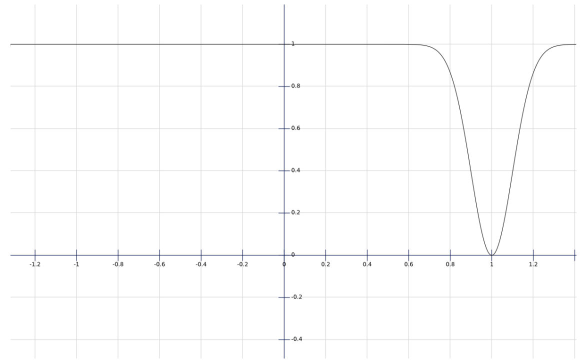


Figure 4.5: Data Term

Smooth Term

The Smooth Term function, for a given pair of neighbors nodes p and q, and a pair of labels l_p and l_q (not necessarily $l_p \neq l_q$) returns a cost for associating l_p to p and l_q to q. Intuitively, we must avoid adjacency between faces with opposite orientations, because they aren't physically developable: we can avoid this simply verifying if l_p and l_q are opposite and, in that case, returning a cost greater than another normal cost calculated using non-opposite labels. Moreover, if the two triangles are associated at the same label, the cost will be 0. In all other cases, we have to use an appropriate function.

In this function we will use a user parameter, the compactness factor c. Increasing this parameter, will increases the compactness of the carts, reducing the number of small charts and small features. However, a too big value can return a bad segmentation, that depends by the input model resolution. An example of segmentation using a too big compactness factor is shown in Figure 4.6.

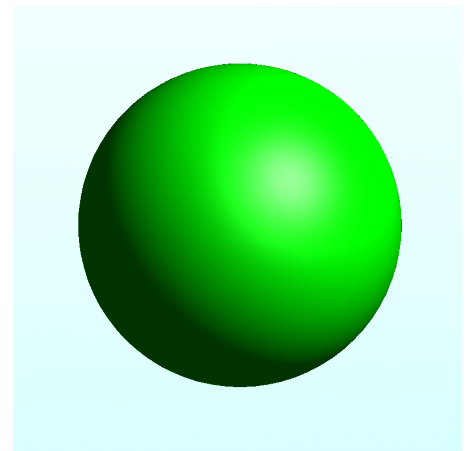
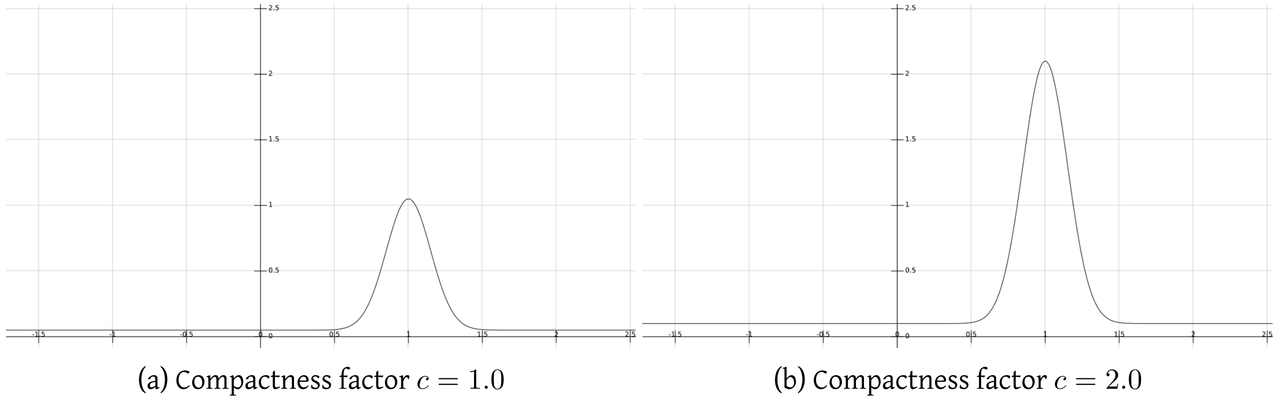


Figure 4.6: Sphere model with compactness factor $c = 20$

Intuitively, we want a segmentation that divide areas with different orientations: if two adjacent triangles have different



(a) Compactness factor $c = 1.0$ (b) Compactness factor $c = 2.0$

Figure 4.7: Different compactness factors with $\epsilon = 0.05$

labels but same orientations, they pay the maximum cost. Otherwise, if the angle between the two triangle normals is at least $45°$, they pay the minimum cost. Therefore, the function is:

$$SmoothTerm(\overrightarrow{l_p}, \overrightarrow{l_q}, \overrightarrow{t_p}, \overrightarrow{t_q}) = c \cdot \left(e^{-\frac{1}{2}\left(\frac{\overrightarrow{t_p} \cdot \overrightarrow{t_q} - 1}{\sigma}\right)^2} + \epsilon \right)$$

Also in this case, the σ term indicates the bell amplitude. Here, $\sigma = 0.15$: as shown in Figure 4.7, in this way we have a considerable increase on about $45°$.

In the smooth term we need also a term $\epsilon \neq 0$ because the compactness factor c must influence also when we have a 0 cost. However, we must be careful because the ϵ term goes to be multiplied with the compactness factor, increasing its weight in the resulting function. Therefore, we need a small value, but different by zero. In our implementation, $\epsilon = 0.05$. Figure 4.7 shows the function plot with different compactness factors.

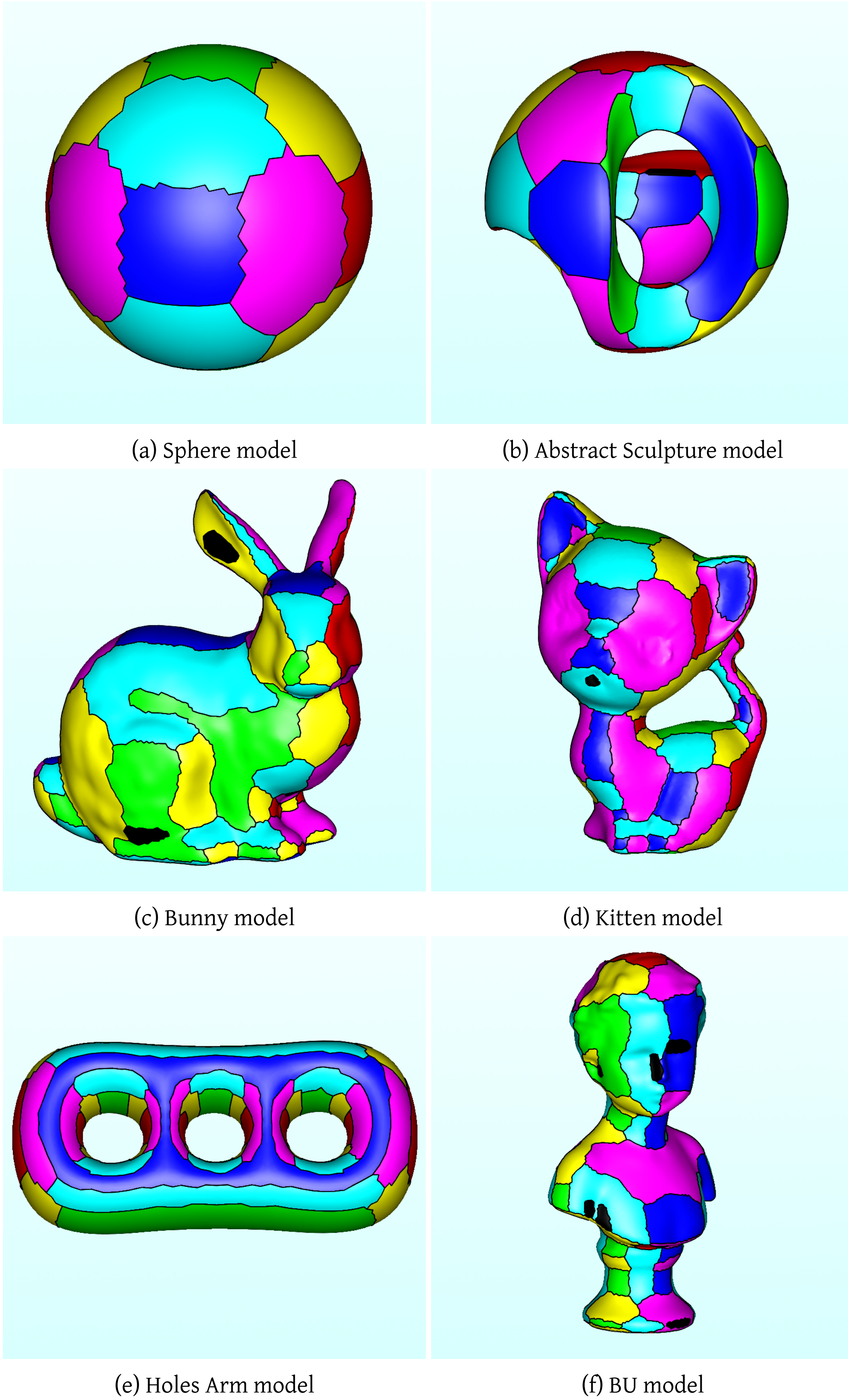
Results

In the examples shown in Figure 4.8 we used a compactness factor $c = 2.0$. The black faces are particular faces that have two or less corner vertexes, and then are not physically developable. In the Section 4.4 we show how identify these faces and how we generally manage a segmentation.

4.4 Managing Segmentation

The resulting segmentation from the graph-cut execution is not yet a good segmentation. Graph-cut can manage only local constraints, such as two adjacent triangles that can't have two particular labels. However, a good segmentation must have some global features, defined at "chart level". In our context, a chart is a group of adjacent triangles having the same label. A chart is characterized of a set of triangles with the same label, a set of boundary edges (edges incident on a triangle of the chart and on a triangle that not belongs to the chart), a set of chart neighbors and a set of corners, that are vertexes with at least three incident charts.

Data Structures and Algorithms



(a) Sphere model
(b) Abstract Sculpture model
(c) Bunny model
(d) Kitten model
(e) Holes Arm model
(f) BU model

Figure 4.8: Graph-Cut Labelings

To managing a segmentation, we need a data structure and a set of algorithms suitable for our purposes. Our segmentation is a graph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, where every chart is identified by a node $p \in \mathcal{N}$, and when two charts p and q are adjacent there is an edge $(p, q) \in \mathcal{A}$. A node $p \in \mathcal{N}$ is composed of a list of t triangles, a list of n external boundary edges, k lists of internal boundary edges and a list of m corners.

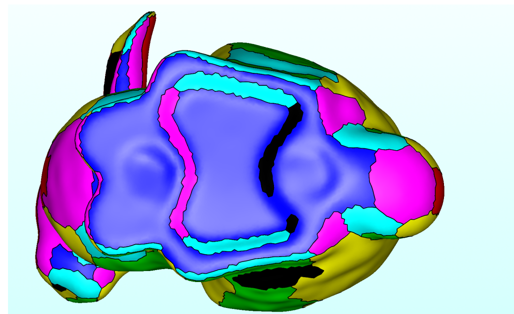


Figure 4.9: Possible segmentation on the base of the Bunny model

Finding the t triangles of a chart is quite simple. Starting from a triangle of the chart, we can do a simple flooding to the adjacent triangles with the same label, and marking as visited the triangles already added to the triangle list.

For the boundary edges, the problem is more complex. It may happen that a chart has one or plus holes, like in the Rocker Arm model (Figure 4.8f) or in the base of the Bunny model (Figure 4.9). To obtain list of external boundary edges and the lists of all internal boundary edges (a boundary for every hole), we can start from a unordered list of boundary edges seen during the flooding of the triangles. Starting from an edge on that list, and following the verse of the edge in the incident triangle of the chart, we can simply navigate the mesh following the edges that are incident in a triangle of the chart and in a triangle that don't belongs to the chart, until we don't reach the starting edge. If the boundary was external, we walked counter-clockwise, otherwise we walked clockwise, as shown in Figure 4.10. We build boundary edges until we haven't visited all the edges that are in the unordered list. Now we have to find which boundary is the external boundary. To do that, we calculate a point that is an average of the coordinates of all vertexes in the chart. After that, we look for the vertex on the boundaries farthest from the center. This vertex belongs to the external boundary, and all other boundary are then internal boundaries.

Now is simple to find the list of the corners for a chart: we can simply walk every boundary and look for the number of charts incident in every boundary vertexes: if that number is greater than or equal to three, the vertex is a corner.

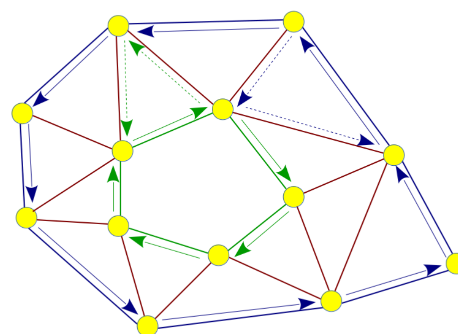
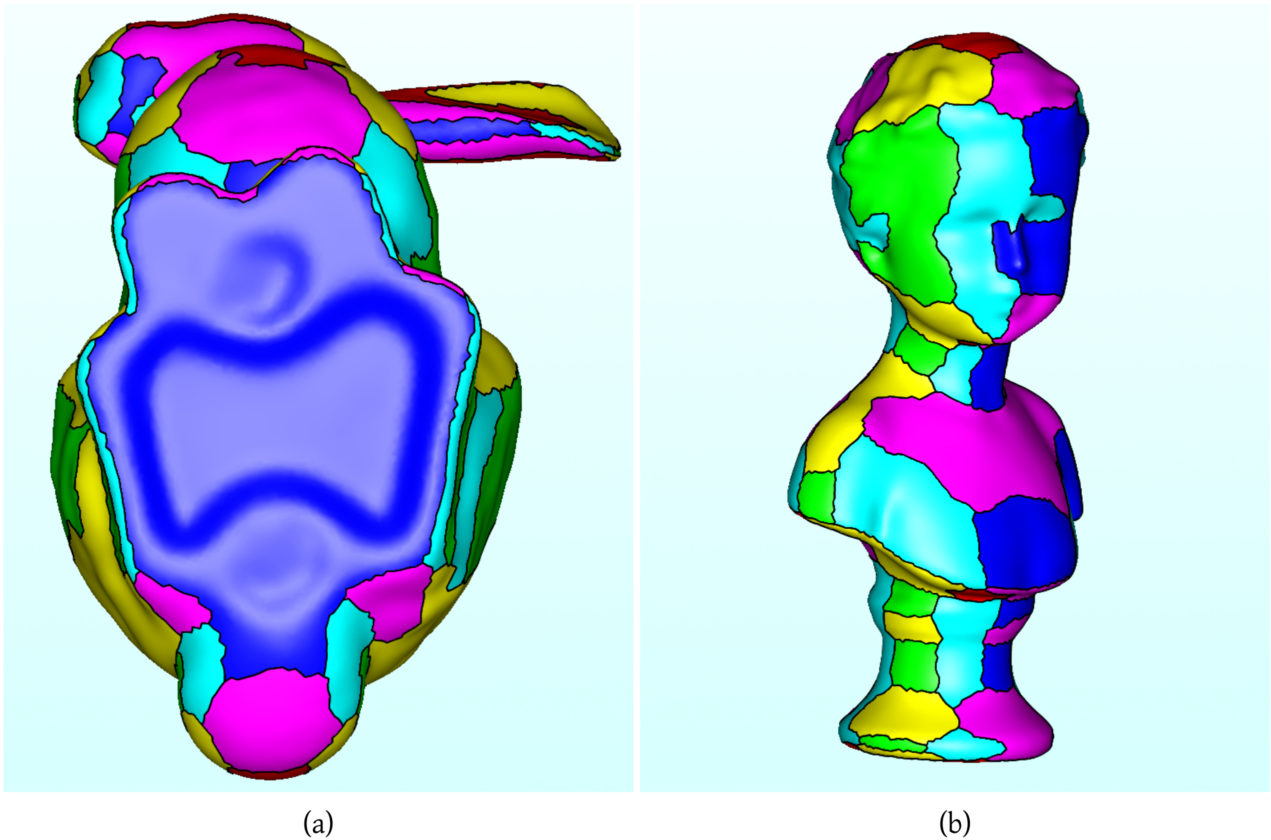


Figure 4.10: External boundary (blue) and internal boundary (green)

Doing these operations for every chart, until there are unvisited triangles, and inserting an edge on the graph for every pair of neighbors charts, we have build the graph of the segmentation. The last step is to find the bad charts. In our algorithm, a chart is bad if:

- has number of corners less than or equal to two;
- is a triangle strip and hasn't two neighbors with opposite label.

If the number of corner is less than or equal to two, we can't have a surface polygon, and that chart has to be eliminated. If the chart is a triangle strip, it may be noise on segmentation. However, graph-cut may inserted that chart with the only purpose to separate two charts with opposite labels, and in that case, the triangle strip must survive.



(a) (b)

Figure 4.11: Segmentation on Bunny and BU models after iterated bad patches deletion.

In all other cases, triangle strips are considered bad charts. Examples of bad charts can be seen in Figure 4.8 and 4.9 (gray charts).

Deleting Bad Charts

The only way to delete a bad chart is to merge it with one of its neighbors. The best way to do that is to select, from the set of neighbors labels, the closest label to the orientations of the triangles of the bad chart. Therefore, when a chart is bad we calculate the average of the normals of the triangles, and we calculate the dot product between the average normal and the orientations of the neighbors: the highest value of the dot product gives the new label. After that, it's necessary to recalculate all the graph $\mathcal{G}$: after deleting a bad patch is possible that other bad patches has been created, so is necessary to iterate this process, until there are no more bad patches (the process always converges). The result after the iteration is that in the segmentation there are only charts with minimum three corners, and noise is completely eliminated. Figure 4.11 shows segmentations after iterated bad patches deletion.

Chapter 5

Making Geometry

5.1 Goals

Given a good segmentation of a 3D model, the next step is to deform the model such a way that every chart is perfectly planar and perfectly oriented to the label of the chart. For doing this we first calculate a driven-deformation, that gives a triangle mesh not perfectly planar, but sufficient to determine the final geometry. From this model we can extract a coarse model characterized of only corner vertexes, that is composed of any kind of polygonal faces, not only triangles. The result is a coarse model very similar to the final result, but where faces are not perfectly oriented. On this model we apply an operation of edge collapsing, aimed to delete small edges not suitable for the cutting process. Finally, we force the orientation of every face, finding the final geometry.

5.2 Deformation

The first deformation is based on the work of Gregson et al. [GSZ11]. The purpose is to take every surface vertex v_i and rotate it in such a way that the vertex normal is closer to the label orientation of the chart, and the length of all edges is preserved.

The deformation is divided in two steps: in the first step we calculate for every surface vertex v_i, its frame field F_{v_i} (calculated on the vertex normal, that is the average of all adjacent triangle normals); in the second step we solve a linear system that gives the new coordinates for every surface vertex.

5.2.1 Compute

In the compute step, our purpose is to calculate, for every vertex v_i, a frame field F_{v_i} that allows to align the vertex normal $\overrightarrow{n_{v_i}}$ to a given target normal $\overrightarrow{t}$. To understand this argument, we start with a 2D example, shown in Figure 5.1. We have a segment (v_i, v_j) with normal $\overrightarrow{n_{ij}}$, and we want to align the segment to the y axis, then the target normal of the segment will be $\overrightarrow{t} = x$. Let θ the angle between the $\overrightarrow{n_{ij}}$ and $\overrightarrow{t} = x$ (Figure 5.1a). If we rotate the frame composed of the x and y axis by θ degrees, we obtain a frame $F = x', y'$ where, viewed by prospective of the original frame, the edge $\tilde{v}_i\tilde{v}_j$ is aligned to the target orientation (Figure 5.1b). For obtaining the new positions $\tilde{v}_i$ and $\tilde{v}_j$ on the original frame, we can use the dot product between two vectors u and w, that calculates the w projection on u. For preserving the edge length, we want that the projection of the edge v_iv_j will be equal to the projection of the edge $\tilde{v}_i\tilde{v}_j$. Therefore, we can consider the vector $v_i - v_j$ (Figure 5.1d),

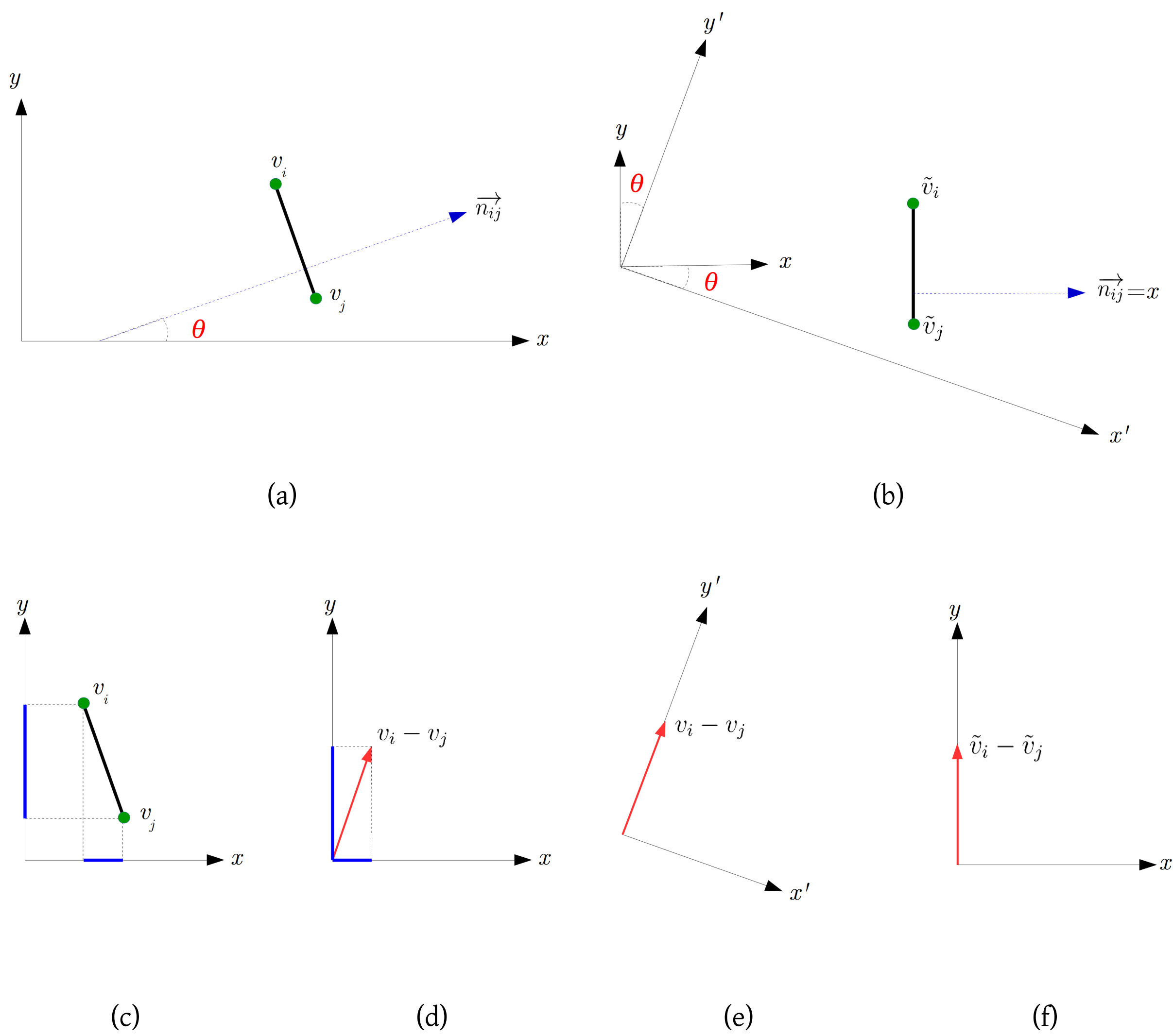


Figure 5.1: Rotation of a Frame

that has aligned projections in the rotated frame x', y' (Figure 5.1e), and these projections must be the same for the rotated vector $\tilde{v}_i - \tilde{v}_j$ in the original frame (Figure 5.1f). That means that:

$$(\tilde{v}_i - \tilde{v}_j)I = (v_i - v_j)F$$

where I is the matrix composed by the original axis x and y, and F is the **frame field** calculated as

$$F = \begin{bmatrix} x'_{(x)} & y'_{(x)} \\ x'_{(y)} & y'_{(y)} \end{bmatrix}$$

In other words, the original edge projections on the rotated frame must be the same of the rotated edge projections in the original frame. I is the identity matrix, therefore the equation becomes:

$$(\tilde{v}_i - \tilde{v}_j) = (v_i - v_j)F$$

We can directly extend this concept from 2D to 3D, considering also the rotation axis. Given a rotation angle θ and a rotation axis u, the rotation matrix is:

$$R = \begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}$$

where u_x, u_y and u_z are the components of vector $\overrightarrow{u}$. A 3D vector $\overrightarrow{v}$ can be rotated on an axis $\overrightarrow{u}$ by an angle θ in the following way:

$$\overrightarrow{v_r} = \overrightarrow{v}R$$

Therefore, the frame field F_{v_i} of the vertex v_i is calculated simply by rotating the base frame:

$$F_{v_i} = I \times R = R$$

where I represent the original frame x, y, z, and

$$\theta = \cos^{-1}(\overrightarrow{n_{v_i}} \cdot \overrightarrow{t_{v_i}})$$

$$\overrightarrow{u} = \overrightarrow{n_{v_i}} \times \overrightarrow{t_{v_i}}$$

where $\overrightarrow{n_{v_i}}$ is the normal of v_i vertex, and $\overrightarrow{t_{v_i}}$ is the target normal.

For vertexes that lies in edges of the segmentation, we have two or plus targets (if they are corner vertexes). For these vertexes, t_{v_i} is not on the set of target normals, but is calculated in the following way:

$$t_{v_i} = \frac{1}{m_{v_i}} \sum_{j=1}^{m_{v_i}} \overrightarrow{t_{t_j}}$$

where m_{v_i} is the number of incident triangles in v_i, and $\overrightarrow{t_{t_j}}$ is the target normal of the j-th incident triangle. In other words, the target normal of vertex v_i will be the average of all target normals of incident triangles.

Calculating the frame fields for every vertex is an operation that can be viewed like a gradient calculation of the input mesh: indeed, after the compute step, we have only local information of every vertex related with all incident triangles orientation, without taking into account spatial references such as vertex coordinates. Solving the linear system described in Subsection 5.2.2 is like an integrate operation, that recalculate coordinates for every vertex taking into account the rotation described by the frame field.

5.2.2 Integrate

Our goal now is to rotate every vertex in such a way that edges length is preserved. Therefore, for every pair of adjacent vertexes v_i and v_j, we want calculate the new vertexes position $\tilde{v}_i$ and $\tilde{v}_j$ in such a way that will be preserved the edge projection on the three axis:

$$\begin{cases} x: & \tilde{v}_i^{(x)} - \tilde{v}_j^{(x)} = (v_i - v_j)F^{(x)} \\ y: & \tilde{v}_i^{(y)} - \tilde{v}_j^{(y)} = (v_i - v_j)F^{(y)} \\ z: & \tilde{v}_i^{(x)} - \tilde{v}_j^{(z)} = (v_i - v_j)F^{(z)} \end{cases}$$

or, in a compact form

$$\tilde{v}_i - \tilde{v}_j = (v_i - v_j)F$$

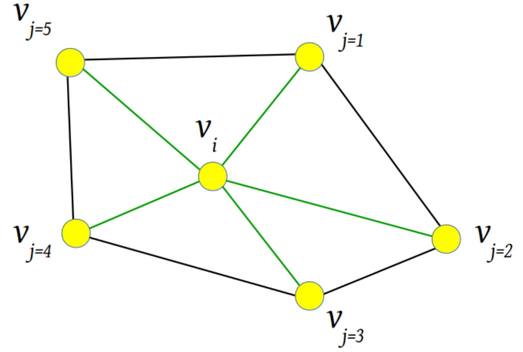


Figure 5.2: Neighbors vertexes of v_i

where F is the average between F_{v_i} and F_{v_j}.

Extending the concept on all v_j neighbors of v_i,
as shown in Figure 5.2, we can simply involve all neighbors using Laplacian weights:

$$L_j = \frac{1}{k_i}$$

where k_i is the number of v_i neighbors (for example, on Figure 5.2, $k_i = 5$). Therefore, the system for a vertex becomes:

$$\begin{cases} x: & \frac{1}{k_i}\sum_{j=1}^{k_i}\left(\tilde{v}_i^{(x)} - \tilde{v}_j^{(x)}\right) = \frac{1}{k_i}\sum_{j=1}^{k_i}\left((v_i - v_j)F^{(x)}\right) \\ y: & \frac{1}{k_i}\sum_{j=1}^{k_i}\left(\tilde{v}_i^{(y)} - \tilde{v}_j^{(y)}\right) = \frac{1}{k_i}\sum_{j=1}^{k_i}\left((v_i - v_j)F^{(y)}\right) \\ z: & \frac{1}{k_i}\sum_{j=1}^{k_i}\left(\tilde{v}_i^{(x)} - \tilde{v}_j^{(z)}\right) = \frac{1}{k_i}\sum_{j=1}^{k_i}\left((v_i - v_j)F^{(z)}\right) \end{cases}$$

Like a standard integration, we have one liberty degree representing possible translations of the input model. For this reason, we need to add another equation, where we force the position of one vertex:

$$\tilde{v}_0 = v_0$$

Therefore, our system has the form

$$Ax = b$$

where A is a $(n+1)\times n$ matrix, and n is the number of vertexes. In the i-th row, we will have 1 in the i-th column, and $-\frac{1}{k_i}$ in every j column, $j \in N(i)$, plus a row with 1 only in column 0, that is the row where we fix the position of vertex v_0. x will have n rows and three columns, one for every coordinate value (x, y, z). b will have $n+1$ rows and three columns. In the i-th row there will be the sum over all v_i neighbors of the projections $(v_i - v_j)$ in the average frame F multiplied with the Laplacian weight, plus a row with the coordinates of the fixed v_0 vertex. Therefore, we solve three systems, one for each coordinate (x, y, z).

In the example in Figure 5.2, we will have the following row in the A matrix:

$$A = \begin{matrix} \\ \vdots \\ i \\ \vdots \end{matrix}\begin{matrix} \cdots & v_i & \cdots & v_{j=1} & \cdots & v_{j=2} & v_{j=3} & v_{j=4} & v_{j=5} & \cdots \\ \end{matrix}$$

$$\begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & 1 & \cdots & 1/5 & \cdots & 1/5 & 1/5 & 1/5 & 1/5 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

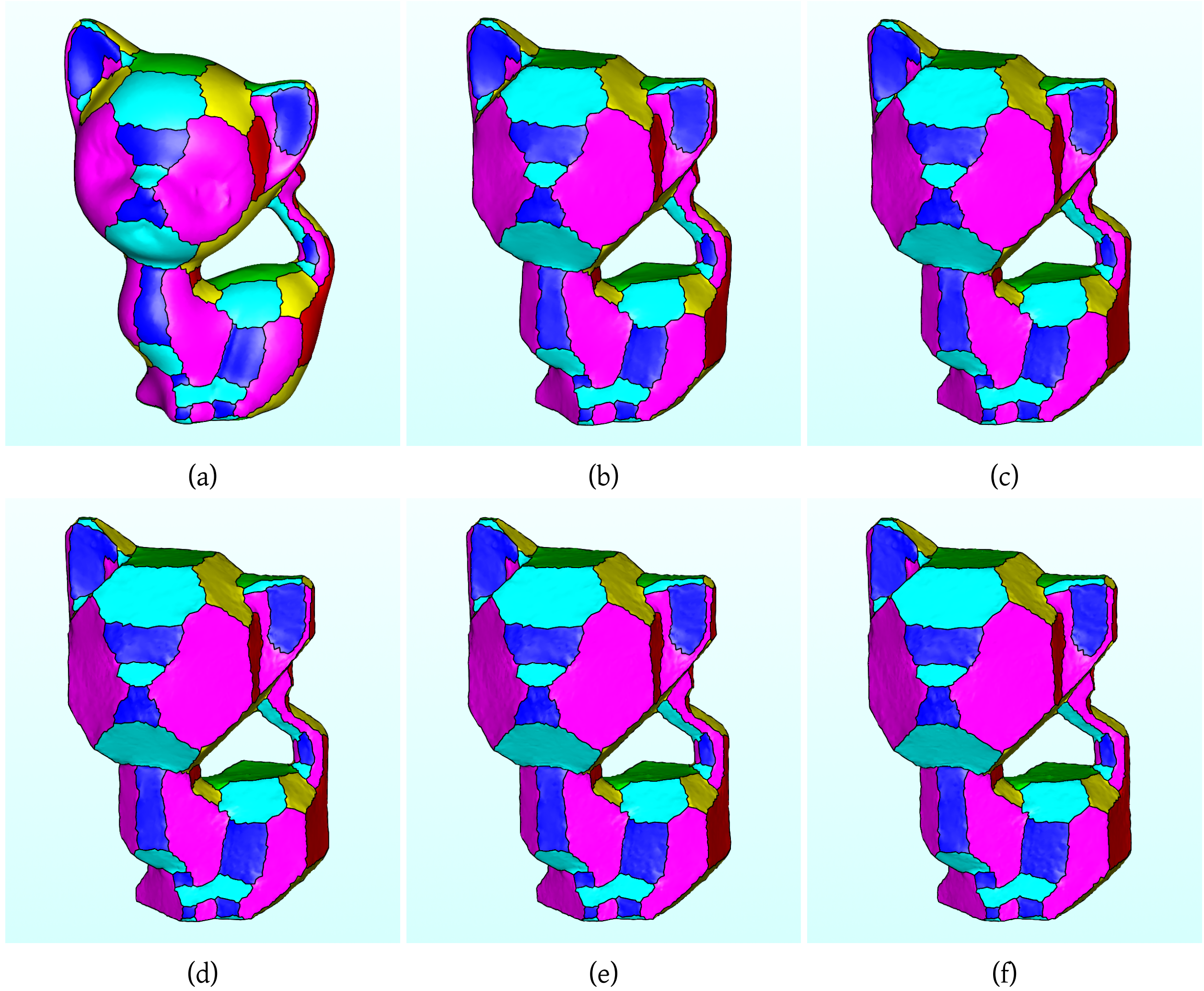


Figure 5.3: Deformation of kitten model iterated 5 times

This system doesn't have a solution, therefore we solve this system using the Ordinary Least Squares method, that allows to find the solution that minimizes the distance with the ideal solution. The Least Squares system becomes:

$$A^T Ax = A^T b$$

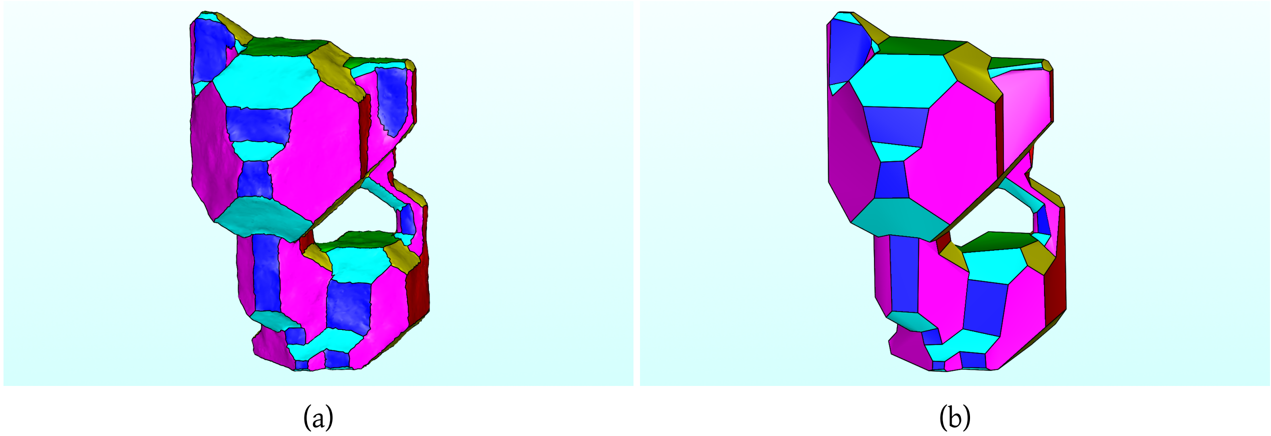
where the solution is

$$x = (A^T A)^{-1} A^T b$$

In this way, the solution is unique: if A is a $n \times m$ matrix, then $A^T A$ will be a $m \times m$ matrix, and $A^T b$ will be a column vector of m terms.

5.2.3 Iterated Deformation

Since the deformation is solved by Ordinary Least Squares, the resulting mesh will be not perfectly deformed, in fact this deformation only flattens triangles in a "gentle" way. Is possible to have a better (but not sufficient) deformation by iterating for a number of times this deformation. Figure 5.3 shows the deformation applied 5 times.



(a) (b)

Figure 5.4: Triangle model and Coarse Model

5.2.4 Coarse Model Extraction

Starting from this point of the algorithm, we can operate using only the coarse model, that is the model composed by only corner vertexes on the deformed model segmentation. A model with these features has a face for every chart in the starting segmentation, and a surface composed by polygons of any possible form, and not only triangles. Figure 5.4 shows the two bunny models before and after this step: Figure 5.4a shows the triangle mesh deformed, and Figure 5.4b shows the coarse model extracted from the triangle mesh.

5.3 Edge Collapsing

The purpose of the Edge Collapsing is to remove remaining noises from graph-cut segmentation and to remove faces and edges that are irrelevant compared to the total size of the model (Figure 5.5). The maximum length δ of a collapsable edges is calculated as a fraction of the bounding box diagonal of the model. We are starting from a model where every vertex is incident in at least three different faces, and one of our purposes is to obtain, at the end of edge collapsing, a model with the same features, where vertexes have at least three incident faces. Indeed, vertexes with only two incident faces are useless for our purposes, and can only cause noise.

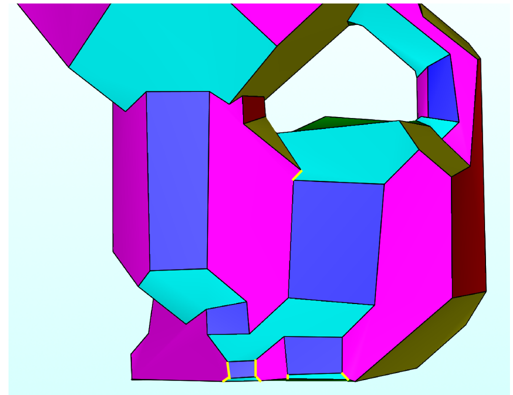


Figure 5.5: Collapsable Edges

Assuming that our model is described by a DCEL data structure, we can assume that a collapsable edge is described by two half edges e_1 and e_2, incident in two vertexes v_1 and v_2 and in two faces (that are arbitrary polygons) f_1 and f_2. Edge collapsing consists, at least, of the deletion of half edges e_1 and e_2, and vertexes v_1 and v_2, substituted by a new vertex v (Figure 5.6).

In our implementation, all the edges having length $d_i \leq \delta$ are inserted in a list l. All edges on this list are processed starting from the smallest. We first must check if the edge is collapsable. A edge is collapsable if its incident vertexes are not incident in two faces with opposite labels. These edges, even if sufficiently small, are in the model with the purpose to separate two faces with opposite

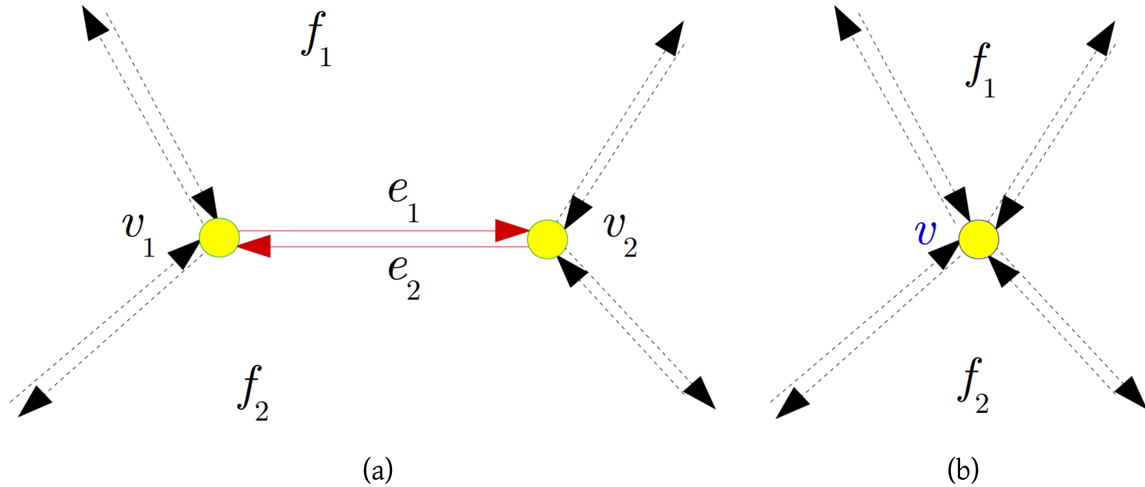


Figure 5.6: Before (a) and After (b) Edge Collapsing if there are no incident triangles

labels, and cannot be collapsed. Otherwise, we have to distinguish between three different cases, described in the next subsections.

5.3.1 Case 1: No Incident Triangles

The first case is represented in Figure 5.6a. Faces f_1 and f_2 are polygons with at least four edges. This case can be simply handled by creating a new vertex v that, by approximation, has average coordinates between v_1 and v_2. We add all v_1 and v_2 incident edges in v and we delete half edges e_1 and e_2 and vertexes v_1 and v_2. No faces will be deleted, therefore all vertexes remains corner vertexes with at least three incident faces (Figure 5.6b).

5.3.2 Case 2: One Incident Triangle

The second case is represented in Figure 5.7. In this case, we must collapse half edges e_1 and e_2, and face f_1. We can create a vertex v as average of v_1 and v_2, and this new vertex will have at least three incident faces: even without f_1, we know that v_1 is incident at least in other two faces (f_2 and another one at least, for example f_3), and the same for v_2 (f_2 and another one at least, for example f_4). After f_1 deleting, new vertex v will be incident in these faces (f_2, f_3 and f_4 at least).

After deleting f_1, v_3 will have one less incident face: we must differentiate between the case when v_3 has at least four incident faces, and the case when v_3 has only three incident faces.

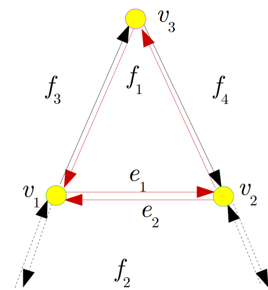


Figure 5.7: One Incident Triangle

Case 2a

If v_3 has at least four incident faces before collapsing (Figure 5.12a), we have to create the new vertex v as average of v_1 and v_2, adjust all incidences in v, and delete v_1, v_2, e_1, e_2, f_1 and all incident half edges in f_1. After these operation, v_3 is still a corner vertex with at least three incident faces (Figure 5.12b).

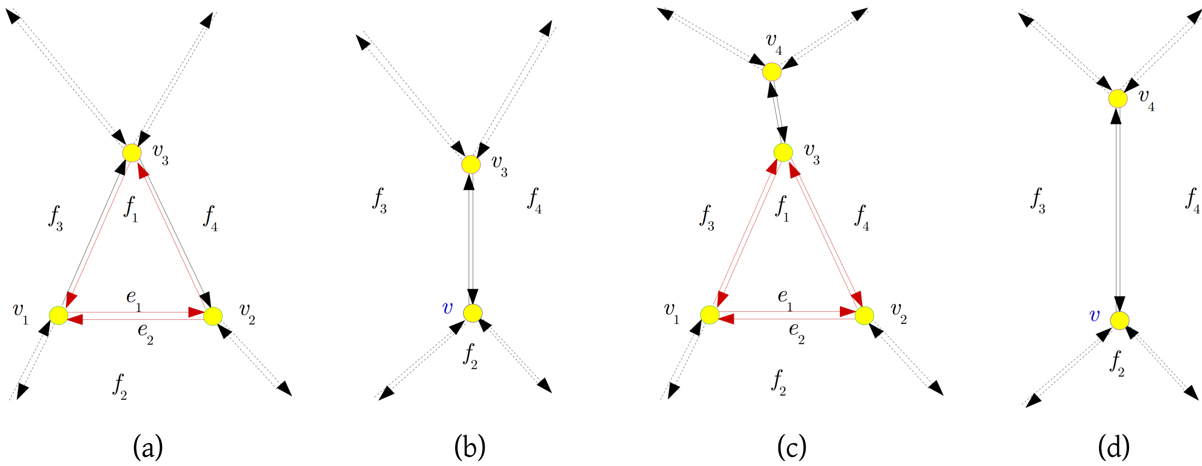


Figure 5.8: Edge collapsing cases (a, c) and solutions (b, d) if there is one incident triangle

Case 2b

If v_3 has only three incident faces (Figure 5.8c), after collapsing will not be a corner vertex because we deleted f_1. Therefore, we have to considerate the remaining v_3 adjacent vertex, that we call v_4. We assume that v_4 is a corner vertex, and then has at least three adjacent faces. In this case, in addition to the operation performed in the previous case, we must delete all half edge twins of incident half edges in f_1 and vertex v_3. The new vertex v will be linked to v_4, that was and still remains a corner vertex (Figure 5.8d).

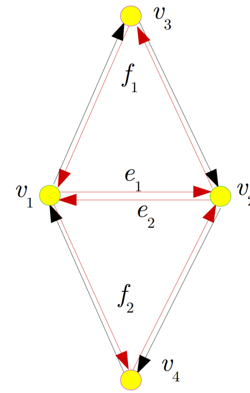


Figure 5.9: Two Incident Triangles

5.3.3 Case 3: Two Incident Triangles

The third case is represented in Figure 5.9. This is a difficult case to deal with, because there are a different (similar) sub-cases to consider: we must delete two faces (f_1 and f_2), that involve four vertexes (v_1, v_2, v_3 and v_4). We have to handle six different sub-cases:

a) The union of v_1 and v_2 incident faces is composed of at least five faces, and both v_3 and v_4 have at least four incident faces (Figure 5.10a);

b) The union of v_1 and v_2 incident faces is composed of at least five faces, v_3 (v_4) has three incident faces and v_4 (v_3) has at least four incident faces (Figure 5.10b);

c) The union of v_1 and v_2 incident faces is composed of at least five faces, and both v_3 and v_4 have only three incident faces (Figure 5.10c);

d) The union of v_1 and v_2 incident faces is composed of only four faces, and both v_3 and v_4 have at least four incident faces (Figure 5.10d);

e) The union of v_1 and v_2 incident faces is composed of only four faces, v_3 (v_4) has three incident faces and v_4 (v_3) has at least four incident faces (Figure 5.10e);

f) The union of v_1 and v_2 incident faces is composed of only four faces, and both v_3 and v_4 have only three incident faces (Figure 5.10f);

All cases are handled as explained in the following sections.

Case 3a

This case (Figure 5.10a) is the simplest: we create a new vertex v as average of v_1 and v_2 (adjusting the half edge incidences), and we delete v_1, v_2, f_1, f_2 and all incident half edges on these two faces. All other incident faces on v_1 and v_2 will become incident faces in v, that are at least three and therefore v is a corner vertex. v_3 and v_4 had at least four incident faces, therefore now they have at least three incident faces and they are corner vertexes (Figure 5.11a).

Case 3b

This case (Figure 5.10b) is similar to Case 2a, when only one triangle is incident in the collapsable edge: after executing the operations described in the Case 3a, v_3 is no longer a corner. Therefore v_3 must be eliminated, and the new vertex v will be linked to the remaining v_3 adjacent vertex v_5, that still remains a corner vertex (Figure 5.11b).

Case 3c

This case (Figure 5.10c) is the basically the same of Case 3b, where both v_3 and v_4 must be eliminated. Therefore, new vertex v will be linked to v_5 and v_6. All these vertexes remains corner vertexes (Figure 5.11c).

Case 3d

In this case (Figure 5.10d) the union of v_1 and v_2 incident faces is composed of only four faces: after delete f_1 and f_2, new vertex v couldn't have three incident faces at least, and therefore couldn't be a corner vertex. In this case we don't create a new vertex v: we just delete v_1, v_2, f_1, f_2 and all incident edges on these faces, and we simply link v_3 and v_4, that still are corner vertexes (Figure 5.11d).

Case 3e

This case (Figure 5.10e) is the union of Cases 3b and 3d. As in Case 3d, vertex v will not be created, v_3 will be deleted because isn't a corner vertex, and vertex v_4 is directly linked with v_5, that still are corner vertexes (Figure 5.11e).

Case 3f

This case (Figure 5.10f) is the union of Cases 3c and 3d. As in case 3d, vertex v will not be created, v_3 and v_4 will be deleted because aren't corner vertexes, and vertex v_6 is directly linked with v_5, that still are corner vertexes (Figure 5.11f).

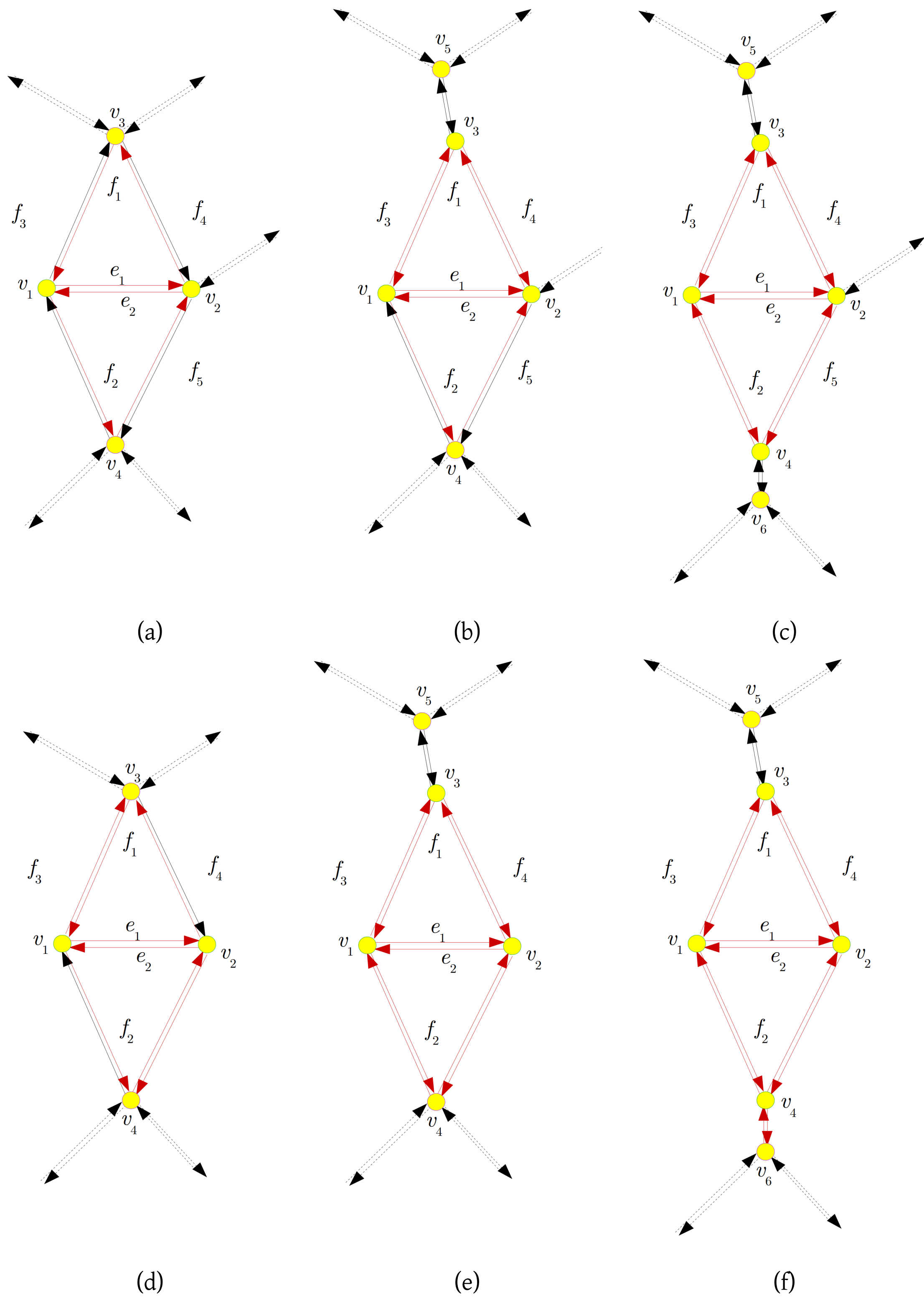


Figure 5.10: Six different cases of edge collapsing if there are two incident triangles

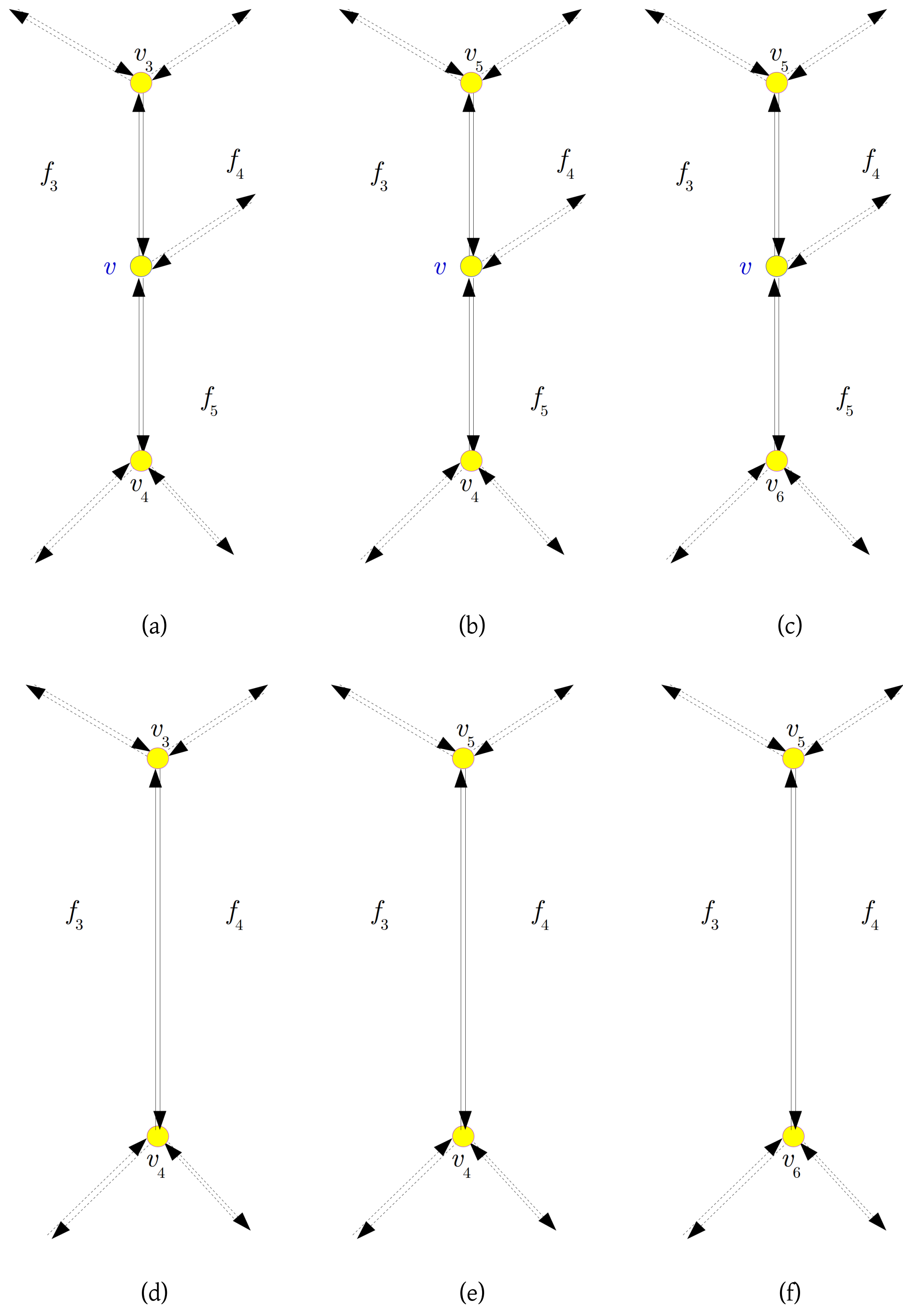
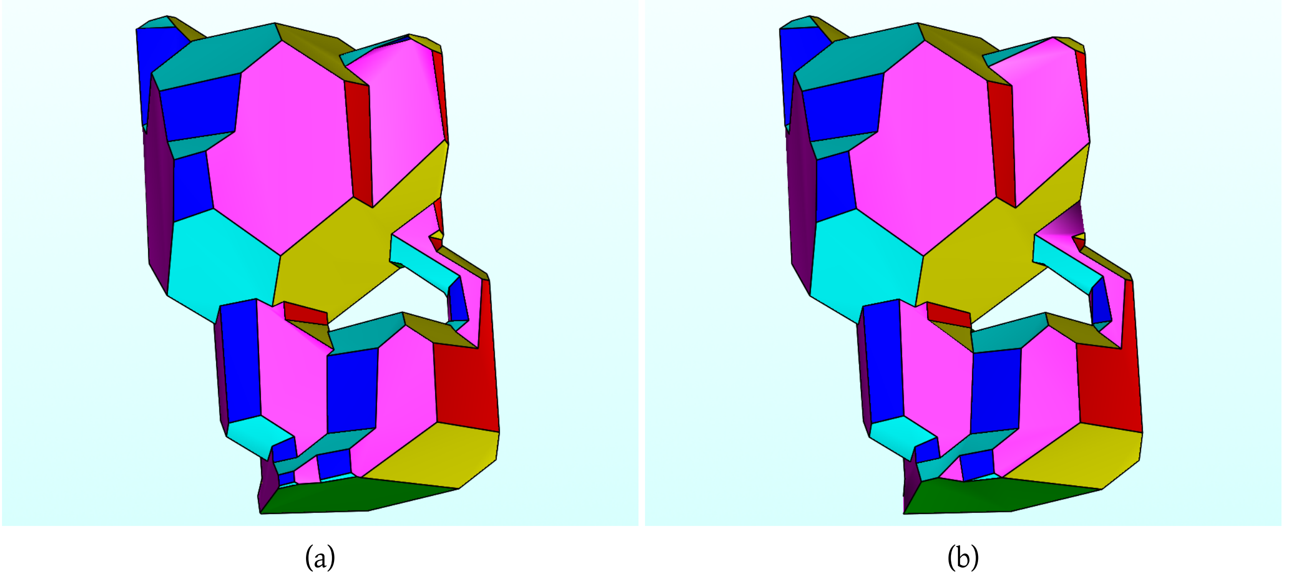


Figure 5.11: Six different cases after edge collapsing



(a) (b)

Figure 5.12: Kitten model before (a) and after (b) edge collapsing

5.3.4 Results

A result of a model where is executed edge collapsing is shown in Figure 5.12. In our algorithm, edge collapsing is executed repeatedly with planarization (described in Section 5.4). In this way, we are able to identify and eliminate edges that are collapsing after a given number planarization iterations.

5.4 Planarization

Last step of our algorithm, that we called "Planarization", has the goal to return all charts perfectly planar and oriented to their label normal, starting from a model where charts have orientations similar but not equals to the goal orientations. Starting orientations are inherited from iterated deformation on the triangular mesh (Section 5.2) and edge collapsing (Section 5.3), that modifies orientations when a new vertex is created as average of two old vertexes.

Planarization consist of solving a system of equations, where every vertex is related to its vertex neighbors, and every chart is forced to be tangent with a plane oriented on the right normal. The system of equation is of the type $Ax = b$, and a simple scheme that represent it is shown in Figure 5.13.

The equation of a generic plane is

$$ax + by + cz - d = 0$$

where a, b and c are the component of the normal to the plane. Therefore, given a face f_j that we want to orient to the normal a_j, b_j, c_j and a set of vertexes incident to f_j, our unknowns are all the vertexes coordinates and the d_j terms that fixes all the incident vertexes in the plane

$$a_j x + b_j y + c_j z - d_j = 0$$

Then, we have $3n_v + n_f$ unknowns, where n_v is the number of vertexes (three unknowns for every vertex: one for x coordinate, one for y coordinate and one for z coordinate) and n_f is the number of faces.

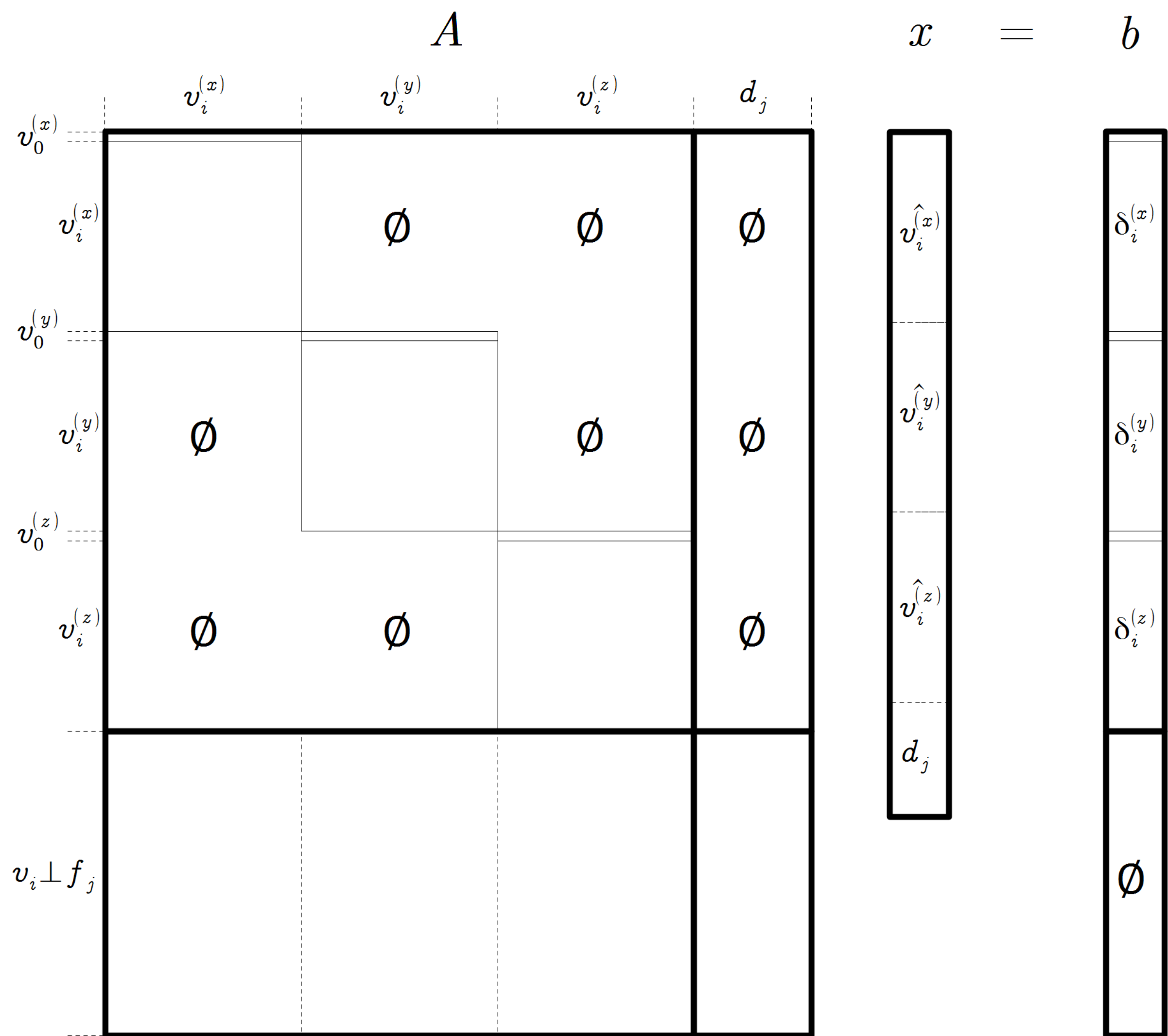
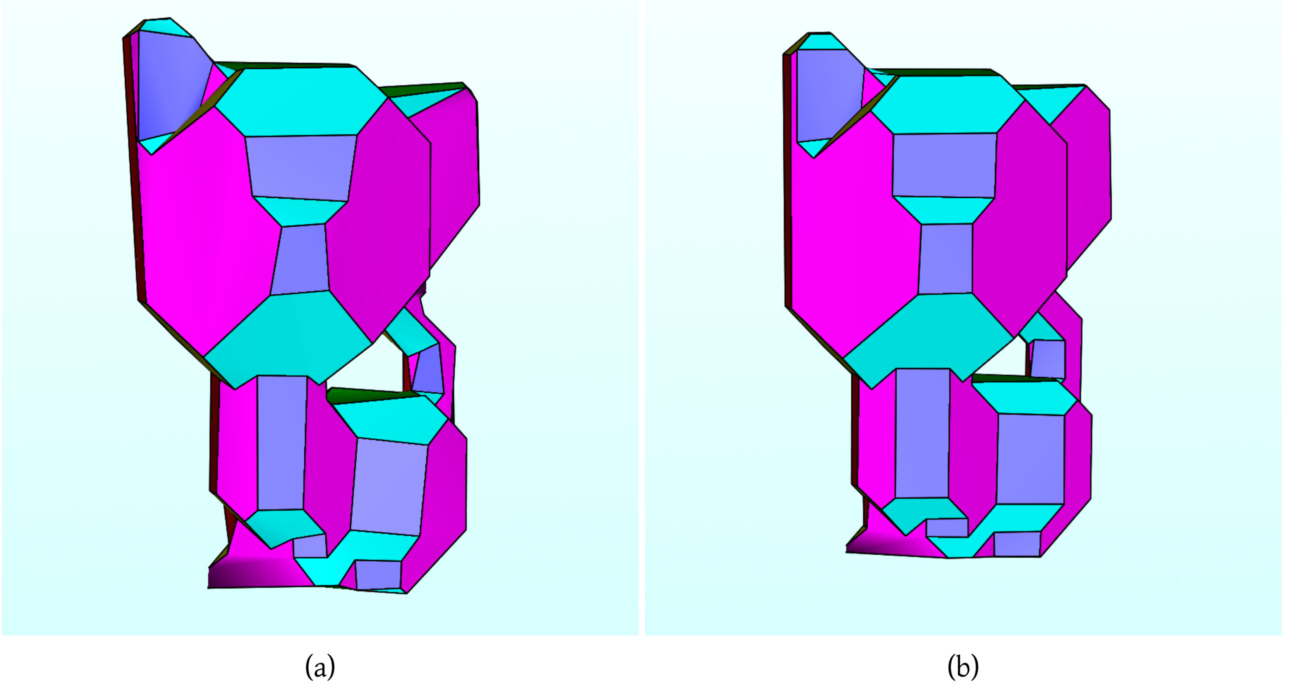


Figure 5.13: Scheme of the Planarization System



(a) (b)

Figure 5.14: Kitten model before (a) and after (b) planatization

As explained by Sorkine [Sor06], we can relate every vertex with its neighbors using δ-coordinates. Given a vertex v_i, δ-coordinates of v_i are the difference between absolute coordinates of v_i and the center of mass of its neighbors:

$$\delta_i = \left(\delta_i^{(x)}, \delta_i^{(y)}, \delta_i^{(z)}\right) = v_i - \frac{1}{k_i} \sum_{j \in N(i)} v_j$$

where $N(i)$ is the set of index neighbors of v_i and k_i is the number of v_i neighbors. We want that these center of mass remains basically the same, therefore, for every vertex we put the following constraint in the system:

$$\hat{v}_i - \frac{1}{k_i} \sum_{j \in N(i)} \hat{v}_j = \delta_i$$

where $\hat{v}_i$ are the vertex coordinates unknowns. Like the deformation system, also in this case passing from absolute to δ-coordinates we lose the information of the position of the model in the space. Therefore, returning to absolute coordinates has a liberty degree given by all possible translations of the model in the space. Also in this case, to have as result a planarized model in the same position of the input model, we need to fix the coordinates of a vertex v_0 by simply adding the following constraint in the system:

$$\hat{v}_i = v_i$$

Unlike the deformation systems, in the planarization system the coordinates of all vertex are related by the plane equations (described in the following paragraphs), therefore we can't solve three separate systems, but we must solve a single big system where there are all x, y and z coordinates of all vertexes.

Last rows in the A matrix and b vector are related to the planar constraints. We used the symbol $\perp$ for the incidence relation: $v_i \perp f_j$ means that vertex v_i is incident in face f_j.

Let a_j, b_j, c_j, d_j the components that describes the plane associated to the face f_j. If vertex v_i is incident in face f_j, we have to add the following constraint in the system:

$$a_j\hat{v}_i^{(x)} + b_j\hat{v}_i^{(y)} + c_j\hat{v}_i^{(z)} - d_j = 0$$

that imposes that v_i lies in a plane with normal a_j, b_j, c_j, with unknown d_j.

For every vertex v_i, there are many equations as the number of its incident faces:

$$\hat{v}_i = \{\hat{v}_i^{(x)}, \hat{v}_i^{(y)}, \hat{v}_i^{(z)} \mid \forall j : \hat{v}_i \perp f_j, a_j\hat{v}_i^{(x)} + b_j\hat{v}_i^{(y)} + c_j\hat{v}_i^{(z)} - d_j = 0\}$$

where $\hat{v}_i^{(x)}, \hat{v}_i^{(y)}, \hat{v}_i^{(z)}, d_j$ are the unknowns.

For a vertex in a mesh, the number of incident faces is equal to the number of incident edges, and every edge is incident in two vertexes, therefore the number of planar constraints is $2n_e$, where n_e is the number of edges.

Therefore, the size of A matrix is $(3n_v + 2n_e) \times (3n_v + n_f)$, and is composed of the coefficients that multiplies unknowns in every equation.

Also in this case, this system has no exact solution, then we solve that iterating Ordinary Least Squares solutions. A model before and after the planarization is shown in Figure 5.14.

Chapter 6

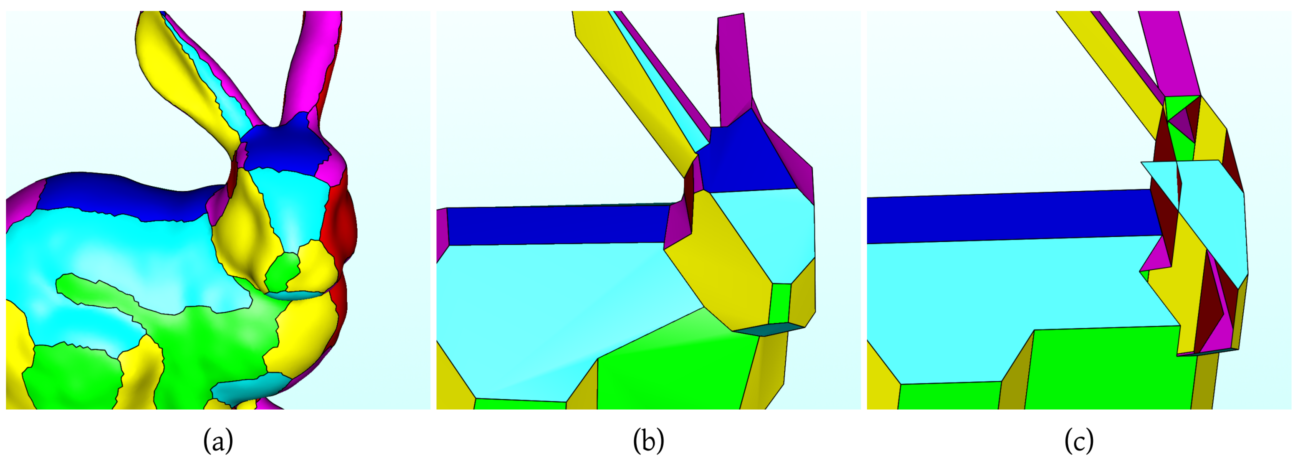
Results

Unfortunately, after we have implemented all the modules of the project, we realized that there are other topology constraints that must be met to obtain physically developable models. These constraints are very difficult to identify and solve in our method, because are global features impossible to avoid with graph-cut.

On Figure 6.1 is shown an example of these problems. In the bunny head, after planarization, the 3D model viewer can't visualize blue and yellow charts because are not simple polygons, and there are two overlapping edges (Figure 6.1c). This happens because there is an incident edge in the blue and yellow charts (the right magenta chart is collapsed) that separates cyan and red charts (Figure 6.1b). But these charts wants simply to intersect each others, making impossible to exist the edge incident in blue and yellow charts. Rather, there should be an edge incident in cyan and red charts.

This problem starts on segmentation time (Figure 6.1a), and is very hard to automatically identify it because it depends on all the geometry of the model (in this case, the bunny head shrinks due to charts in the back), and occur after planarization, when we obtain the final model.

However, there are other models that don't have these problems. Algorithm times (seconds) of some test models are shown in 6.1 and Algorithm steps and results are shown in the next pages. On these results, are executed five iterations of deformation and two iterations of edge collapsing followed by planarization. In the total time, in addition to the single steps times, are counted times for loading the input and for saving outputs.

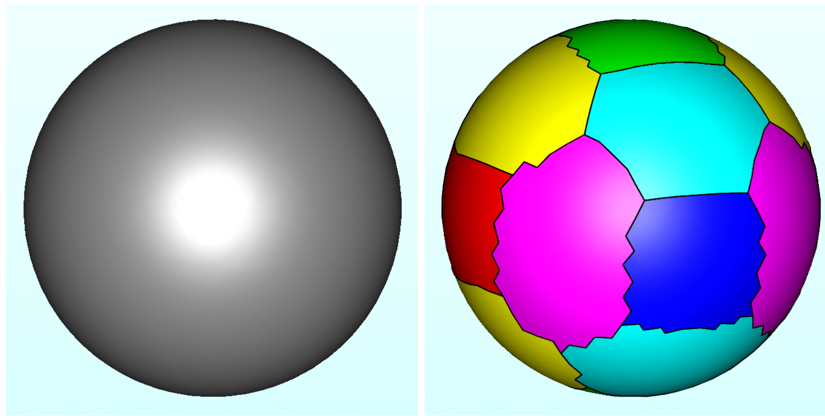


(a) (b) (c)

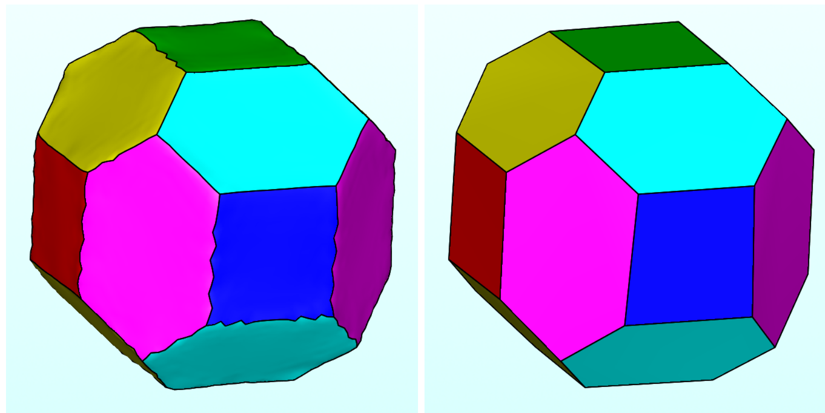
Figure 6.1: Problems with head bunny model

Models	Number of Vertexes	Graph-Cut	Deleting Bad Charts	Deformation	Edge Collapsing and Planarization	Total
Sphere	5,762	0.462563	0.016439	1.05252	0.021953	1.7065
Abstract Sculpture	8,286	0.606282	0.047811	2.18808	0.082549	3.14733
Bunny	28,088	4.65914	0.264375	7.97025	0.086137	13.6832
Kitten	24,979	3.24461	0.218971	5.52845	0.092082	9.7585
Holes	7,982	0.807621	0.02452	1.38435	0.047647	2.50787
BU	36,224	5.71704	0.356566	10.7044	0.066344	17.715

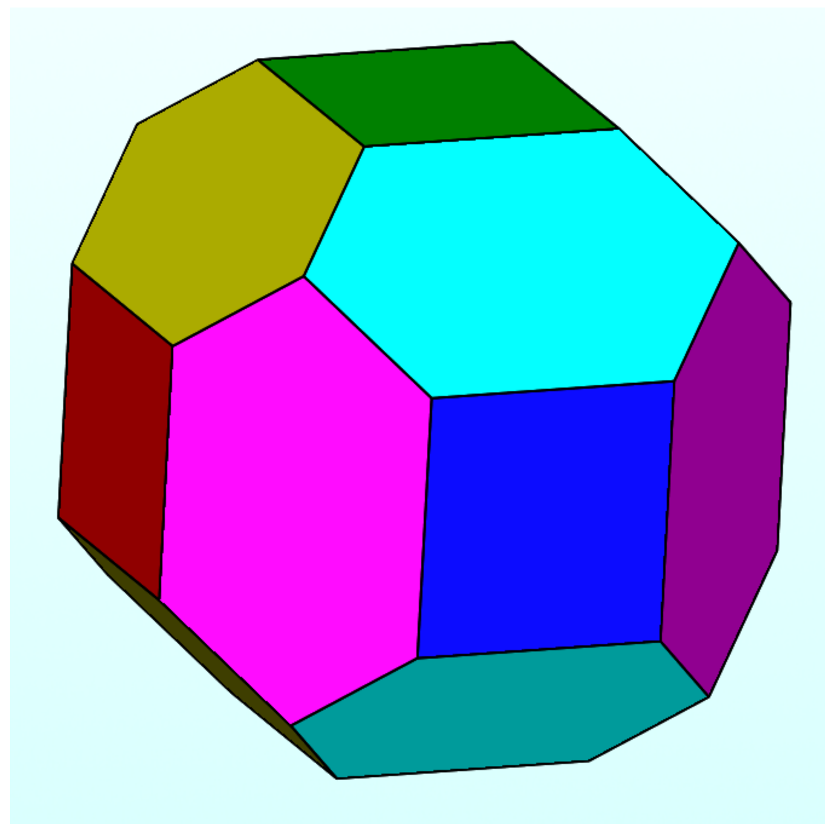
Table 6.1: Execution Times



(a) Input model (b) Segmented model

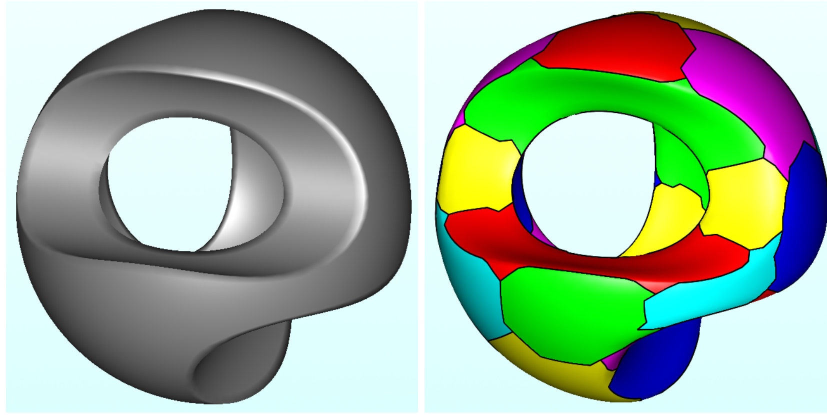


(c) Deformed model (d) Coarse model

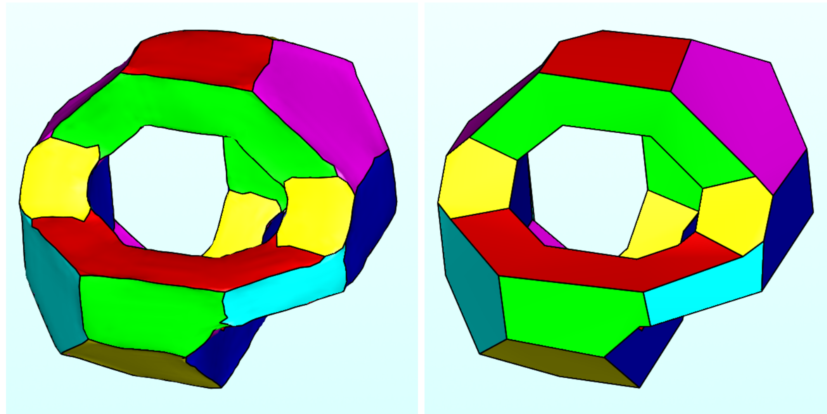


(e) Collapsed and Planarized model

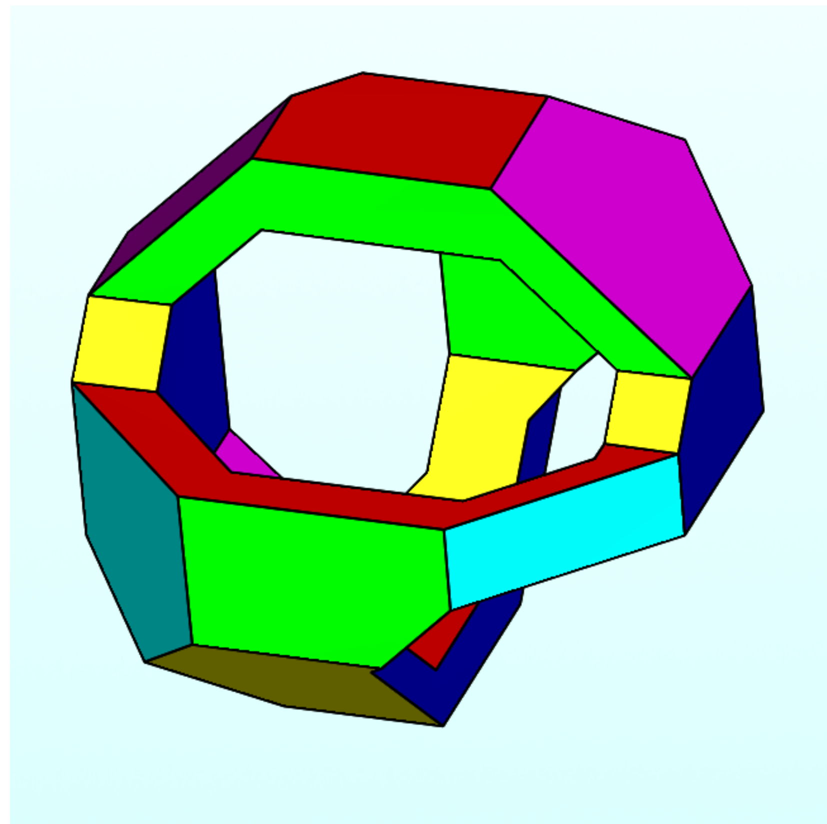
Figure 6.2: Sphere Model



(a) Input model (b) Segmented model

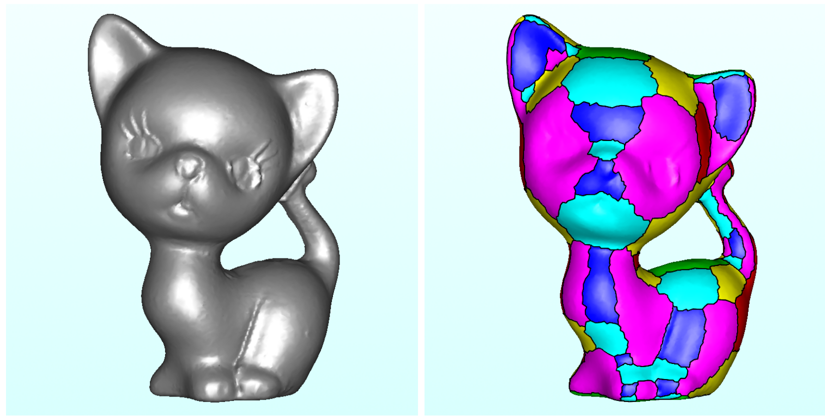


(c) Deformed model (d) Coarse model

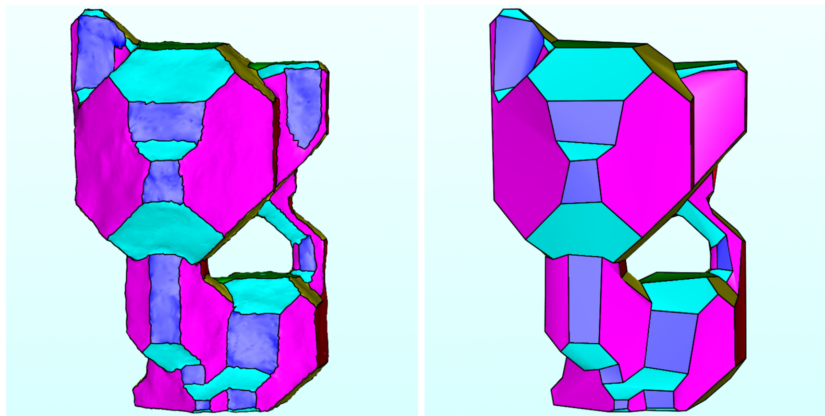


(e) Collapsed and Planarized model

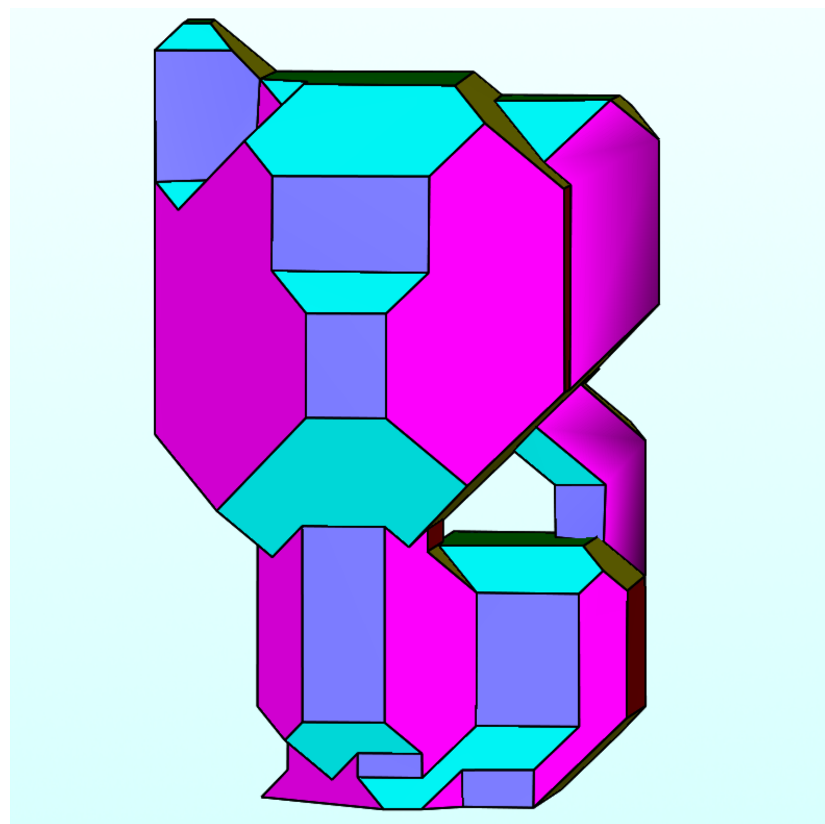
Figure 6.3: Abstract Sculpture Model



(a) Input model (b) Segmented model

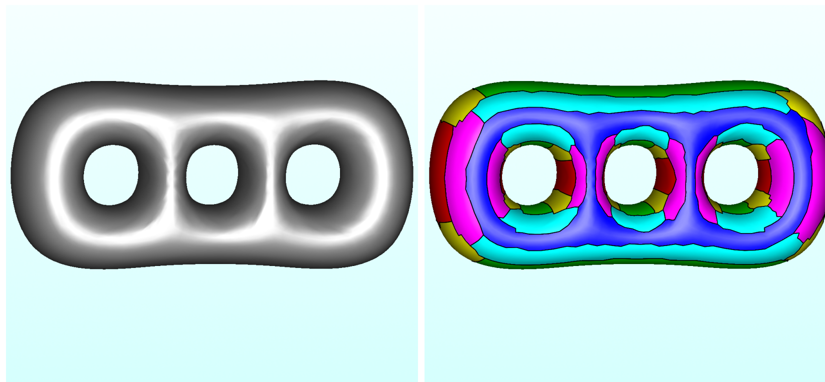


(c) Deformed model (d) Coarse model

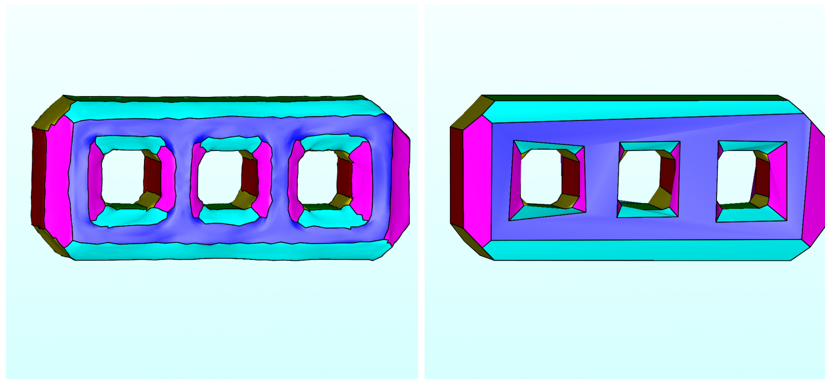


(e) Collapsed and Planarized model

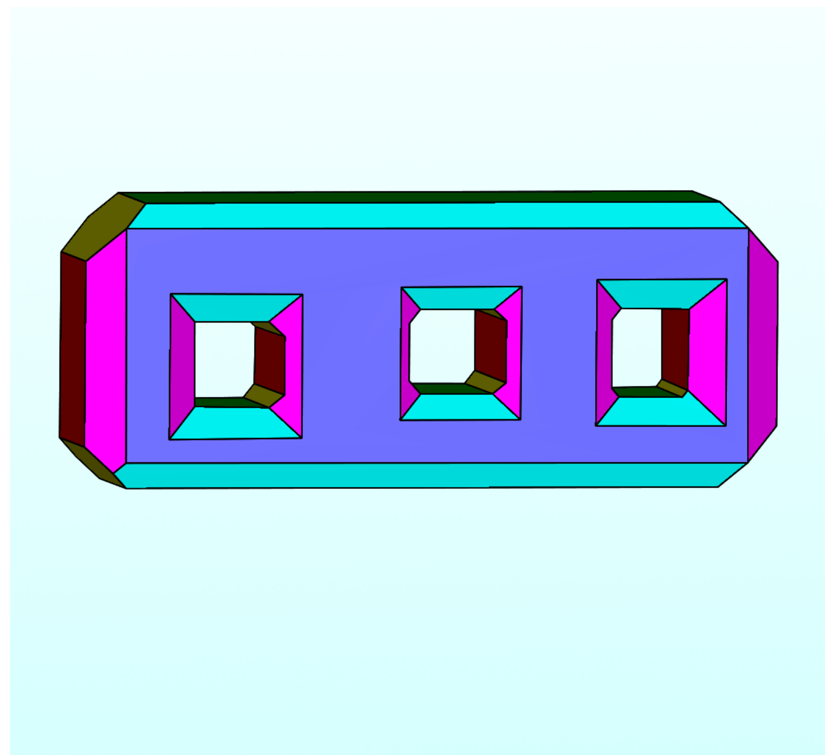
Figure 6.4: Kitten Model



(a) Input model (b) Segmented model

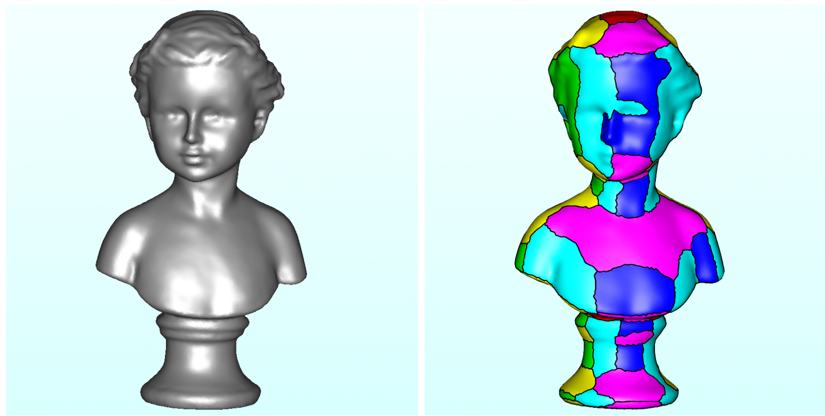


(c) Deformed model (d) Coarse model

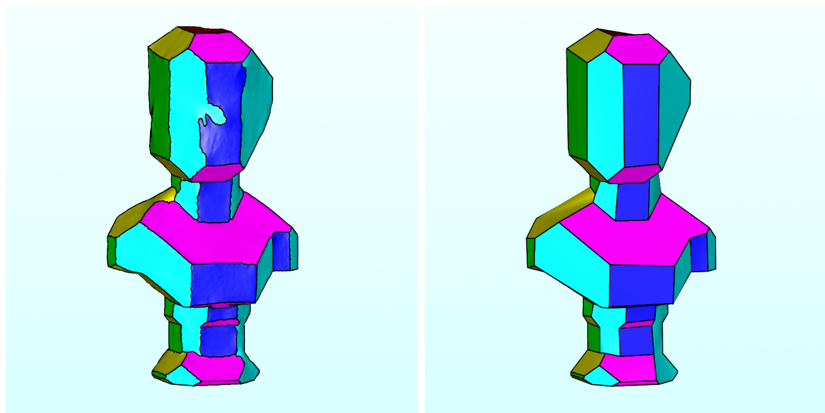


(e) Collapsed and Planarized model

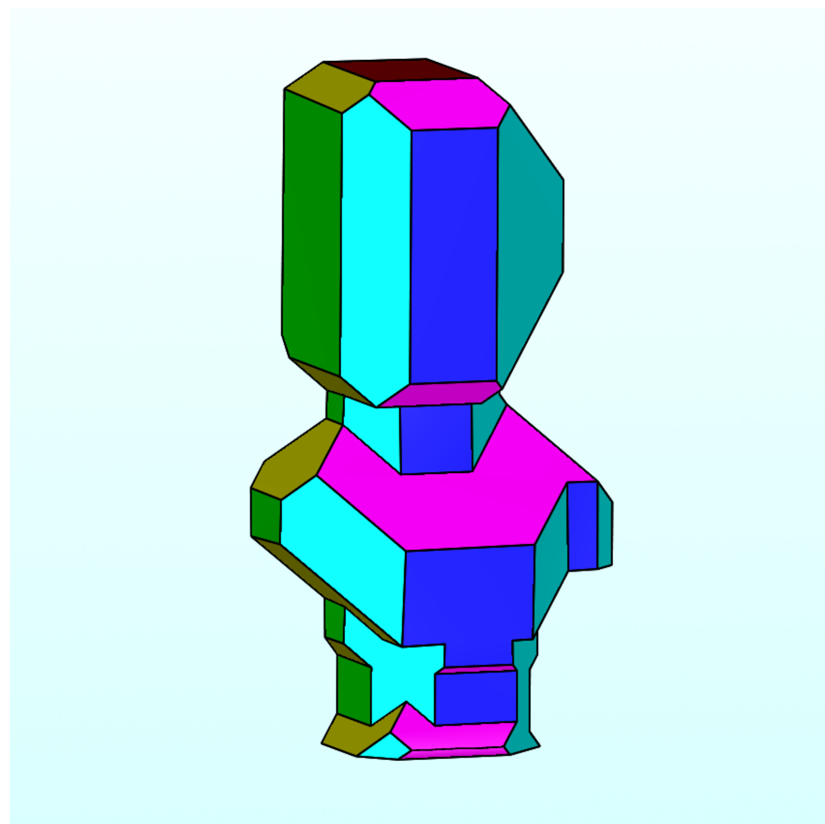
Figure 6.5: Holes Model



(a) Input model (b) Segmented model



(c) Deformed model (d) Coarse model



(e) Collapsed and Planarized model

Figure 6.6: BU Model

Chapter 7

Conclusions and Future Works

The goal of this thesis, given a generic 3D model, was to obtain a coarse 3D model that had good features that allows a simplified construction process using rigid materials like wood or thick paper. The resulting algorithm developed during this work is an algorithm that doesn't work with all possible models. This fact is consequence of a bad segmentation obtained using an optimization algorithm that is general and doesn't solve the problem from a geometric point of view.

We tried to solve this gap using several techniques (e.g. edge collapsing, an ideal angle collapsing (that collapses where there is an angle less than a given threshold angle) or the alternation of edge collapsing and planarization), but unfortunally these techniques are not sufficient. Trying to solve these problems using other different post-processing starting from a segmentation obtained with Graph-Cut is a very hard problem. These facts suggests that Graph-Cut is not the best way to obtain a good segmentation. Our purpose is to have a segmentation with rare imperfections which are easy to handle and solve (without, for example, handle too many cases of edge collapsing). Therefore, a basic future work is to find another way to obtain better segmentations.

Understand if a segmentation is physically developable by simply looking at it is difficult: a possible future work can be the formalization of the concept of "good segmentation", where every chart has to be planar, and eventually the developing of an algorithm that understands if a segmentation is good for our purposes.

Another interesting future work is to formalize a optimization problem that, given a planarized model and the dimensions of a sheet of wood/paper, gives in output the minimum number of sheets containing the unfolded faces that can be adjacent during the milling process (Figure 7.1).

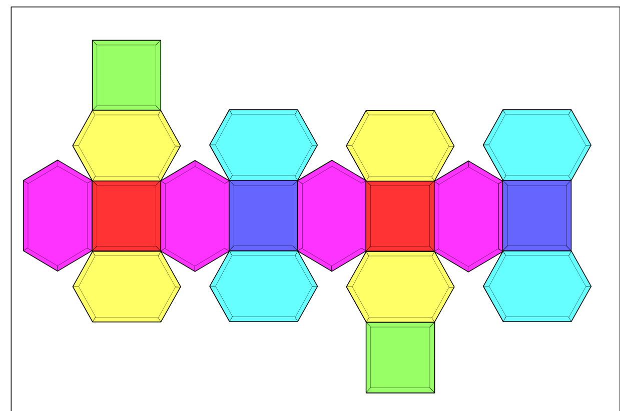


Figure 7.1: Unfolded Sphere Model

Bibliography

[Aut14] AUTODESK: 123d make. http://www.123dapp.com/make, 2014.

[BK04] BOYKOV Y., KOLMOGOROV V.: An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *Pattern Analysis and Machine Intelligence 26*, 9 (2004), 1124–1237.

[BVZ01] BOYKOV Y., VEKSLER O., ZABIH R.: Fast approximate energy minimization via graph cuts. *Pattern Analysis and Machine Intelligence 23*, 11 (2001), 1222–1239.

[CSAD04] COHEN-STEINER D., ALLIEZ P., DESBRUN M.: Variational shape approximation. *ACM Transaction on Graphics 23*, 3 (2004), 905–914.

[Elb95] ELBER G.: Model fabrication using surface layout projection. *Computer-Aided Design 27*, 4 (1995), 283–291.

[GSZ11] GREGSON J., SHEFFER A., ZHANG E.: All-hex mesh generation via volumetric polycube deformation. *Computer Graphics Forum 30*, 5 (2011), 1407–1416.

[JKS05] JULIUS D., KRAEVOY V., SHEFFER A.: D-charts: Quasi-developable segmentation. *Computer Graphics Forum 24*, 3 (2005), 581–590.

[KZ04] KOLMOGOROV V., ZABIN R.: What energy functions can be minimized via graph cuts? *Pattern Analysis and Machine Intelligence 26*, 2 (2004), 147–159.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: a high resolution 3d surface construction algorithm. *ACM Computer Graphics 21*, 4 (1987), 163–169.

[LVS*13] LIVESU M., VINING N., SHEFFER A., GREGSON J., SCATENI R.: Polycut: monotone graph-cuts for polycube base-complex construction. *ACM Transactions on Graphics 32*, 6 (2013), 171.

[MS04] MITANI J., SUZUKI H.: Making papercraft toys from meshes using strip-based approximate unfolding. *ACM Transaction on Graphics 23*, 3 (2004), 259–263.

[MSS94] MONTANI C., SCATENI R., SCOPIGNO R.: Discretizing marching cubes. *Proceedings of the conference on Visualization* (1994), 281–287.

[Sor06] SORKINE O.: Laplacian mesh processing. *Tel Aviv University* (2006).

[STL06] SHATZ I., TAL A., LEIFMAN G.: Paper craft models from meshes. *The Visual Computer 22*, 9-11 (2006), 825–834.

[ZLAK14] ZIMMER H., LAFARGE F., ALLIEZ P., KOBBELT L.: Zometool shape approximation. *Graphical Models* (2014).