FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Migration of Surface Curve to Most Concave Isoline

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Maximilian Mayrhauser
Matrikelnummer 0926916

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Mohamed Radwan

Wien, 13. Dezember 2016

Maximilian Mayrhauser	Mohamed Radwan

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Migration of Surface Curve to Most Concave Isoline

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Maximilian Mayrhauser
Registration Number 0926916

to the Faculty of Informatics
at the TU Wien
Advisor: Mohamed Radwan

Vienna, 13th December, 2016

Maximilian Mayrhauser	Mohamed Radwan

Erklärung zur Verfassung der Arbeit

Maximilian Mayrhauser
Schindlergasse 36/3, 1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 13. Dezember 2016

Maximilian Mayrhauser

Acknowledgements

I want to thank Mohamed Radwan for being very helpful whenever there were any issues.

Kurzfassung

Ich präsentiere einen Algorithmus, um eine Kurve auf einer dreidimensionalen Oberfläche zur Isolinie mit höchstmöglicher Konkavität in ihrer Nähe zu verschieben. Als Ansatz erstelle ich aus der Umgebung der ursprünglichen Kurve einen Graph, für welchen anschließend der kürzeste Pfad gesucht wird. Um dadurch eine konkave Linie zu erhalten, wird die Gewichtung der Kanten anhand derer Krümmungsverhalten angepasst. Essentiell handelt es sich um einen Segmentierungsalgorithmus aus einer anderen Perspektive.

Die resultierenden Segmentierungslinien sind von ähnlicher Güte, wie jene von existierenden Segmentierungsalgorithmen. Wegen einer Laufzeit von unter einer Sekunde ist die von mir präsentierte Variante aber geeignet, um eine Verbesserung des User Inputs in Echtzeit durchzuführen.

Abstract

In this paper, I present a solution for migrating a curve on a three dimensional surface to the most concave isoline in its vicinity. Essentially, this problem statement tackles mesh segmentation from a different angle. The search for a suitable segmentation boundary is reduced to a shortest path problem.

First, a graph is built using the mesh's vertices and edges near the input curve. Then, the shortest path is found using the Dijkstra algorithm, whereas a modified weighting scheme that makes the passing through of concave edges cheaper, among other factors, results in a path suitable as segmentation boundary.

The final algorithm provides segmentation boundaries of a quality similar to existing segmentation algorithms. The runtime generally lies below a second, thus making it viable for on the go optimization of the user's input.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**

2 State of the Art **3**

3 Methodology **5**
3.1 Edge Curvature 5
3.2 Dihedral Angle 6
3.3 Dijkstra 6
3.4 Princeton Benchmark 6

4 Algorithm **9**
4.1 The Graph 9
4.2 Optimal Path 11

5 Implementation **13**
5.1 Building the Initial Lasso 13
5.2 Building the Region 14
5.3 Creating the Split 15
5.4 Edge Weights 16
5.5 Dijkstra and Global Minimum 18

6 Results **19**
6.1 Benchmark result for Cut Discrepancy 20
6.2 Performance 22

7 Discussion and Future Work **25**

APPENDICES **26**

A Screenshots of segmentations evaluated with Benchmark **27**

Bibliography **33**

CHAPTER 1

Introduction

In graphics applications dealing with 3D meshes, the selection of regions on a surface is an important function in multiple regards. For example it can be useful when painting textures, or when inspecting a model in detail. An existing application I'll be working with lets the user select such a region by drawing a lasso around it by hand, creating a selection by projecting that lasso onto the mesh's surface. This presents a certain issue: while the user has a lot of freedom in the selection process, the selection can be undesirably inaccurate due to manual error.

In this paper I'm proposing an algorithm to help that issue, by taking an edge loop on a mesh - in my case, resulting from the user's lasso - and migrating it to the best fitting edge loop that exists in its vicinity. The fitness of such an edge loop is decided by curvature and concavity, and should result in a boundary closer to what the user had in mind. If a user draws the lasso around a bunnies ear, for example, he likely wants the selection's boundary to be placed exactly at the ear's base. This is achieved by maximizing the edge loop in terms of concavity and curvature, two criteria that are commonly used in mesh segmentation[APP+12].

To implement this idea, I suggest a transformation into a graph problem. Using a shortest path algorithm with a weighting scheme dependent on curvature and concavity, it is possible to find good candidate boundaries.

Since the user's input is modified during runtime, performance is a big concern. The suggested implementation is viable in this regard, generally having a runtime below one second, depending on the number of vertices covered by the initial curve.

Essentially, this is a segmentation problem, approached from a different perspective. Segmentation has been a much researched topic in all kinds of contexts, be it images, audio, and 3D meshes[CGF09][LHMR08]. Many existing mesh segmentation algorithms are designed to be fully or mostly automatic, taking a mesh as an input, and trying to find a good segmentation without additional information, except sometimes for the

number of expected segments. The problem statement handled in this paper differs from that, as the user will give an initial curve as input.

In the following section I will give an overview about the state of the art of mesh segmentation. Following that, I will present my proposed algorithm, first in general terms, and in more detail afterwards. Finally, I will present the results and show an evaluation of my method using a benchmark, comparing mine to the results of other segmentation algorithms.

CHAPTER 2

State of the Art

There have been a number of ventures into mesh segmentation, using a variety of approaches. They differ both in use case as well as methodology. While the concrete input and goal of most existing algorithms differs from the purpose of the problem addressed in this paper, the necessity of finding good segmentation boundaries is the same. I will give an overview of existing segmentation techniques, some of whose concepts are integral to my method.

In „Fast Mesh Segmentation using Random Walks“ Lai et al. applied the random walk method from image segmentation to a three dimensional case[LHMR08]. The algorithm first generates seed faces and assigns to all other faces a likelihood of whether it belongs to that seed. As the number of seeds generated is generally larger than the number of wanted segmentation clusters, some pieces are then merged. The initial likelihood of faces is determined by random walks to seeds. The main goal of this approach is efficiency, as it achieves a runtime of 0.034 seconds per 1000 triangles. In terms of accuracy, it is decent according to the Oxford benchmark, having a Rand Index of 21% (this metric will be explained in Chapter 3). While efficiency is also the goal of my algorithm, the problem is much more localized, making the principle of seed spreading unnecessary.

Another existing approach to the mesh segmentation problem is based on Shape Diameter Function values. This method was explored by Shapira et al. in „Consistent Mesh Partitioning and Skeletonisation using the Shape Diameter Function“[SSCO08]. While this paper does not solely explore segmentation, it is one of two major topics for which they found SDF values to be useful; it is also worth mentioning that this is the algorithm implemented in the CGAL library for segmentation purposes. The approach to segmentation here is the volume of the object, which as its results show is reasonable. One advantage of this as opposed to curvature is that the volume is largely pose invariant, which is advantageous for their second use of extracting skeletons. While the approach is very solid, the computation time of SDF values prevents it from being viable for the case of optimizing user inputs on the fly (The calculation of SDF values for a mesh with

20000 triangles was cited as 6.1 seconds `http://doc.cgal.org/latest/Surface_mesh_segmentation/index.html#title13`.

One method is the principle of learning algorithms. Kalogerakis et al. aim to mark each face of a mesh in question with a label describing its features (such as „arm“, „leg“, etc.) whereas the set of labels is predefined[KHS10]. To assign labels they use an array of values pertaining local geometric features, then minimize an energy term according to a Conditional Random Field model. To determine the parameters for said model, a learning method using an exemplar and a validation set was used.

Benhabiles et al. followed up with another learning approach in „learning boundary edges for 3D-mesh segmentation“[BLVD11]. Whereas the previous work maps labels to the mesh and performs the segmentation from there on out, Benhabiles et al. directly look for boundary edges of the segmentation. For a large collection of manual segmentations, edges are processed and classified as to whether they are boundary edges or not. Then, the edges of the mesh in question are measured against the precomputed data. With potential boundary edge candidates determined this way, concrete boundaries are found with region thinning and snake movement. The results from learning methods provide very accurate segmentations according to the Princeton benchmark, having around 9% error as opposed to the 15% record held in place before learning algorithms were used. While accurate however, the run time for these methods is again too high to be viable in my application, ranging between 10 and 40 minutes for the learning step, and still a whole minute for the segmentation step.

An earlier approach consists of fitting primitives, as explored by Attene et al.[AFS06]. Another tries to fit quadric surfaces, minimizing an energy function to gain a segmentation as solution [YWLY12].

Finally, a popular approach is focusing on curvature. In particular, one recent approach bases its segmentation on concavity information exclusively. Au et al. in „Mesh Segmentation with Concavity-aware Fields“ use harmonic fields with a novel weighting by concavity[AZC+11]. By solving a Laplacian system they create a scalar field with high sensitivity to concavity. Multiple candidate boundaries are created by sampling lines from said field, and a greedy algorithm is then used to pick the best fitting boundaries. The main strength of this method is, similar to the random walks method, efficiency while maintaining a reasonable quality. They report a runtime of 10 seconds for computing the segmentation fields, and 2 seconds for the other steps, on a mesh of 100K triangles, which is by far the best run time for any of the methods explored.

CHAPTER 3

Methodology

My approach reduces the segmentation problem to a shortest path finding problem. For this purpose, I construct a graph out of the mesh's vertices and edges near the user's initially input curve. The reduction to a graph problem makes sense as long as the user input gives a rough estimate about shape and position of the cut, although there is ample room for error (especially regarding the shape). General principles behind segmentation problems are still relevant, though. For one, the shortest path problem has similarities to the use of SDF values, where the segmentation is determined by the mesh's diameter at different position. Likewise, my approach tries to minimize the diameter at different points, although the diameter will be weighted with additional parameters apart from length. Furthermore, the knowledge that segmentation boundaries are often found along concave creases is valuable. In particular, two measurements of curvature and concavity will be needed for the weighting process, which will be explained in the following. Afterwards, I'll discuss the choice of shortest path algorithm and evaluation scheme.

3.1 Edge Curvature

For approximating curvature values of edges, I use the following formula:

$$C_e = \frac{(v_a - v_b) * (n_a - n_b)}{(v_a - v_b)^2}$$

whereas v_a and v_b are the position values of the two points forming the edge, and n_a and n_b are the according vertex normals. The expression returns a positive value in case the curvature at the edge is convex, otherwise it is negative. The higher the absolute value of the expression, the higher the curvature at that edge.

3.2 Dihedral Angle

To detect concave seams I use the Dihedral angle between the two adjacent faces for all edges. The angle between its two adjacent faces is calculated for each edge. Implementationwise, the dot product of the faces normal vectors is calculated and brought into a range between 0 and 2, where 0 means the normals are facing in the same direction. Then, the angle of the normal vectors to PQ (P and Q are the vertices not used by the edge) is calculated to determine whether the surface at the edge in question is convex or concave. In case of concavity, the negative value of the angle is stored.

3.3 Dijkstra

The shortest path problem has been thoroughly researched, so by now there exist many variations in both problem statement as well as solution. Since different or enhanced problem statements did not make sense in my context, however, I decided to use the plain version of the Dijkstra algorithm. To accommodate my use case I use a weighting scheme dependent on multiple parameters, making the traversal of concave edges, edges with high curvature, as well as edges with correct direction cheaper. Using these weights in addition to edge length, it is possible to find good candidates for segmentation boundaries.

3.4 Princeton Benchmark

In order to evaluate the segmentations I utilize the Mesh Segmentation Benchmark of the Princeton University[CGF09]. Before, mesh segmentation algorithms had no clear way to evaluate their quality, usually resorting to a presentation of screenshots with faces colored respective to the segmentation. How good a segmentation was, was left to the readers for personal interpretation. In an attempt to provide a more unified evaluation system, the benchmark was developed in 2008. From a dataset of 380 meshes divided into 19 categories, manual segmentations were created by multiple users, averaging 11 segmentations per mesh. This data is treated as ground truth for evaluating new results. In order to compare segmentations in a meaningful way, four metrics inspired by image segmentation, were used.

- Cut Discrepancy measures the distance from the analyzed segmentation's boundary line to the boundary of the ground truth. Basically, it compares the segmentation's borders.
- Hamming Distance measures the overall region-based difference between segmentation results. Its value is given as Directional Hamming Distance, which pairs segments from the two given segmentations and describes the sum of the mapped pairs difference. Furthermore, two error terms are given. What makes the Hamming Distance unique is its sensitivity to correct or incorrect mapping, which can make the value either very meaningful, or result in noise.

- Rand Index gives the likelihood of whether adjacent faces are in the same segment or not. This is a preferred method of measurement, as it can check for overlaps of segments without needing a segment mapping, thus removing a large error factor from the previous metric.
- Consistency Error, the final metric, compares the hierarchical structures of segmentations. This value is considered volatile when meshes are under- or oversegmented, but can be useful when handling segmentations with hierarchical differences.

Generally, the benchmark is used to evaluate fully or mostly automatic segmentations (at most requiring input parameters like the expected number of resulting segments). This is of course a different case than what I am working with. For this reason, I will make a few modifications to the conduction of the evaluation and set clear conditions under which the experiment will be performed.

CHAPTER 4

Algorithm

The algorithm can be divided into two steps. First, a graph is built using the vertices and edges near the user input. In the second step the Dijkstra algorithm is performed, using a weighting scheme based on various edge properties. As the graph is built out of the mesh's existing edges, the resulting boundary will also be limited to existing edges.

4.1 The Graph

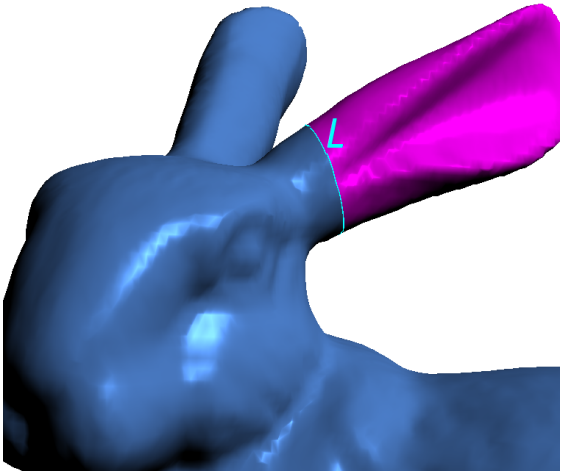


Figure 4.1: Lasso L from user input. The user's selected region is colored in magenta.

Like mentioned, initially the user selects a region of a mesh's surface by projecting a hand drawn lasso. The result of this is an edge loop on the mesh, which splits the surface into selected and unselected regions. Note that the projected edge loop is a prerequisite for my proposed algorithm, how the projection was acquired is not a concern. From here

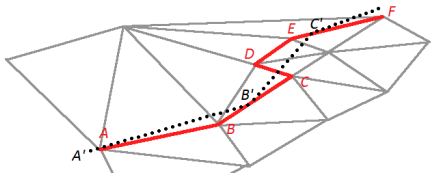


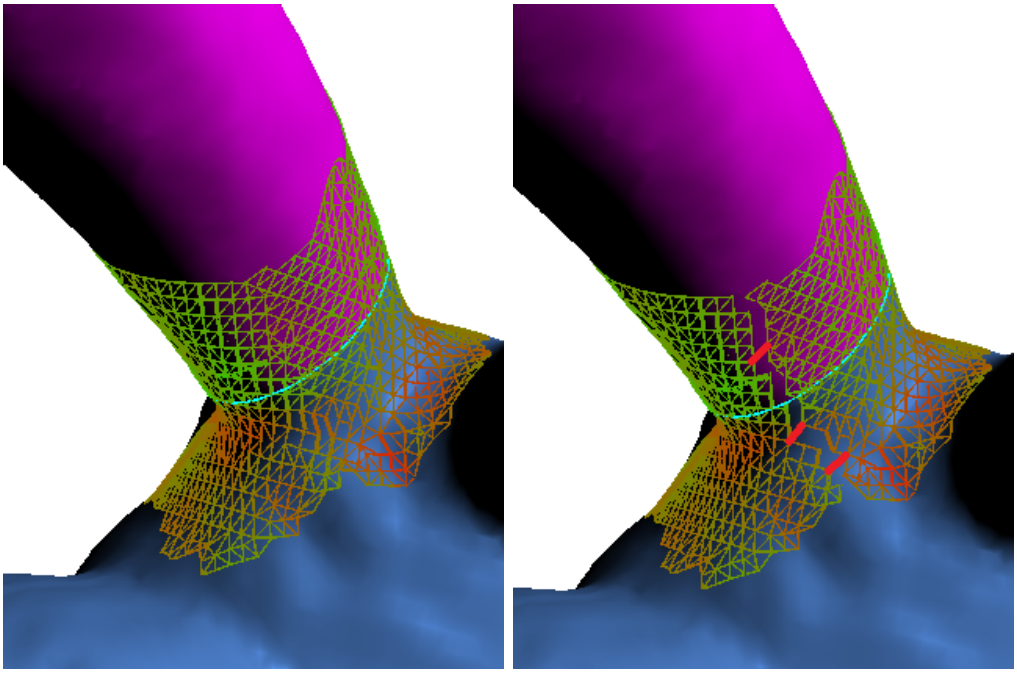
Figure 4.2: Original Lasso in red, L' in black

on out, the term Lasso will be used when referring to the mesh's initially selected edge loop. For my algorithm only the Lasso is relevant; which side of it is selected however makes no difference.

The Lasso will be used as a base for building the graph. Depending on the curve drawn on the mesh, the Lasso's vertices could possibly be distributed unevenly. To reduce potential error caused by this, I create another edge line L', based on the Lasso, by inserting a vertex at every distance d. From this new line, I calculate the approximate direction at each vertex (Figure 4.2).

The graph will be constructed out of the Lasso's nearby vertices. I gather a region of vertices near the lasso, by starting with the lasso's vertices, then propagating the region by adjacent vertices for several iterations. Whenever Region is mentioned from now on, it will be referring to the vertices and edges collected in this step.

Every region vertex will become one node in the graph, and all nodes whose vertices share an edge will be connected. Before the graph can be used, start and end nodes need to be defined. For this purpose I create the "Split", by cutting through the region at an arbitrary point in the Lasso, resulting in multiple pairs of start and end points (see Figure 4.3). At this point, the graph necessary for finding a segmentation path is fully constructed.



(a) Without creating a cut, the region forms a loop of faces around the bunny ear.

(b) Here the Split was applied, resulting in well defined start and end points. Three examples for Split pairs are marked as thick red lines.

Figure 4.3: Region of the Lasso

4.2 Optimal Path

The Dijkstra algorithm will now be used for every pair of nodes in the Split. To receive segmentation boundaries instead of the geometrically shortest paths, a distinct weighting scheme is used. The weight of each edge is given as the sum:

$$Length + Direction + DihedralAngle + Curvature$$

The meaning of these weighting parameters is described below.

- ***Length*** is the edge's geometric length.
- ***Direction*** expresses the deviation of an edge's direction compared to the lasso segment it's mapped to. The higher the deviation, the higher this value, thus making the traversal of edges deviating heavily from the user's lasso expensive.
- The ***Dihedral Angle*** parameter makes passing through edges cheaper when the angle between its adjacent faces is low.

- The ***Curvature*** term favors edges with highly concave curvature values.

The first problem was determining which parameters are useful for finding segmentation boundaries.

Geometric distance is important as it provides a basis for a graph's fast traversal. Since a shorter distance is not necessarily characteristic of segmentation boundaries, ignoring the length term was considered at first. When experimentally discarding it, however, the suggested isolines tended to be more jagged and winding, so it remains in the calculation as fundament for finding a stable line.

A major modification of weighting for segmentation boundaries is the use of curvature and dihedral angles, both of which are popular surface metrics to approach mesh segmentation [CGF09]. The detection of concave seams takes into account equal parts of dihedral angle as well as general curvature in the vicinity. Both of these terms are useful, as the dihedral angle favors individual concave edges, while curvature benefits edges from generally concave areas. The difference is that *Dihedral* term only considers the angle of an edge's directly adjacent faces, while the *Curvature* term considers curvature approximations from all adjacent edges.

The parameter of directional deviation was introduced to help keep the shape of the user's original input. Edges with the same orientation as the corresponding lasso segment have less weight than edges orthogonal to it. The introduction of this term helped with some cases where the resulting boundary shape was too different from the user input.

Finding the right balance between the regular shortest path and boundary conditions was another concern. The values were normalized in such a way that they range between 0 and 1, except for length values which have a median of 1. This normalization keeps the parameter's relative impact equal. While using multipliers for individual parameters was considered and tested, it turned out that the initial 1:1:1:1 distribution gave the most consistent results.

At this point, the individual paths calculated for each Split pair are good, but sometimes the shortest global path is not a good fit as segmentation boundary. To accommodate this without reducing the quality of individual paths, the mean concavity of each path is factored in as $PathLength = PathLength * PathConcavity$, and then the minimal length path is accepted as global optimum (global referring to the graph in question).

CHAPTER 5

Implementation

My concrete implementation was done in C++, using the glm library for geometric operations. The explanation in this section however will not be language specific.

Algorithm 5.1: Top level function calls

```
1 Get Lasso;
2 Build Region;
3 Create Split;
4 Calculate Weights;
5 foreach S in Split do
6 |   CandidatePaths ← Dijkstra(S);
7 end
8 return Minimum(CandidatePaths);
```

5.1 Building the Initial Lasso

In my case the curve is initially given by a collection of subsequent vertices that form the initial boundary. Since the given edges will vary in length, I made a small modification to the lasso I work with in order to reduce error, by recreating a lasso in the same shape but with equal edge length.

I iterate through the Lasso's vertices, accumulating distance. Each time the distance surpasses the segment length wanted for the modified Lasso, a new vector is created and inserted into the new data structure.

From this new lasso, the approximate direction is stored at each point. The estimate is calculated as the unit vector of n – p, where n is the geometric mean of the 3 following lasso points, p the one from previous points.

5.2 Building the Region

To build the graph, retrieving the mesh information near the Lasso is necessary. The general idea is simple, as we need the vertices and edges up to a certain area. For orientation purposes later, though, I found it important to order those vertices instead of inserting them in an arbitrary order.

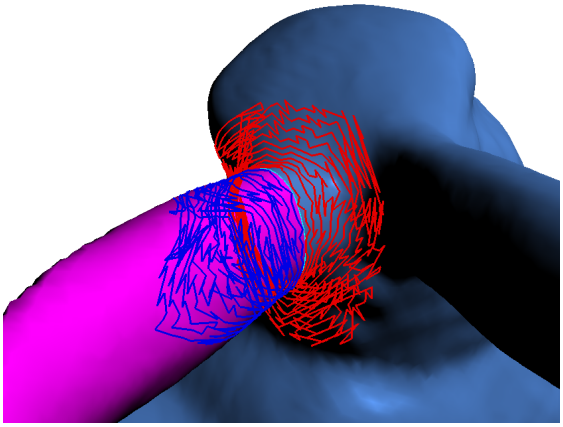


Figure 5.1: Ring-wise creation of the region. Red and blue rings mark whether they are stored as Inner or Outer Ring.

First, I retrieve the adjacent vertices from the original lasso (not the equidistant lasso, the points of which are technically not connected to the mesh). Depending on which side of the initial boundary they lie on, I insert them into either the vertex collection Inner Ring or Outer Ring.

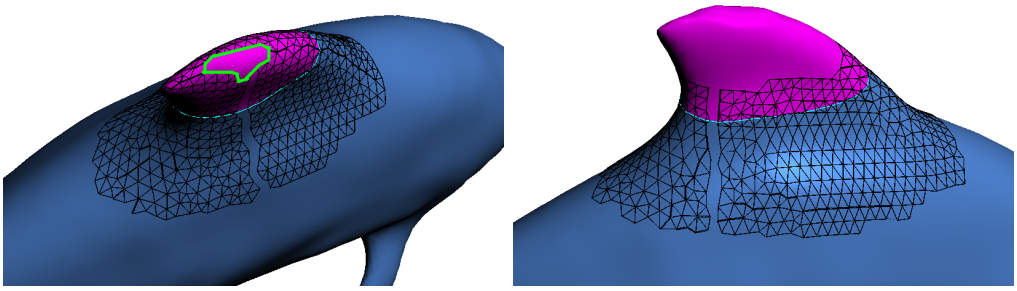
Then, for both Inner and Outer Ring, I again fetch all the direct neighbors that were not previously inserted. This process is repeated for a number of iterations, until the area covered near the lasso is big enough to cover potential boundary lines. In my implementation 20 iterations are usually sufficient, 10 of which are ultimately used (see next paragraph), though this could vary depending on mesh size and error of the initial Lasso.

Generally speaking, the collection of vertices with adjacency to a certain degree is trivial. However, there are a few reasons for splitting them into the hierarchy presented here.

- It is necessary to determine the outermost vertices from the region with which the graph will be built. The outermost rings on both sides will not be used in the graph, which will help orienting the Split later on.
- Sometimes, the initial lasso encircles a closure of the surface, and the region fetched around the lasso is big enough to cover or almost cover said closure. In this case, the shortest path will consequently take the least edges possible right in the middle

of the region around the closure. To detect and prevent this case, I add more inner and outer circles than necessary just for the purpose of fetching a sufficiently large region. If there is indeed a closure, the iteration will face a situation where no more adjacent vertices can be added, as they are all already contained in the region. After the ring wise addition of regional vertices is complete, I discard half of the rings, specifically the outer half on both sides of the lasso. This way, vertices that lie close to peaks of the surface cannot be added to the actual graph. A sketch of this can be viewed below.

- There is one more case that can lead to wrong results, which might occur if there is another non-target protrusion near our wanted part of the mesh. If the region is grown enough to fully cover that other protrusion, the shortest path found later on could potentially cover that protrusion instead of the part actually covered by the lasso. While this case is unlikely due to directional weights, it can happen. To prevent it, it's possible to detect if an unexpected connection is found while expanding the region, and severe the connection at those edges.



(a) Without discarding edges near a closure, a minimum path (as outlined in green) might be found directly at the tip.

(b) Here, the vertices edges near the tip of the fin were discarded, to avoid mistakes of this nature.

Figure 5.2: Curve near surface closures

5.3 Creating the Split

At this point, using all edges and vertices to build the graph would result in a loop of faces, as seen before in Figure 4.3. This prevents the network from being useful for a shortest path problem, so clear start- and endpoints need to be defined. For this purpose, I split the loop at an arbitrary position, marking the vertices on one side of the split as start nodes, those on the other side as end nodes. So, the Split is a collection of vertex pairs that signify start nodes and end nodes.

While there are multiple ways to create that Split, I applied the Dijkstra algorithm with the outermost vertex rings as start and end nodes. This way, the Split is created at the

position where the loop is the narrowest, resulting in the fewest possible amount of vertex pairs as start- and end nodes.

Now, all the vertices and edges as well as connectivity information necessary for the graphs creation has been collected. Before the edge- and vertex values for weighting in the Dijkstra algorithm can be determined, each vertex needs to be mapped to one lasso segment. This mapping is necessary to determine the Directional Deviation weighting parameter that looks to keep the general shape of the user's initial Lasso.

To perform the mapping, I calculated the geometric distance from each vertex to each (modified) lasso segment, and assign each vertex to the segment it is closest to. There are some other methods that might yield more consistent results. One alternative method would be the use of geodesic distance, in case the mesh topology does not allow the geometric distance to be a meaningful measurement. Another approach would be to map each vertex to the lasso segment closest in terms of adjacency. However, in my testing I found the use of geometric distance to be sufficiently accurate.

5.4 Edge Weights

The Weight for each edge individually will be

$$Length + Direction + DihedralAngle + Curvature$$

where the exact calculation of each term is shown below.

5.4.1 Length

The first weighting term is an edge's geometric length. For the weighting process a normalized value is used – the „normalized length“ is calculated by multiplying the edge length with $\frac{1}{MedianLength}$. This way, the edge length will predictably lie in a small range of 1, regardless of a mesh's scale.

5.4.2 Directional Deviation

The next parameter of edges is the directional deviation from the according lasso segment. As mentioned before, the Lasso is recreated as equidistant edge loop Lasso', and each vertex is mapped to one vertex of that Lasso'. Now, for each edge I consider the Lasso' segments its vertices are mapped to, and normalize the sum of their directions. The resulting vector will be called the expected direction. Then, I calculate the dot product of the edge direction and the expected direction, which gives us a value between -1 and 1 depending on the angle between these vectors. Since it is not certain in which direction an edge will be traversed, the dot product's absolute value will be used. Therefore the value 0 stands for the maximum deviation of 90 degrees, while 1 means the edge has the same orientation as the according lasso segment. Since the passing through of edges with

the expected direction should be cheaper, however, the value will be inversed. Therefore, edges of the expected orientation are more likely to be chosen, cheaper to traverse in the algorithm, than edges that are close to orthogonal to the accompanying lasso segment.

Since the problem statement suggests that the initial lasso is not a perfect representation of the wanted shape, there is decent room for error here. To lessen the impact of such imperfections, one improvement I found for the directional deviation was using the squared value for weighting instead. Thus, small deviations from the expected direction have no tremendous impact, while stronger deviations retain their significance.

5.4.3 Dihedral Angle

The dihedral angle between two adjacent faces is calculated for each edge as described in the Methodology section, so that lower angles between faces result in a smaller value for the dihedral angle. To even out the impact of the Dihedral Angle compared to other parameters, a normalized value is used. The final value will range from 0 to 1, 0 being the lowest Dihedral Angle of all edges in the graph, 1 being the highest, with linear scaling in between.

5.4.4 Curvature

The curvature estimation of a single edge is given using the aforementioned term:

$$C_e = \frac{(v_a - v_b) * (n_a - n_b)}{(v_a - v_b)^2}$$

By extension, vertex curvatures are estimated as the geometric mean of curvature values from all edges adjacent to the vertex in question $C_v = \frac{1}{edgenum} * \sum_{n=1}^{edgenum} C_e$. Note that at the outer- and innermost rings, some edges will be missing; the curvature values for those edges were calculated additionally in my implementation, though the edges were not stored. While these edges could probably be ignored, there might be some corner cases where the vertex curvature estimation has a slight error when ignoring the outerbound edges.

Finally, one more Curvature estimation is done for edges, by forming the mean of its two vertex curvatures. This is the value which will actually be used in the algorithm's weighting processes.

To give curvature a similar impact in the weighting processes regardless of the region's topology, I normalize the curvature terms in such a way that vertex and edge with minimum curvature will have a value of 0, and the maximum curvature will be 1, with the values in between again being scaled linearly.

5.5 Dijkstra and Global Minimum

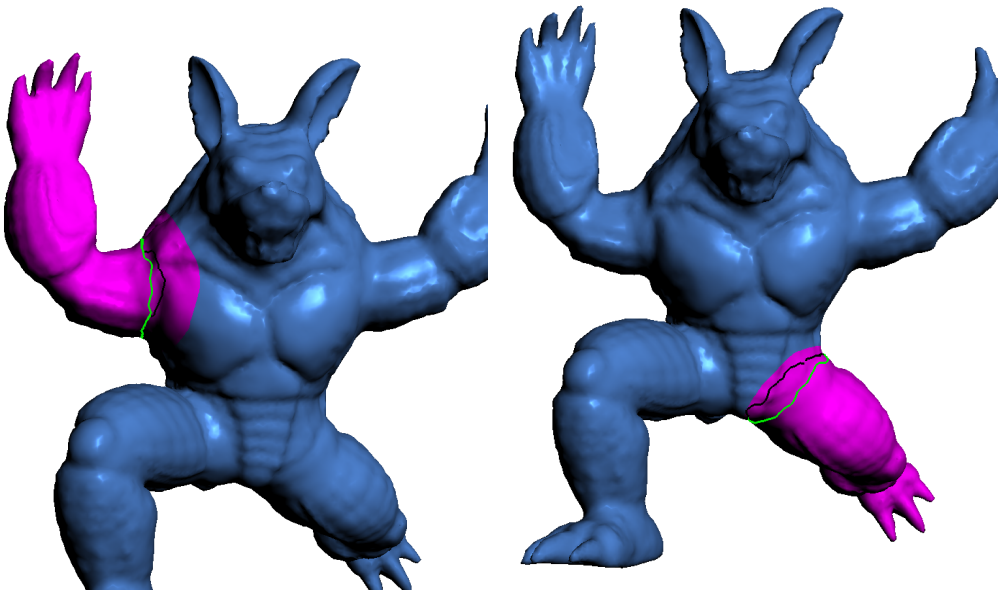
With the weight values gathered, the Dijkstra algorithm can be applied to find the shortest path in the created graph. For each pair of vertices from the Split, the algorithm is used once, with one of the vertices set as start node, the other as end node.

At first, paths for all Split pairs are stored. While using the shortest path out of those already provides good results in many cases, sometimes the shortest path would lie in a region that was not suitable for segmentation, most commonly choosing a surface traversal with little curvature when concave regions would be nearby. Changing the weighting to place more focus on curvature however would only produce unnatural boundaries, while not necessarily helping the actual problem.

To solve this issue, I calculate the geometric mean of each path's vertex curvatures, and multiply the path's total length by that value. Then the path with minimum length becomes the suggested boundary line. This way, boundaries at concave regions are preferred without unnecessarily interfering with the path's weighting.

CHAPTER 6

Results



(a) Armadillo right arm (b) Armadillo left leg

Figure 6.1: The border to the purple area shows the initial lasso, the green line shows the algorithm's suggested boundary. Occluded parts of boundary line shown in black.

For segmentation evaluations using the Princeton benchmark, usually the entire catalog of 380 meshes is used. However, since the presented algorithm is not fully automatic, I ran the evaluation using a subset of the given library. From 19 categories each containing 20 meshes, I use the first mesh of 7 arbitrarily chosen categories. Since I use only a fraction of the original dataset, I also ran the evaluation for the given segmentation algorithms delivered with the benchmark, in case their results differ from the original

test. In fact, the results differ a notable amount, although consistently so, as the Cut Discrepancy is lower by approximately 30%. The results are still relevant, however when looking at the numerical results for each metric, this should be kept in mind. The reason for this discrepancy could be the choice of meshes evaluated in this test, as benchmark results differ to some degree depending on categories, not all of which were covered.

Aside from the smaller sample size, there are other implications from using the benchmark with only a semi automatic segmentation method. First, the number and location of segments will be determined by the user. Secondly, results can vary depending on user input. For example, it would theoretically be possible to a perfect segmentation boundary by hand, as initial curve, which might be even better than estimations given by the algorithm based on imprecise user input (note that the algorithm is designed for the case of improving such an initial, imprecise curve).

As such, the presented values are only evaluations of the algorithm's boundary improvement based on my user input done when conducting the experiment. To have a look at the manual selections used in the evaluation, screenshots of all evaluated segmentations are included in the appendix, contrasting the initial curve to the algorithm's improved segmentation boundary (shown as line segment colored in green for visible, black for occluded parts).

As an enhancement to the algorithm, I give the user the option to manually improve the initial suggested boundary. However, in the creation of segmentations evaluated with the benchmark, this functionality was not used, so all tested segmentations are done with the initially proposed boundary.

From the tables shown below, the first column lists the result of the ground truth, the following columns show results for different existing segmentation algorithms. Before viewing the result tables, the benchmark value's meanings should be reiterated: All four metrics use an error based function. As such, the lower the value, the lesser the deviation from the ground truth, thus the better the segmentation's quality.

I performed two evaluations in particular: One test, listed under „SmartCuts“, uses segmentations optimized with my proposed algorithm. The other test, „ManualCuts“ in the table, gives the results for the initial hand drawn segmentation.

6.1 Benchmark result for Cut Discrepancy

As already mentioned in the beginning, Cut Discrepancy is the only metric calculated using the segmentation boundaries, as opposed to segmentation areas. Since boundaries are this algorithm's main concern, it is the most interesting metric when evaluating segmentations in our context.

As can be seen in Table 6.1, the tests for CD yield good results, being below both average as well as median compared to the other algorithms presented with the benchmark.

Table 6.1: Cut Discrepancy

	Benchmark CD	RandCuts CD	ShapeDiam CD	SmartCuts CD	ManualCuts CD
Average	0,124438	0,20353575	0,168965	0,189383	0,316547
Human	0,219124	0,23184475	0,325744	0,360306	0,321401
Airplane	0,087436	0,2104495	0,096345	0,171953	0,418087
Ant	0,062818	0,09123875	0,068484	0,058762	0,294756
Chair	0,127186	0,24598525	0,420634	0,291096	0,413275
Octopus	0,043385	0,05845425	0,03524	0,042123	0,301303
Table	0,207517	0,41456925	0,11677	0,130289	0,167428
Armadillo	0,123598	0,17220775	0,119538	0,27115	0,299576

Table 6.2: Cut Discrepancy (2)

	NormCuts CD	CoreExtra CD	RandWalks CD	FitPrim CD	KMeans CD
Average	0,19106775	0,257101	0,270405	0,242969	0,294707
Human	0,25471025	0,341739	0,2294705	0,226289	0,29739325
Airplane	0,401257	0,224828	0,4155625	0,44578225	0,4325815
Ant	0,1256845	0,139458	0,248071	0,22871175	0,31091675
Chair	0,127929	0,211716	0,52951675	0,24780075	0,3289765
Octopus	0,08193275	0	0,08393325	0,222774	0,17331375
Table	0,1881395	0,360302	0,200782	0,1790455	0,32033025
Armadillo	0,1578225	0,26456	0,18549925	0,15037975	0,1994395

Also, the improved cut shows a drastic improvement to the initial user, although again the degree of this discrepancy is heavily dependent on the initial selection. There is a notable fluctuation between different models, particularly, for one category (Human) the boundary suggested by my algorithm performs worse than the initial selection. As such, there are cases where my algorithm's proposed boundaries do not match the segmentation boundary sought for. For these cases, the aforementioned option of a manual correction was implemented, while in most cases it should not be necessary.

6.1.1 Benchmark results for other metrics

The results for the Rand Index metric (Table 6.3) show results of a similar quality, my proposed method performing only marginally worse than the Shape Diameter Segmentation and Normalized Cuts method, while outperforming the others.

As both Hamming Distance as well as Consistency Error results span multiple values, I only include the results for the Benchmark and Random Cuts method for comparison(Table 6.6, Table 6.9). The results of the Hamming Distance are by average similar to those of the random Cuts method. As for Consistency Error, my algorithm's results are closer to the

benchmark than to other segmentation algorithms' evaluation.

While not as meaningful as the Cut Discrepancy, the relatively consistent results across all metrics speaks for the legitimacy of the modified benchmark experiment.

Table 6.3: Rand Index

	Benchmark RI	RandCuts RI	ShapeDiam RI	SmartCuts RI	ManualCuts RI
Average	0,074293	0,16266325	0,102474	0,125649	0,192103
Human	0,072779	0,11055725	0,239578	0,106401	0,121777
Airplane	0,159938	0,184311	0,13106	0,148654	0,245704
Ant	0,010543	0,03042075	0,010922	0,01172	0,119947
Chair	0,054288	0,2009765	0,161955	0,377411	0,446014
Octopus	0,021728	0,051923	0,016268	0,018131	0,128168
Table	0,132301	0,4293695	0,076868	0,075675	0,108159
Armadillo	0,068472	0,13108575	0,080664	0,141548	0,174951

Table 6.4: Rand Index (2)

	NormCuts RI	CoreExtra RI	RandWalks RI	FitPrim RI	KMeans RI
Average	0,11405075	0,16615	0,13870775	0,136809	0,15720625
Human	0,15150625	0,187127	0,092733	0,087526	0,10950275
Airplane	0,23638425	0,169961	0,26315525	0,2398675	0,256126
Ant	0,05395825	0,033561	0,08331875	0,0761135	0,122126
Chair	0,06411475	0,123671	0,21741225	0,21530725	0,16874575
Octopus	0,058446	0	0,052966	0,10729075	0,0807205
Table	0,12776375	0,337029	0,14534225	0,13909025	0,23801975
Armadillo	0,10618075	0,145549	0,11602475	0,09246775	0,12520275

6.2 Performance

The algorithm's performance depends on the mesh's resolution and the selection size, as well as the concrete implementation of 3D data structures.

The path finding itself takes 120 milliseconds, with no correlation to either mesh nor lasso size. In my implementation the biggest part of the runtime lies in the creation of the graph, which takes 250 milliseconds for a mesh with 10 000 faces, up to one second for a mesh with 50 000 faces.

The runtime measurements were taken with an i5-4690K 3.5GHz quad core CPU.

Table 6.5: Hamming Distance (Benchmark and Random Cuts)

	Benchmark			RandCuts		
	Hamming	Hamming-Rm	Hamming-Rf	Hamming	Hamming-Rm	Hamming-Rf
Average	0,099824	0,099824	0,099824	0,172754	0,19544075	0,15006725
Human	0,173668	0,173668	0,173668	0,1805075	0,15581225	0,20520325
Airplane	0,153289	0,153289	0,153289	0,20265075	0,2745065	0,130795
Ant	0,046834	0,046834	0,046834	0,06152075	0,03274775	0,09029375
Chair	0,057844	0,057844	0,057844	0,215945	0,27813375	0,153757
Octopus	0,030996	0,030996	0,030996	0,05086125	0,02908625	0,072636
Table	0,084085	0,084085	0,084085	0,28410275	0,431067	0,13713775
Armadillo	0,152049	0,152049	0,152049	0,21369075	0,16673225	0,26064825

Table 6.6: Hamming Distance

	SmartCuts			ManualCuts		
	Hamming	Hamming-Rm	Hamming-Rf	Hamming	Hamming-Rm	Hamming-Rf
Average	0,133981	0,051782	0,216179	0,232705	0,155196	0,310214
Human	0,210379	0,07705	0,343708	0,33398	0,213769	0,454192
Airplane	0,166893	0,174785	0,159001	0,279429	0,292363	0,266494
Ant	0,04168	0,025402	0,057957	0,228856	0,212704	0,24501
Chair	0,188285	0,014646	0,361923	0,22699	0,053417	0,400562
Octopus	0,025176	0,019116	0,031237	0,142878	0,135714	0,150041
Table	0,045633	0,007459	0,083806	0,076005	0,03977	0,11224
Armadillo	0,259818	0,044016	0,475619	0,340795	0,138634	0,542956

Table 6.7: Consistency Error (Benchmark)

	Benchmark			
	GCEf	LCEf	GCE	LCE
Average	0,060233	0,041917	0,060374	0,041488
Human	0,10235	0,069394	0,098422	0,066028
Airplane	0,112151	0,072698	0,113119	0,072449
Ant	0,031355	0,023788	0,032408	0,024394
Chair	0,039009	0,024384	0,040576	0,024556
Octopus	0,032166	0,023805	0,033826	0,024287
Table	0,021285	0,015952	0,021796	0,015887
Armadillo	0,083312	0,063398	0,082473	0,062818

Table 6.8: Consistency Error (Random Cuts)

	RandCuts			
	GCEf	LCEf	GCE	LCE
Average	0,1513725	0,08983175	0,14937975	0,086589
Human	0,1543345	0,0786755	0,1490495	0,072978
Airplane	0,18082875	0,10152525	0,18033675	0,09926975
Ant	0,05426275	0,03777075	0,0558	0,0382525
Chair	0,234155	0,12035725	0,22935175	0,11173575
Octopus	0,03956525	0,0281285	0,04105725	0,028921
Table	0,20634	0,12517875	0,201539	0,11863625
Armadillo	0,1901225	0,13718525	0,18852325	0,13632925

Table 6.9: Consistency Error

	SmartCuts			
	GCEf	LCEf	GCE	LCE
Average	0,065279	0,046427	0,066151	0,04583
Human	0,111216	0,08772	0,112301	0,085969
Airplane	0,150608	0,085274	0,150837	0,083063
Ant	0,042335	0,028029	0,044011	0,028777
Chair	0,025405	0,014888	0,027202	0,014737
Octopus	0,031416	0,023225	0,033294	0,023676
Table	0,013361	0,009118	0,014038	0,009141
Armadillo	0,082613	0,076737	0,081374	0,075447

Table 6.10: Consistency Error (Manual input)

	ManualCuts			
	GCEf	LCEf	GCE	LCE
Average	0,192521	0,128652	0,19594	0,12614
Human	0,220414	0,193017	0,226719	0,188875
Airplane	0,301101	0,183242	0,302171	0,178487
Ant	0,275499	0,162233	0,276513	0,158105
Chair	0,074646	0,038356	0,07759	0,036543
Octopus	0,208952	0,104476	0,215795	0,101538
Table	0,057409	0,034987	0,063988	0,03475
Armadillo	0,209627	0,184251	0,208802	0,184683

CHAPTER 7

Discussion and Future Work

My presented approach to finding a segmentation boundary uses many known concepts, but as far as I am aware it is the first one to be based on a shortest path problem. Looking at test results, it can clearly compete with other existing segmentation algorithms. So, this new approach certainly shows promise, and there might still be room for improvement.

There is no certainty of whether the currently used weight calculation is optimal. I tuned the parameters continuously, and from all the configurations tested, the one suggested in this paper was the best both in terms of my judgment as well as benchmark results. However, just as the current formula is the result of multiple test iterations, it seems probable that other, not yet used weighting schemes yield even better results.

Another potential modification would be the inclusion of a smoothing function for the final boundary line.

A thought that might be worth exploring would be a different algorithm for the calculation of the shortest path. The Dijkstra algorithm's weights are static, once nodes are visited, they are considered closed. However, using dynamic weights that change depending on the previous path, might result in smoother boundaries. The issue I imagine with this is runtime. Using the Dijkstra algorithm fits the problem nicely, as speed is a high priority for our application.

The runtime was one of this algorithm's main concerns, and delivers acceptable results. While still reasonable, the runtime for large meshes could potentially be bothersome. It might be interesting to optimize the region generation process, which makes up the biggest portion of the run time for larger meshes.

All in all, I do think the algorithm presented in this paper gives satisfactory results. The quality of segmentations is objectively good, and while not the number one in terms of benchmark results, they position themselves nicely in the middle field. Considering the

goal of making fast, on the go optimizations to the user input, this trade off seems very acceptable.

APPENDIX A

Screenshots of segmentations evaluated with Benchmark

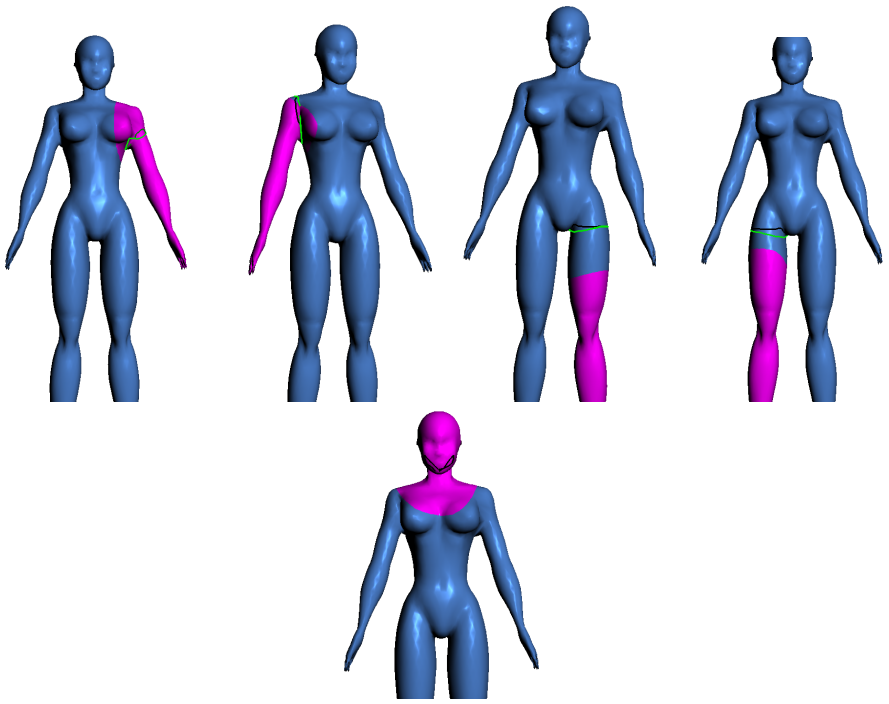


Figure A.1: Benchmark mesh 1

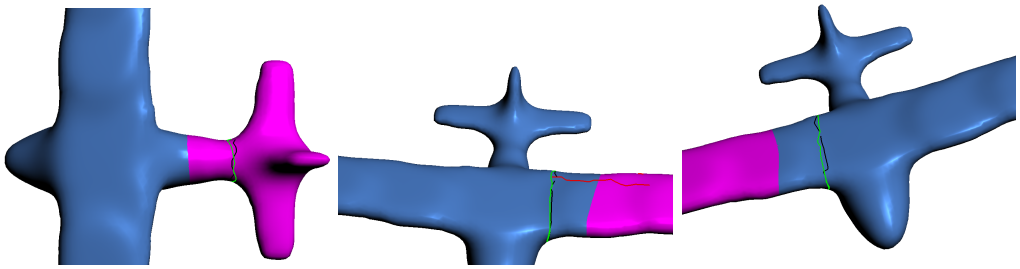


Figure A.2: Benchmark mesh 61

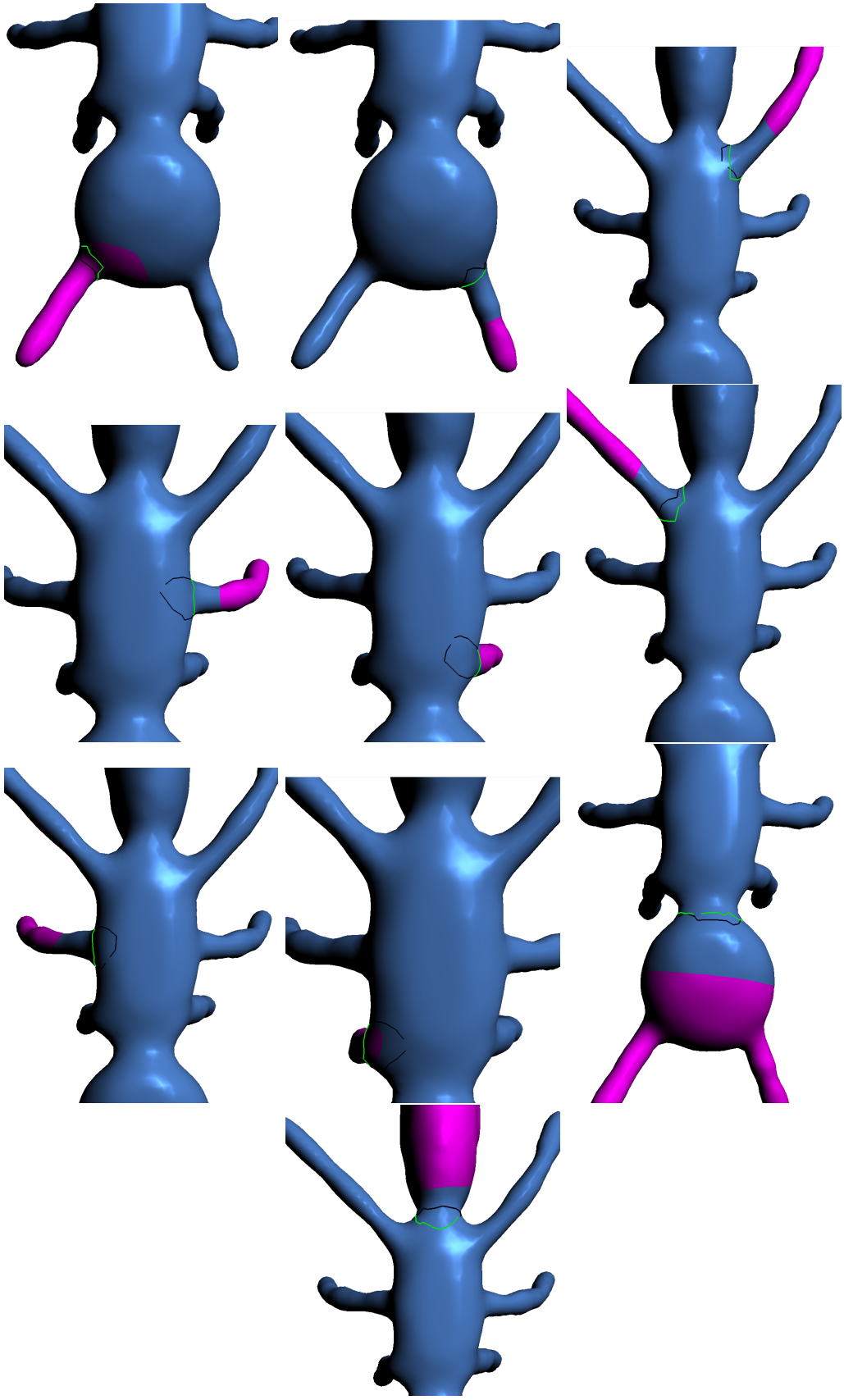


Figure A.3: Benchmark mesh 81

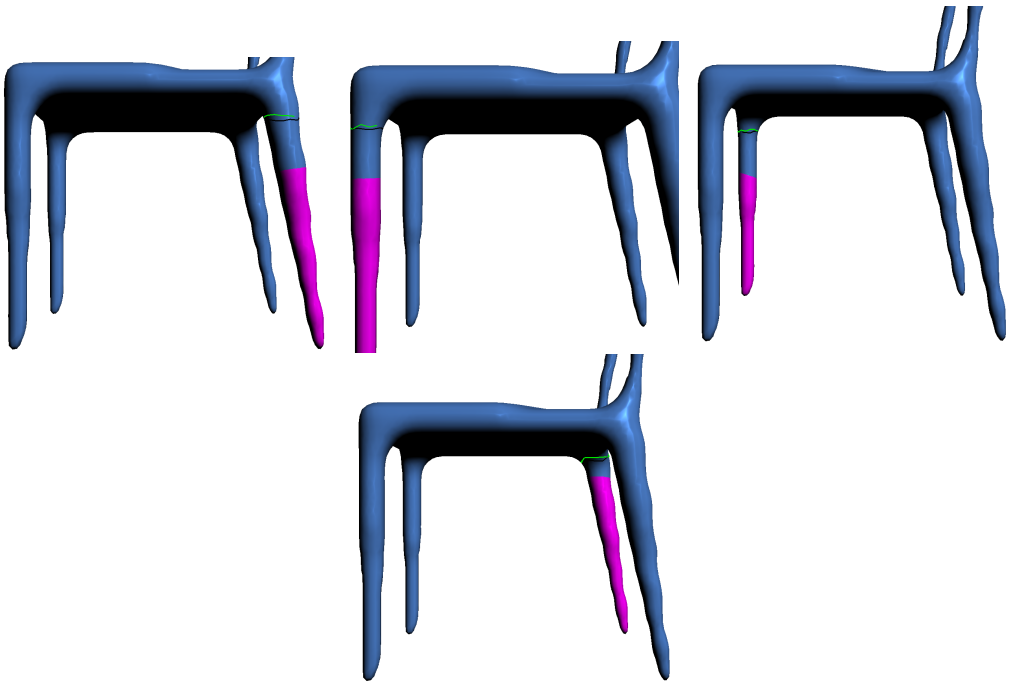


Figure A.4: Benchmark mesh 101

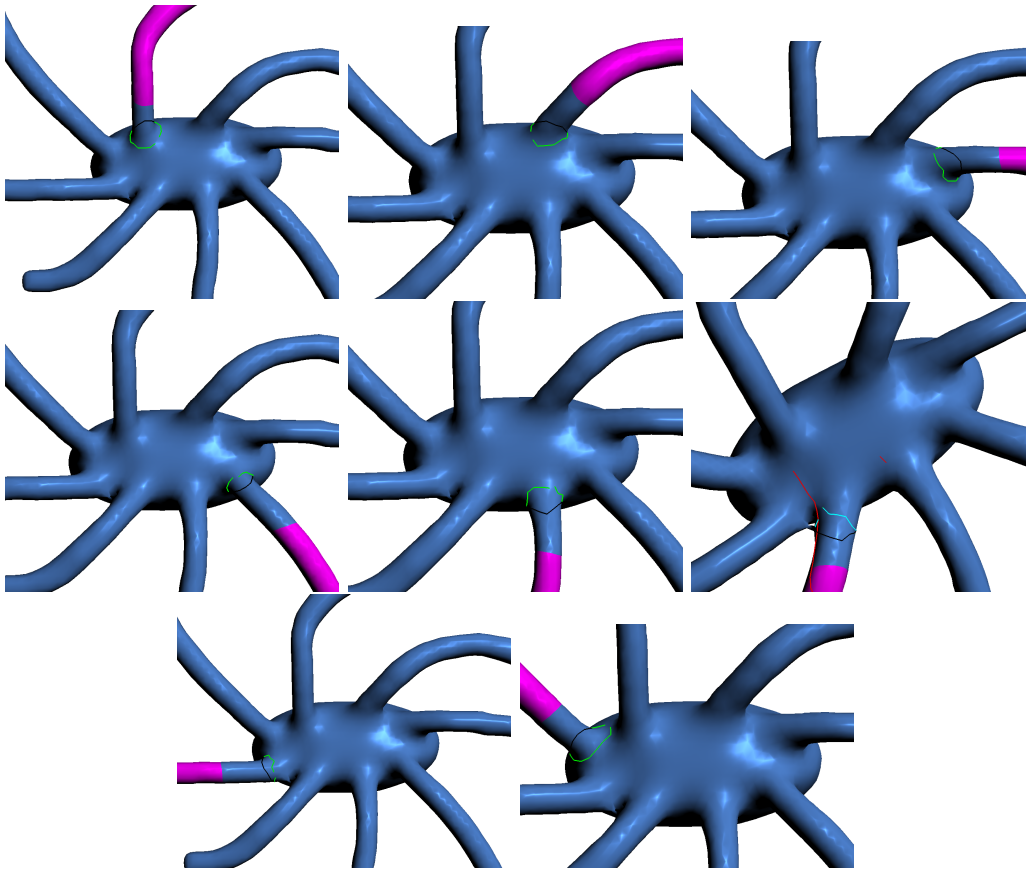


Figure A.5: Benchmark mesh 121

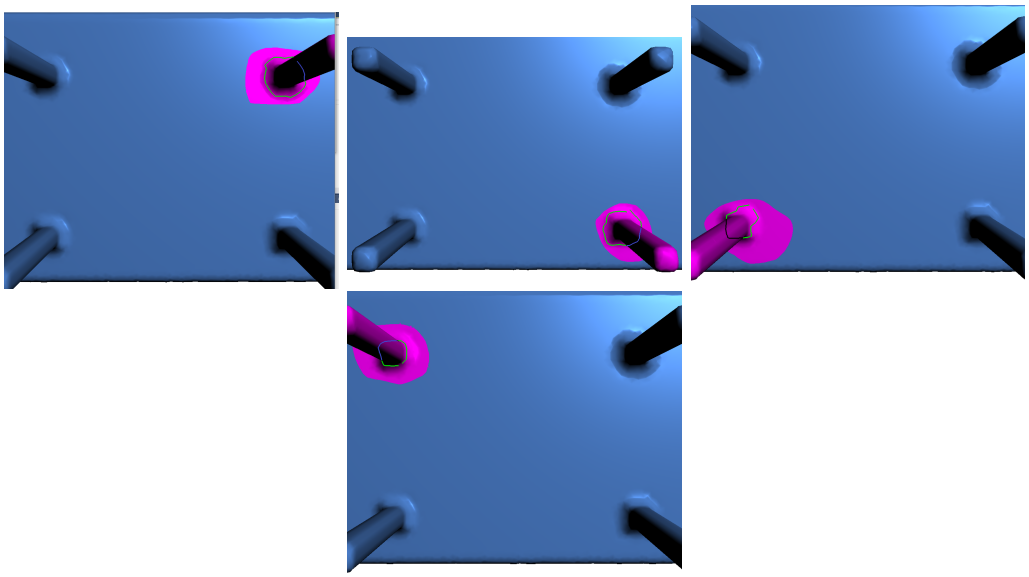


Figure A.6: Benchmark mesh 141

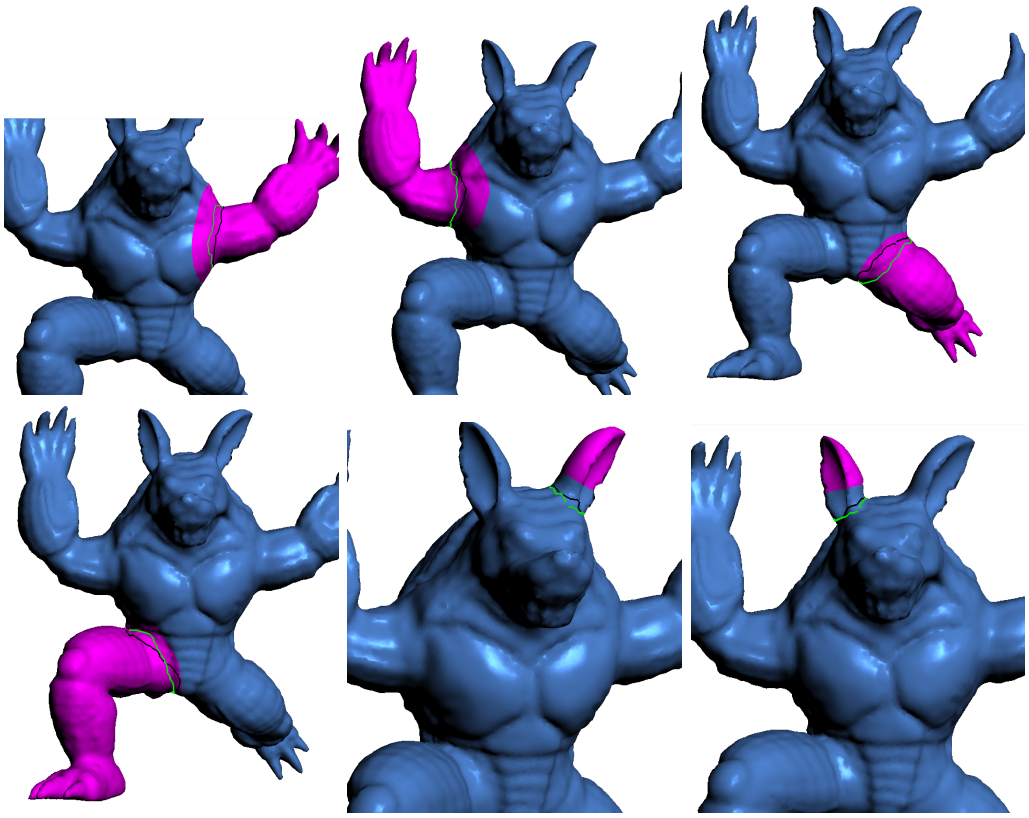


Figure A.7: Benchmark mesh 281

Bibliography

[AFS06] Marco Attene, Bianca Falcidieno, and Michela Spagnuolo. Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer*, 22:181–193, 2006.

[APP+12] Alexander Agathos, Ioannis Pratikakis, Stavros Perantonis, Nickolas S. Sapidis, and Philip Azariadis. 3d mesh segmentation methodologies for cad applications. *Computer-Aided Design*, 4:827–841, 2012.

[AZC+11] Oscar Kin-Chung Au, Youyi Zheng, Menglin Chen, Pengfei Xu, and Chiew-Lan Tai. Mesh segmentation with concavity-aware fields. *IEEE Transactions on Visualization and Computer Graphics*, 18:1125–1134, 2011.

[BLVD11] Halim Benhabiles, Guillaume Lavoué, Jean-Philippe Vandeborre, and Mohamed Daoudi. Learning boundary edges for 3d-mesh segmentation. *Computer Graphics Forum*, 30:2170–2182, 2011.

[CGF09] Xiaobai Chen, Aleksey Golovinskiy, and Thomas Funkhouser. A benchmark for 3d mesh segmentation. *ACM Transactions on Graphics*, 28, 2009.

[KHS10] Evangelos Kalogerakis, Aaron Hertzmann, and Karan Singh. Learning 3d mesh segmentation and labeling. *ACM Transactions on Graphics*, 29, 2010.

[LHMR08] Yu-Kun Lai, Shi-Min Hu, Ralph R. Martin, and Paul L. Rosin. Fast mesh segmentation using random walks. *Proceedings of the 2008 ACM symposium on Solid and Physical Modeling*, pages 183–191, 2008.

[SSCO08] Lior Shapira, Ariel Shamir, and Daniel Cohen-Or. Consistent mesh partitioning and skeletonisation using the shape diameter function. *The Visual Computer: International Journal of Computer Graphics*, 24:249–259, 2008.

[YWLY12] Dong-Ming Yan, Wenping Wand, Yang Liu, and Zhouwang Yang. Variational mesh segmentation via quadric surface fitting. *Computer-Aided Design*, 44:1072–1082, 2012.