Alexander Weinrauch, BSc

Skeleton Extraction using Natural Reeb Graphs

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Markus Steinberger
Institute of Computer Graphics and Vision

External Advisor

Dr. Rhaleb Zayer
Max Planck Institute for Informatics, Saarbrücken Germany

Graz, February 2020

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

Acknowledgment

First, I want to thank my supervisor Markus Steinberger and advisor Rhaleb Zayer for all the productive discussions, especially on Thursdays.
I also wish to express my deepest gratitude to my family, in particular to my parents Manuela and Alfons, for all the support in the good and bad times throughout my journey, which lead to this thesis.
Finally, I have to thank the person who kept me motivated to complete this journey and did never fail to be a helpful rubber duck when needed. Thank you, **Barbara**.

Kurzfassung

Eine vereinfachte Beschreibung eines 3D-Objekts ist eine wesentliche Voraussetzung für viele existierende Anwendungen in der Computer Grafik und Vision. Reeb Graphs sind eine gut erforschte Möglichkeit, um solch eine Beschreibung zu erhalten. Die Reeb Graphs werden durch eine Abbildungsfunktion über die Oberfläche definiert. Diese Abbildungsfunktion ist jedoch nur schwer ohne Benutzereingaben zu erzeugen. Diese Arbeit beschreibt eine Abänderung der Reeb Graphs, Natural Reeb Graphs, um Benutzereingaben zu vermeiden. Die explizite Abbildungsfunktion wird durch eine implizite Funktion ersetzt, die durch einen natürlichen Diffusionsprozess auf der Oberfläche gebildet wird. Die Natural Reeb Graphs erfassen alle Äste und Löcher einer Fläche korrekt, was für die Beschreibung der 3D-Form unerlässlich ist.

In dieser Arbeit wird zusätzlich auch ein Algorithmus zur Extraktion eines Skeletts auf der Grundlage von Natural Reeb Graphs vorgestellt. Skelette können in verschiedenen Bereichen der Informatik verwendet werden, beispielsweise Computeranimationen, Datenvisualisierung, Objektabfrage und Objektkomprimierung. Die produzierten Skelette unserer Methode sind zentriert und transformationsinvariant, entscheidende Merkmale für weitere Anwendungen. Außerdem benötigt unsere Methode keine Benutzereingaben und ist vollständig für die Grafikkarte (GPU) ausgelegt. Die Laufzeitkomplexität skaliert linear mit der Anzahl der Dreiecke des 3D-Objekts, was es uns ermöglicht, große und komplexe Objekte wie moderne 3D-Scans von Menschen zu handhaben. Unser Ansatz ist in Bezug auf die Laufzeit, aber auch in Bezug auf die Qualität der produzierten Skelette konkurrenzfähig gegenüber bestehenden Methoden.

Abstract

A high-level understanding of a 3D mesh is an essential requirement for many applications in computer graphics and vision. To obtain such a high-level understanding, Reeb graphs are a well-researched option. Reeb graphs rely on a mapping function defined on the surface of the object, which is challenging to create without any user input. To overcome this requirement, we propose a derivation of Reeb graphs, called Natural Reeb graphs. The explicit mapping function is replaced by an implicit function formed by a natural diffusion process on the surface. Natural Reeb graphs correctly capture all branches and loops of a surface, which is mandatory to provide a correct high-level explanation.

This work also presents a mesh skeleton extraction algorithm based on Natural Reeb graphs. Skeletons can be used in various fields of computer science, including computer animations, data visualization, object retrieval, and mesh compression. The output skeletons of our method are centered and transformation invariant, which are critical features for further applications. Additionally, the skeletons do not depend on any user-defined input. Our method is designed entirely for the graphics processing unit (GPU). The runtime complexity scales linearly with the number of triangles, which enables us to handle large and complex meshes like modern 3D-scans of humans. Our approach is competitive against existing methods in terms of runtime, but also in terms of quality of the produces skeletons.

Contents

List of Figures

List of Tables

1 Introduction

The Reeb graph [REE46] represents the topological skeleton of an n-dimensional object and providing essential information about the topology. Reeb graphs are generated by evaluating a continuous scalar function $f : \mathbb{X} \rightarrow \mathbb{R}$ on the topological space $\mathbb{X}$. Tracing the connected components in the level sets of f is the fundamental idea behind building the Reeb graph. The appearance of a new component, splitting, merging, and vanishing of existing components form the nodes of the Reeb graph. The correspondence of the affected components to previously generated nodes form the edges. Reeb graphs are a fundamental tool used in computer graphics, computational geometry, geometric processing, data visualization, and image processing. Some applications are object retrieval [AH12; Hil+01], surface understanding [Hil+01] or shape segmentation [WXS06].
Due to recent developments of high-resolution 3D scanners and big data applications, the demand for a fast algorithm to compute the Reeb graph or more general topological structures as a whole is increasingly growing. Concurrency and parallelism are two main options to improve efficiency, which are strongly supported by the most recent hardware developments. More and more cores in a single CPU which favors concurrency, but also the GPU has matured as a general-purpose highly parallel processor.
This paper focuses on parallelism and presents a method to approximate the Reeb graph for unstructured grids, which is designed to run efficiently on the GPU. Our method can not produce the real Reeb graph defined by a user-defined mapping function f, but instead is motivated by a recently published method called Layered Fields [**Fields**]. The main application for Layered Fields shown in the paper is natural tessellation. However, we show

that it can be used to approximate the Reeb graph without changing the method. We deduced the name "Natural Reeb Graphs" from this aspect.
The scalar mapping function f on the topology is implicitly extracted from Layered Fields growing on the surface of a triangulated mesh. The evolution of the energy sampled with high frequency provides the level sets needed to determine the critical points of the shape. Critical points signal a topological change. They are created from the events affecting the connected components of the level sets. In contrast to other approaches, our method cannot handle any arbitrary user-defined scalar mapping function on the topology. Therefore it should not be seen as a replacement for existing methods to calculate the Reeb graph.
Applications build on top of Reeb graphs rely on a well-defined mapping function to produce useful results. In Figure 1.1 three examples with different mapping functions are shown. The Reeb graph shown in the left image of Figure 1.1 provides a good description of the shape. The second example does already add complexity, which lowers the semantic value of the Reeb graph. However, especially the third example has many critical points that do not help to get a global view. The Reeb graph will, therefore, have a low semantic value for further applied algorithms. As described by Bajaj et al. .s [BPS97], finding a proper mapping function is not trivial and is often the main difficulty when applying Reeb graph-based methods.
Natural Reeb graphs, on the other hand, do not rely on finding a well-suited mapping function. The natural diffusion process of Layered Fields provides a mapping function that adjusts to the local geometry during the evolution process. Additional to meaningful Reeb graphs, this also provides transformation invariant Reeb graphs. Invariance to mesh transformations is a key feature required in skeleton creation and shape matching based on the Reeb graph [BPS97].

Skeleton creation for 3D objects based on natural Reeb graphs is one application covered in this paper. The idea of embedding the Reeb graph into the mesh to deduce a skeleton was already applied by previous methods. However, by exploiting additional information Layered Fields are offering, we can remove some common artifacts skeletons based on the Reeb graph

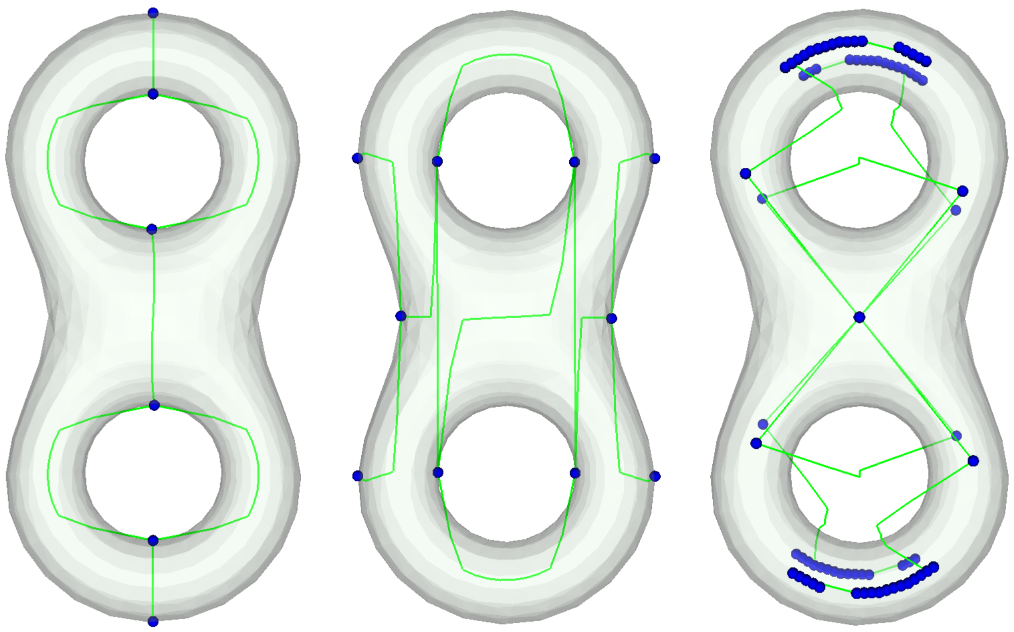

Figure 1.1: Three Reeb graphs defined by the y-axis (left), x-axis (middle) and z-axis (right) as mapping function. Blue points represent the ciritcal points and the green lines are the edges of the Reeb graph embedded into the mesh.

show.

The most common use case for mesh skeletons is animations, where a vertex position is influenced by one or more bones of the skeleton. This type of animation is generally referred to as skinning because the vertices are following the bones as skin would do. Existing methods are built around a mapping function over the surface to generate such a skeleton. However, the problem of finding an appealing visual mapping that is invariant to the orientation of the 3D object is still and open problem [Bia+03]. This comes from the fact that in contrast to the Reeb graph, the mesh skeleton has no real mathematical definition. Other methods try to define it as an optimization problem where the objective function is the distance to all vertices of the shape. These methods tend to create centered skeletons that may be desired based on the application. Modeling a human backbone is a good example where centeredness is not desired because it is not located in the center of the chest. Our method does not involve an explicit mapping function but still produces mainly centered skeletons. By not relying on a mapping function we also show that our method is orientation-invariant if the same starting points can be selected but even if this is not possible the skeleton is mainly orientation-invariant except for the starting region.

2 Background and Related Work

This chapter explains the idea behind Layered fields and covers existing work about Reeb graph computation and mesh skeleton extraction.

2.1 Layered Fields

The goal of this paper is to mimic natural tessellation on surfaces by having cells growing on the surface. These cells mainly start growing from a single distinct start vertex, called seed, but defining a start region is also possible. The boundary of a cell is modeled as a smooth transition of the real-valued energy where the value zero indicates no membership to the cell, and one means full membership. Each cell may grow on a separate layer, hence the name Layered Fields. As shown in Figure 2.1, cells on different layers block each other naturally during their growth process. At intersection areas, both cells have a smooth overlapping boundary. This formulation avoids numerical problems and discontinuities that would occur with sharp boundaries.
Layered Fields were designed to run fast and highly parallel on the GPU. The energy states in the system are stored in a $m \times n$ sparse matrix where m is the number of vertices and n the number of layers. To update the system state, the system matrix is multiplied by the Laplacian matrix of the mesh. The Laplacian matrix holds information about how the energy of a specific vertex should influence the energy of adjacent vertices. This is a very simplified explanation but covers all concepts which are needed to understand our method which uses Layered Fields.

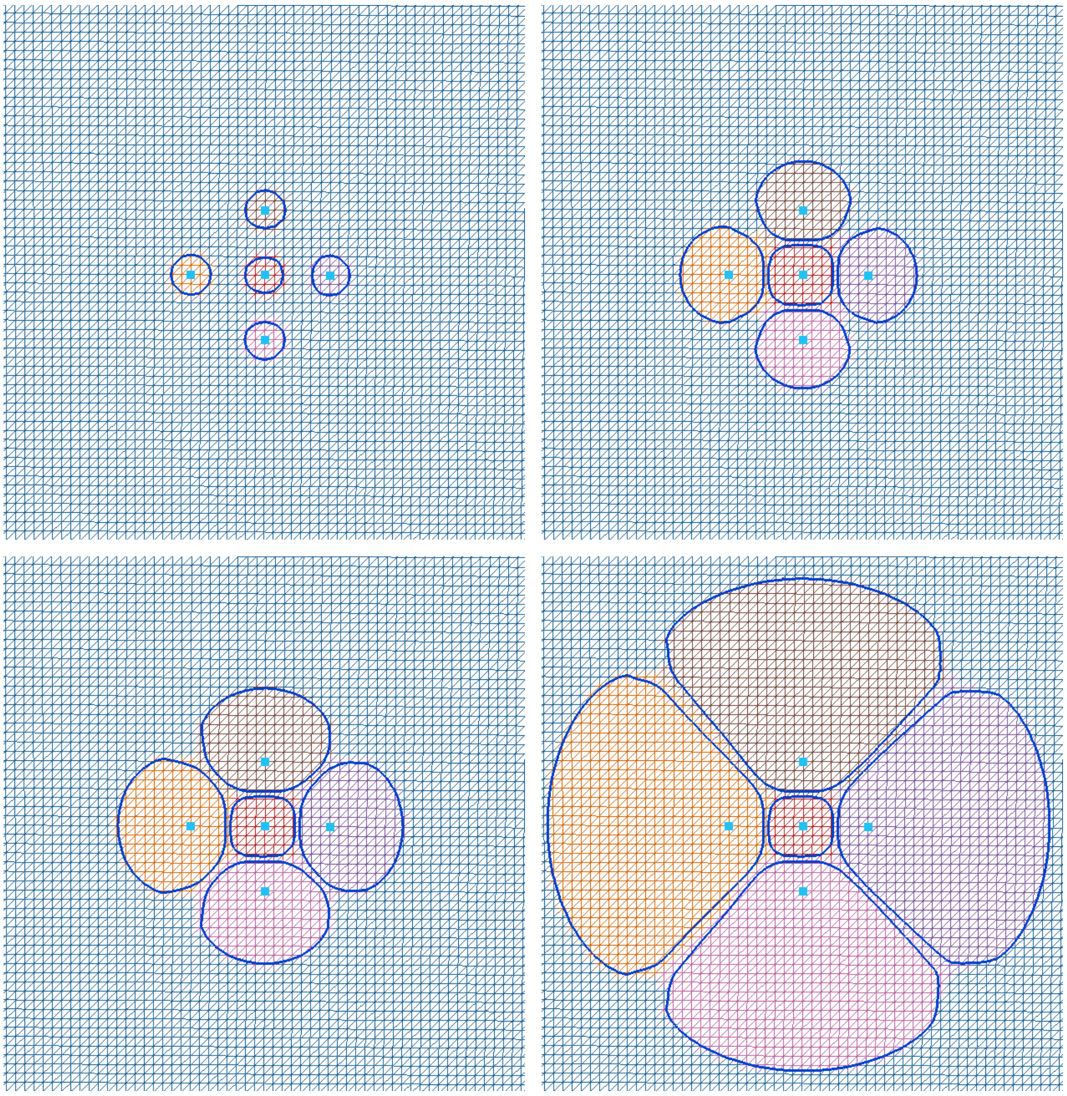

Figure 2.1: Growth of 3 layers on a plane. The light blue dots represent the start seeds and the blue lines the sharp boundary used to create Natural Reeb Graphs. In the image of the right the disability to overgrowth another layer is visualized.

The GPU-friendly design and available implementation were the motivation behind designing an algorithm which does also run nearly entirely on the GPU to avoid costly copy operations from the graphics card's memory to the system memory and synchronization points between CPU and GPU.

2.2 Reeb Graph methods

To our knowledge, there exists no method to compute or approximate the Reeb graph for unstructured grids designed for the GPU. Only algorithms for the contour tree, which is a connected and circle free Reeb graph, on structured grids can be found in the literature [AN15]. The first algorithm to correctly compute the Reeb graph dates back to 1991 [SKK91] and had a runtime of $O(n^2)$. Later, this was optimized to have a runtime of $O(n \log n)$ by maintaining a sorted structure for the input vertices. Pascucci et al. [Pas+07] published in 2007 an on-line algorithm that performed very well in practice despite its worst-case runtime of $O(n^2)$. At the time of writing, no faster algorithm can be found in the literature. It streams the triangles of the mesh and builds the Reeb graph on-line, by storing additional data for each node in the Reeb graph they can merge local Reeb graphs efficiently together if a new triangle connects them.
Tierny [BPS97] developed high-level Reeb graphs which do not require a user-defined scalar mapping function. Instead, feature points are extracted, which are invariant to rotation and scaling of the mesh. Each vertex is mapped to the closest feature point based on the geodesic distance. They introduce discrete level lines, which are traced during a geodesic propagation algorithm starting from the feature points. The idea of tracking during propagation is very similar to our method, resulting in transform invariant Reeb graphs.

2.3 Skeleton Extraction methods

There are many approaches to generate skeletons for 3D objects. Cornea, Silver, and Min in[CSM07] define classes of algorithms, namely Thinning and Boundary Propagation, Distance Field methods, Geometric methods, Reeb graph methods, and General Field functions. A discussion about the advantages and disadvantages of each class can also be found inside this paper. More recent work is based on mesh contraction [Au+08], where the mesh is contracted to a zero-volume shape by successive application of Laplacian smoothing. The curve-skeleton is then obtained by edge-collapsing while preserving the connectivity of the initial mesh. Inspired by this approach, Andrea Tagliasacchi et al. [Tag+12] have proposed mean curvature flow as the contraction method.
Manolas at al.[MLM18] showed how to adapt existing mesh contraction methods to make better use of multi-core computing systems. First, the mesh is segmented into multiple parts. Each part is then given to a mesh contraction method distribute over all available cores. They used [Au+08] as the contraction method because it was well-received at that time. They showed that the skeleton of each segment only depends on the vertex data, which allows exploiting data parallelism. In the second step, all sub-skeletons of the segmented mesh are merged. This is done by creating a candidate point to form an edge between two segments, based on the distance the best one is chosen to build the final connected skeleton.
The method shown by Wang et al. [WL08] is similar to the idea of mesh contraction. Instead of shrinking the triangle representation of the mesh, a voxel representation is shrunk. Iterative least-squares optimization is used to reduce the volume of the voxel representation while still preserving the geometric features of the mesh.
From the shrunk voxel representation, the skeleton is deduced. This method highly depends on the used voxel size, because features smaller than a voxel can not be preserved during the shrinking process. A different approach was developed by Sharf et al. [Sha+07] using a deformable model to reconstruct the given mesh. The reconstruction process uses competing fronts growing

inside the object until the real shape is approximated. The center points of the growing fronts are used as first candidate skeleton points. Candidates are filtered out on-the-fly based on the geometry of the front, the branching structure, and competition performance.

3 Natural Reeb graph computation using Layered Fields

This chapter starts with a high-level explanation of the algorithm and will afterward give an insight into the available implementation on the GPU.

3.1 Contour Lines for Natural Reeb graphs

Contour lines or level sets of a scalar function are sets of inputs for which the function takes a constant value c.

$$L_c(f) = \{(x_1, ..., x_n) | (f(x_1, ..., x_n) = c)\}$$

Existing methods analyze connected components while continuously changing the constant c. Our method instead uses Layered Fields as the function for which the level sets have to be defined differently. Layered fields can be seen as a natural diffusion processed on multiple layers. The diffusion process can be modeled as a function combining n sub-functions, where n is the number of layers. Each sub-function takes the timestep and a vertex as parameters and returns the energy e of the given vertex at the given time t on its layer.

$$f(t, v, i) = f_i(t, v) = e, \{e \in \mathbb{R} \mid 0 \leq e \leq 1\}$$

The energy of a vertex is in the range of $[0, 1]$. A value of zero states that the vertex does not belong to the cell. A value of one states complete membership to the cell. Values in between form the smooth boundary of a cell. Instead of changing the value of c, we are continuously changing t with a fixed c for all vertices of the unstructured grid. Setting the constant c between zero and one makes the contour line follow the smooth boundary. This models the contour line as a sharp boundary on top of the smooth boundary provided by the Layered Fields.
For the Natural Reeb graph calculation, tracking the contour lines directly on the vertices would be sufficient. For extracting the skeleton, on the other hand, a more accurate tracking is required to generate good-looking results. Based on this, we always track the contour lines on the faces level. This is achieved by looking at edges, more specifically at the two energy levels of the vertices forming the edge. The contour lines cross all edges, which have one endpoint larger and one endpoint less or equal to c. The exact crossing point can be computed by linear interpolation between the two endpoints based on their energy level and the constant value c. For manifold meshes and by the definition of the Fields, the contour line also has to go through all faces which use an edge that is crossed by the contour line. The point of the contour line on a face can then also be calculated by linear interpolation on the energy level of the vertices of the face. The exact contour line location is only needed when computing the skeleton. For the calculation of the Reeb graph, we only need the information if a vertex is part of the contour or not.

3.2 Track connected components

For manifold meshes, the contour lines defined by the sharp boundaries will always form one or more circles. In Figure 3.1 the sharp boundaries are visualized as dark blue lines. Before a circle splits into multiple circles, a vertex will be part of multiple circles representing a critical point. This can be seen in the highlighted section of the second image in Figure 3.1 on the connection between the thumb and the index finger. A circle in a graph is defined as a path with the same start and end vertex. Manifold meshes with borders can additionally form paths that end and start from border vertices instead of forming circles. These paths are caused by the fact that border vertices can be part of the contour line at some time point but may only have one adjacent face, which is also part of the contour. Figure 3.2 shows an example of a sharp boundary moving over a hole in the 3D object. The circles or paths form the connected components for each timestep t. The problem of finding those circles and paths is a connected component labeling problem. Connected component labeling will give the same label to all faces that are part of the contour lines and are reachable by traversing over the unstructured grid, ignoring faces which are not part of the contour.

The behavior of the connected components when changing t generates the events signaling a critical point. A relationship between connected components at timestep t and $t + \Delta$ is required to track the components. The naive way to track the connected components would be to analyze the location of their barycenters. The closest barycenter from the old iteration to the current iteration should be the origin of a connected component. This only holds for small movements of the boundaries between iterations, which requires a small time difference Δ between iterations. The naive way requires the calculation of the barycenters of all connected components in each iteration and comparison between all of them. Especially when the number of branches on the surface is high, and therefore the number of growing boundaries is high in a given timestep, the building and the comparing process might be very expensive. To calculate the barycenters, we need to compute all exact points of the level set and label them according to the connected component. Connected component labeling is an performance expensive operation on the GPU. Our connected components also represent the worst case for most algorithms because they form a circle without shortcuts. After computing the barycenters, the distance between all of them needs to be calculated and compared, resulting in $O(n^2)$ comparisons, where n is the number of active boundaries. A more sophisticated implementation

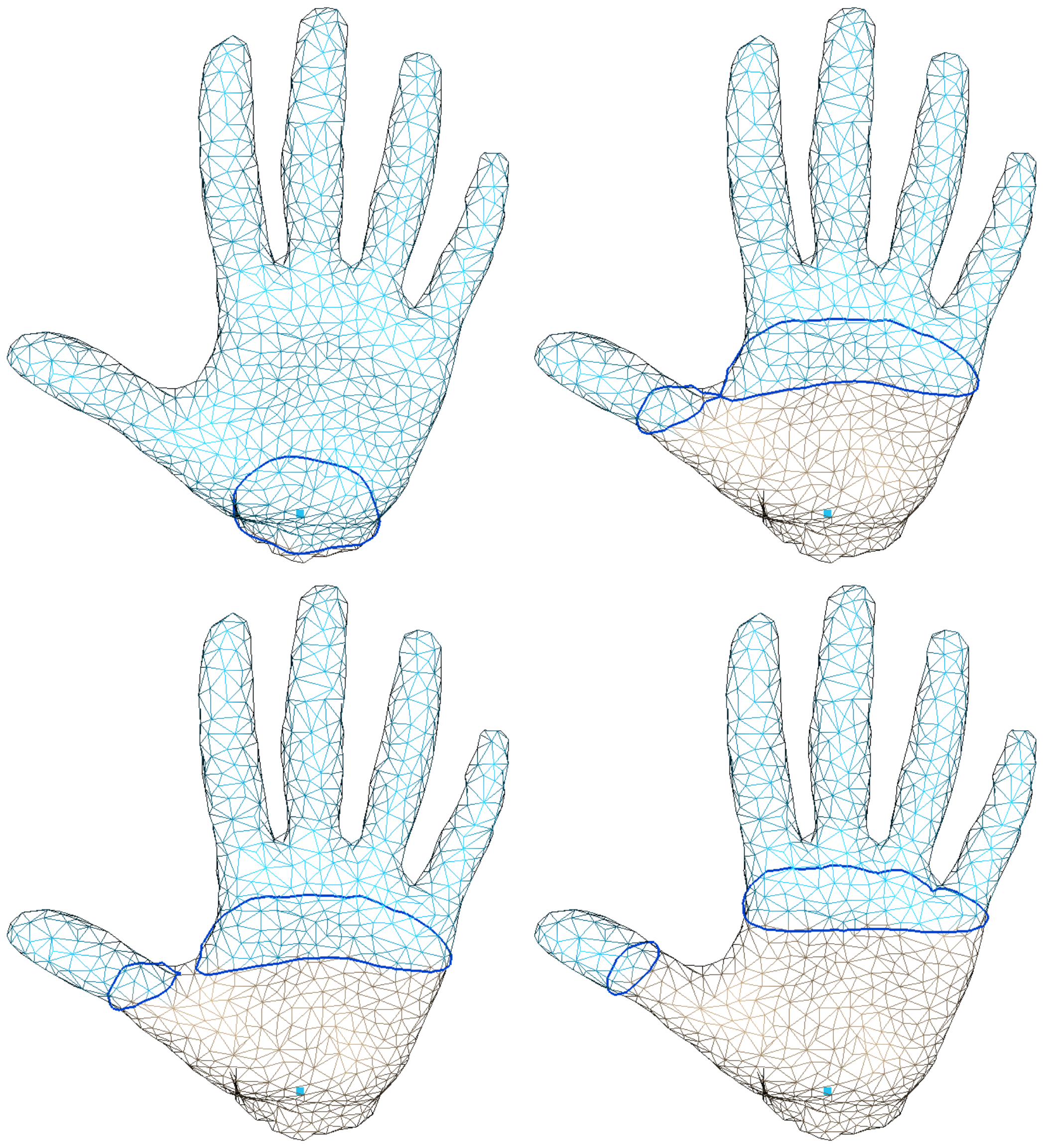

Figure 3.1: Sharp boundaries (dark blue lines) traced over the evolution of one cell. All brown vertices have higher energy than the threshold c, light blue vertices have less energy than c.

would involve a spatial data structure like an Octree, but building and updating such a structure for each iteration is also not ideal in terms of

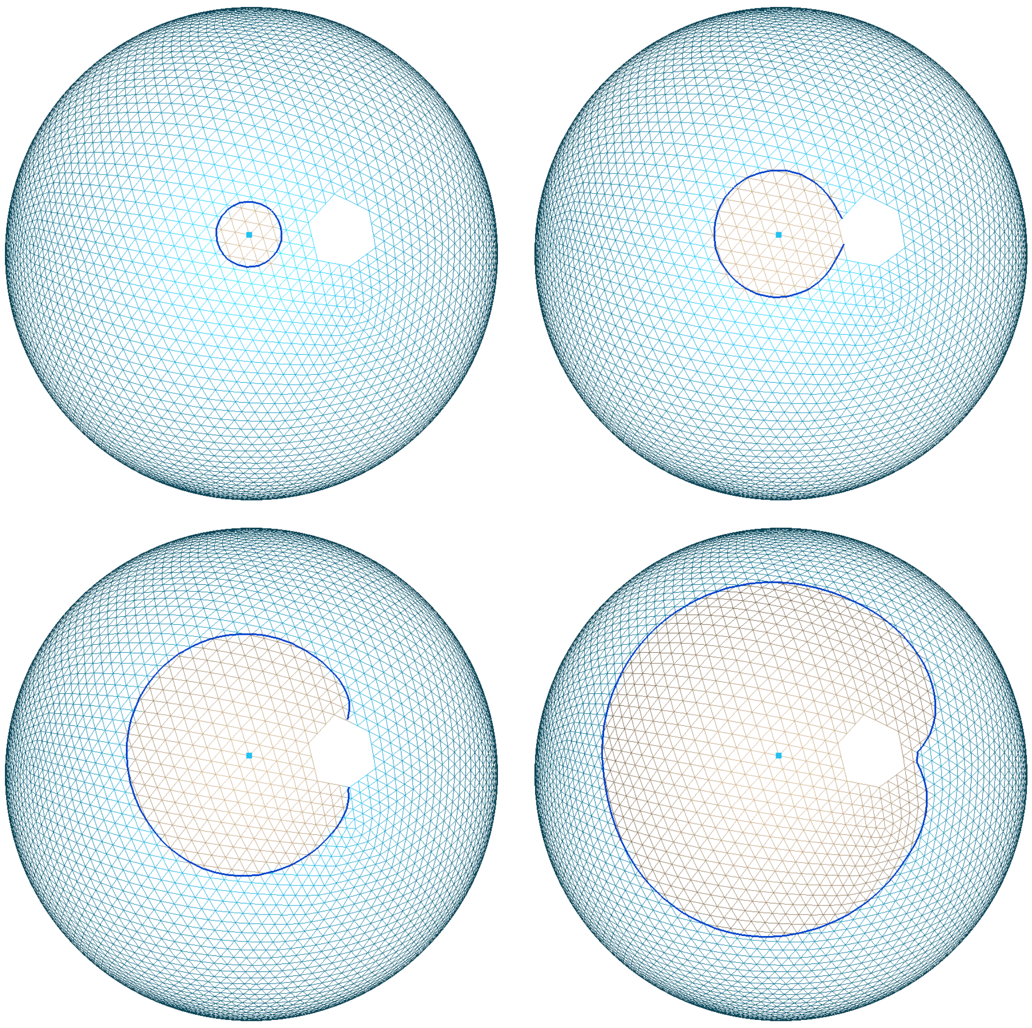

Figure 3.2: Shows the behaviour of a sharp boundary on a sphere with a hole in it. Note that the circle degenerates to a path while the hole is part of the sharp boundary.

performance. To avoid these expensive computations during all iterations, we use a different functionality provided by Layered Fields. The number of layers of Layered Fields is not fixed during simulation, so it is possible to add new layers on-line. We exploit this functionality by splitting a layer as soon as more than one connected component is detected on a layer. In other words, we only want one layer of Layered Fields to host exactly one connected component. If there are $n > 1$ components detected on a layer, we generate n new layers with the start region set to the vertices forming one connected component. Since one layer cannot overgrow another, the new layers will grow in the same direction as the old layer would have been without splitting. When comparing the first two images of Figure 3.3 with the evolution shown in Figure 3.1, it is clearly visible that the child layers behave the same as the parent would have without splitting.

The old layer is marked as finished after splitting because it has no free space to grow further. Instead of splitting into n new layers, it is also possible to let one component grow on the old layer and only create $n - 1$ new layers. However, this does not significantly increase performance and makes the Reeb graph edge extraction and debugging a bit more complicated. Layer splitting also provides direct access to the critical points at which one connected component splits into multiple new ones.
Another advantage of layer splitting is that the correspondence of triangles to connected components is only needed in the case of multiple connected components on one layer. Iterations without multiple components on one layer will occur far more often, and iterations that require splitting will be an exception. This plays an essential role in the performance of the algorithm. Based on this idea, we only need to perform connected component counting and can do an early exit if we detect only one component per layer. Connected component counting, in comparison to the more often needed connected component labeling, does only calculate the number of components, but does not tell which vertices form the components. Connected component counting can be implemented faster than labeling as counting is a sub-problem of labeling.
Identifying a vanished connected component can also be nicely handled by the idea of layer splitting. If no faces are marked as part of the contour on a layer, the affected connected component can be identified by the layer index. To calculate the critical point for a vanishing component, the contour triangles of the last iterations are needed because there is no information left in the current state of Layered Fields. This list is already computed for the

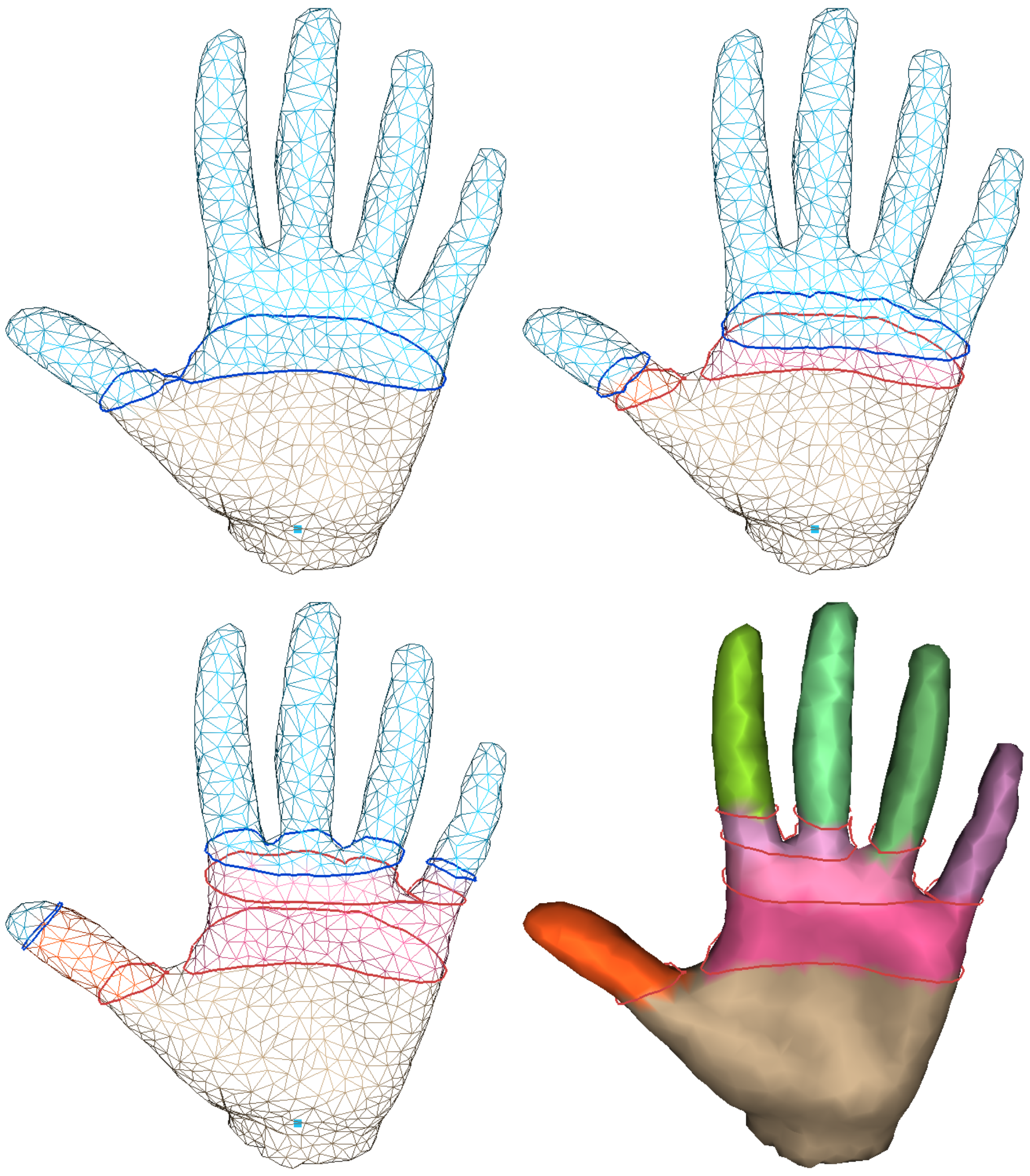
Figure 3.3: The evolution of a cell with layer splitting enabled. The face colors represent the layer index. The red contour lines show iterations where more than one contour line was present on a single layer.

connected component counting and is therefore available without a runtime cost.
The last category of critical points comprises merging connected components. This type cannot be directly modeled during simulation time like the other types. Instead, we are calculating them after the whole simulation is done. A definition of the simulation end is given in the next section. When two layers are growing towards each other, they do not overgrowth but instead, grow alongside each other. After the simulation, both layers have a contour line marking the border between those two layers. The border is the path where they have grown alongside each other. All vertices still part of a contour line are exactly those border vertices. All layers which have some amount of energy on those vertices should be considered merged. For the Reeb graph, we then need to merge all nodes of the layers which share energy on some vertices after the simulation. Due to the fact that this only needs to be done once after the simulation has finished, the performance of this operation is not as critical as compared to a procedure executed at every iteration of the simulation. In Figure 4.1 the two sharp boundaries at the bottom of the torus are an example of two layers blocking each other.

The tracking process described in this section creates all the critical points which are needed to create the Reeb graph. The nodes are the critical points described above, and the edges are the envolved layers during each event.

3.3 Simulation End

The end of the simulation is reached when it is not possible for any boundary to grow. However, Layered Fields do not offer a simple way to extract that information. Therefore, we need to define two different states of how the system may look when the simulation has completed.
On possible state is that all our leaf layers have vanished. Only leaf layers have the ability to grow, because, as already mentioned, parent layers can not grow further after splitting. Another possibility is that two or

more leaf layers did not vanish but are competing with each other over vertices separating them. To detect such a state, we periodically calculate the barycenters and the geometrical length of the contour lines and compare them to the previous calculations. If the difference is below a threshold for some amount of time, we consider the layer to be in its final state. The barycenter alone is not enough because the field may grow as a circle on a plane, left image of Fig.2.1. In such a case, the barycenter will stay the same even if the layer is still growing and will wrongfully detect stagnation. The geometric length of the contour line, on the other hand, grows or shrinks for such a case and does not signal stagnation. Growing alongside a tube with constant radius results in the reverse situation, where the length will not change, but the barycenter does.

3.4 Method summary

Figure 3.4 shows the steps described above applied on a double torus. In the first image, the initial connected component did create a node in the Reeb graph. The second image features a split of layer zero into two new ones. Hence two new nodes are created. Picture three shows one problem not discussed above. Layer one and two grow alongside each other until they reach the second loop. When reaching the second loop, both layers will have two connected components, the one blocking the other layer, and the new one advancing onto the loop. This triggers a split for both layers into two new ones. However, layers three and six start at the connected component representing the blocking front and will, therefore, have no space to grow and never build a contour line. Layers, which did not form a contour line during its lifetime, are then removed as a post-process-step. The last image has those two layers, three and six, removed, and also two new nodes were added to the Reeb graph. The introduced nodes are created based on the detection of two merging regions. Layer one and two have a merging region alongside the connection of the two loops, and layers four and five are connected on the top of the second loop. As previously

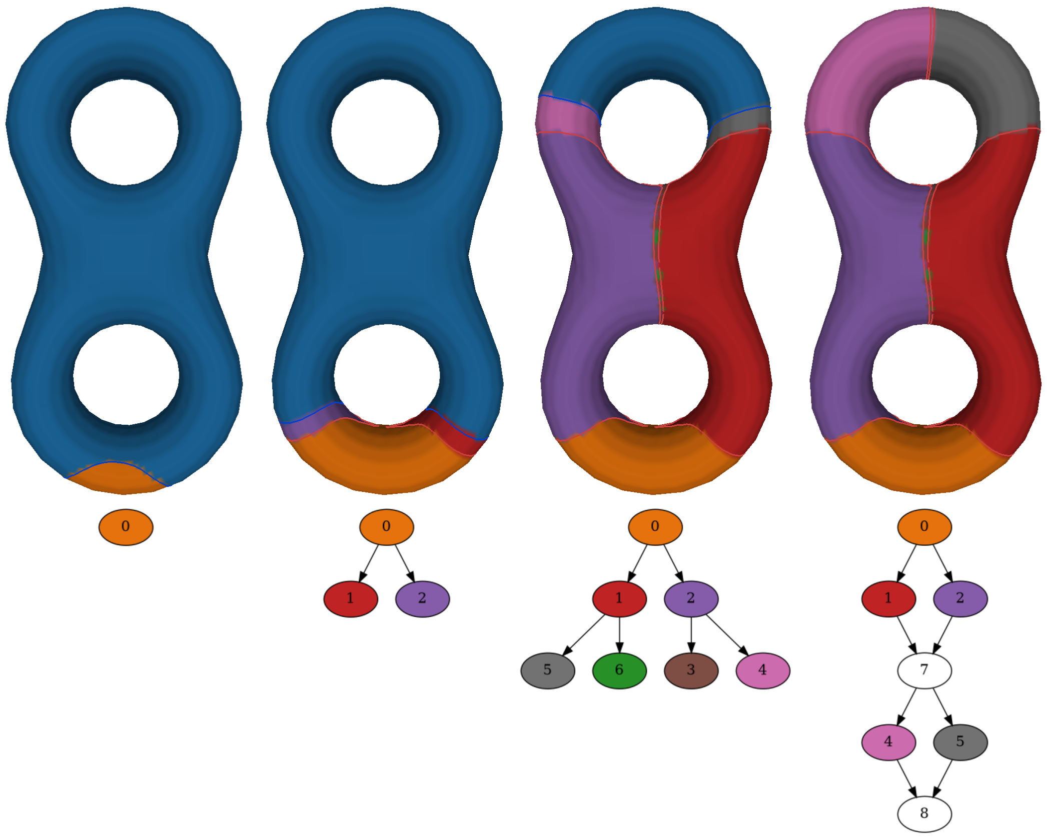

Figure 3.4:
System state at different simulation stages combined with the Reeb graph. Layer seven and eight are caused by the detection of merging layers.

described and shown in the final Reeb graph, those newly added nodes have incoming connections from the merging layers and do get all outgoing connections from those layers.

```
Initialize fields with one starting layer;
add Reeb node;
while more than one active layer do
    for each active layer i do
        calculate the number (n) of connected components of sharp
          boundaries;
        if More than one component then
            calculate exact connected components;
            split into n new layers;
            add n new Reeb nodes for each layer;
            mark new layers as active;
            mark current layers is inactive;
        end
        if zero components or layer growth halted then
            mark layer as inactive;
            add Reeb node for layer;
        end
    end
end
check for merging connected components;
```

Algorithm 1: Overview of the Natural Reeb graph extraction method

4 Skeleton extraction using Layered Fields

This section covers the algorithm and optimizations used to embed the natural Reeb graph as a skeleton into the given mesh. To understand the algorithm used in this section, it is important to understand section 3 because it is built on top of the Reeb graph calculation.

4.1 Skeleton points

A skeleton point can be compared to a joint of a human skeleton. The way joints are connected is determined by the application. The simplest model for the connection is a linear interpolation between two joints, which is equivalent to most human bones, which are a nearly straight connection between joints. Bones can also be modeled with higher-order polynomials or splines.

4.2 Skeleton point calculation

A precondition from the previous chapter is that precisely one connected component lives on one layer. This makes it very simple to calculate skeleton points for a given time t as we do not need to differentiate between multiple components. The first step is to calculate the exact points of the contour line on the crossed edges and faces. This is done by linear interpolation on the energy level of the endpoints of the edge or the vertices which form the triangle. This will produce two line segments for each triangle, from the interpolated points on the edges to the point interpolated on the face. It is not possible that all three edges of a triangle are part of the contour line. Note that each of these points is part of two line segments because each edge is part of two triangles. The interpolation on the same edge of two faces will result in the same point. To calculate the skeleton point, we sum up all interpolated points p weighted by the length $l(s1)$ and $l(s2)$ of its line segments divided by the total length d_{sum} of all line segments.

$$p_{skeleton} = \sum_{p \in P} \frac{1}{2 * d_{sum}} d(s1)d(s2)p$$

This sum where the contribution is weighted by the length is needed because the line segments will not all have the same length. This is also the reason why we cannot only take the average of all points. A comparison between the average and the weighted sum can be seen in Figure 4.1.

4.3 Sample rate of skeleton points

The time difference between iterations in the simulation is quite small, which results in small changes of the skeleton points between iterations. Many applications, like animation, require a rather sparse skeleton. Hence we need to define time points for which we want to calculate the skeleton point and use it in the final skeleton. Timepoints, where critical points occur, are good candidates since they signal a change in topology. Some methods [BPS97] extract feature points from the raw mesh to generate hints about where skeleton points should be extracted. Such methods could also be applied to our method. For this paper, we want to focus on using unique properties Layered Fields are providing over normal Reeb graph-based methods.

For tube-like structures, the growing fields tend to form the boundary representing the normal w.r.t the extensions of the tube. The amount of time the cell needs to adjust to the extent direction depends on the angle the cell did initially grow into the tube. This means the length of the contour line may be used to approximate the perimeter of the tube for the location of the skeleton point. Analyzing the recent history of skeleton points and estimated perimeters, we can detect the endpoint of such tubes. Rapid changes in the estimated perimeter supported by the slower movement of the skeleton point indicate the end. Such a feature detection is especially useful for human-like objects, where multiple parts can be detected by this method. A good example is the transition from the arm into the hand, which will not create a critical point because there is no topological change.

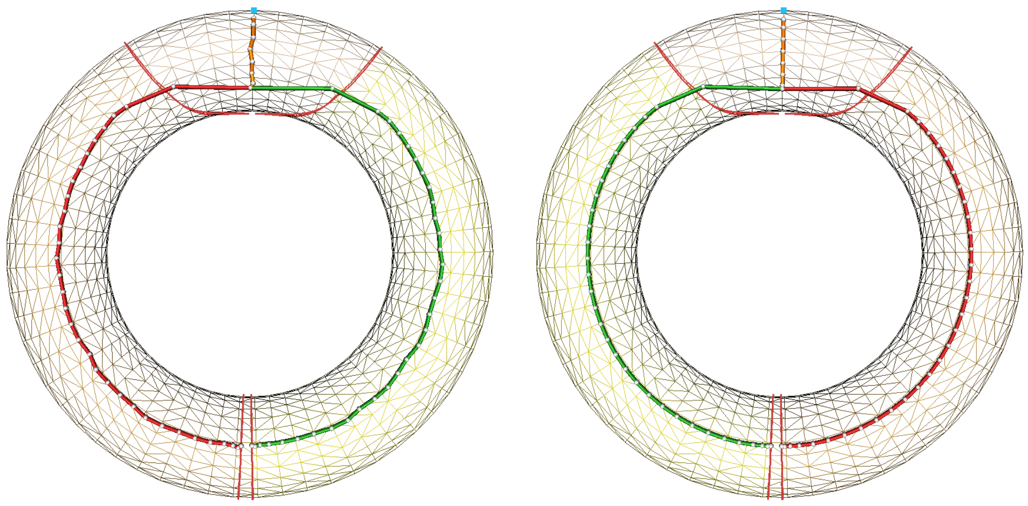

Figure 4.1: On the left, the weighted sum is used, and on the right, the average to calculate the skeleton points with a fixed sample rate. Note that the skeleton using the average does not follow the bend of the shape as accurately as the skeleton using the weighted sum.

4.4 Out-of-mesh bones

Another problem that occurs at critical points are bones that stick out of the mesh. New layers are not stable during the first few iterations, because the diffusion needs to distribute the high energy levels of the start vertices to surrounding vertices. Caused by the distribution process, the energy value of some starting vertices will drop below the threshold c in early iterations. If contour tracking is performed in this unstable state, multiple short paths may be detected, because some start vertices stayed above the threshold c, while others did not. These short paths would result in splitting the layer wrongfully. Therefore, contour line tracking cannot be performed during this unstable phase of a layer. The longer this phase lasts, the more inaccurate the tracking is around the splitting points. This may result in late detection of a further split, or worse, the split may not be detected at all, if a merge happens quickly afterward. Furthermore, this increases the chance of going out-of-mesh, because the layer will already have two or more advancing boundaries which cannot be tracked. This can be seen at the split between the index and middle finger in Figure 4.2. To detect such bones, we perform ray tracing against the mesh. This can be done concurrently while the simulation continues after the creation of a bone. If an out-of-mesh bone is found, we remove the bone and try to connect the later skeleton point to one prior generated. It is important to only test it against earlier joints. Otherwise, the final skeleton may not be fully connected. Skeleton points can also be outside of the mesh if the shape of the growing boundary is convex, because the weighted sum of the vertices of a convex shape may be outside of the shape. Connections to those skeleton points are allowed to cross the surface of the object and are ignored for this optimization.

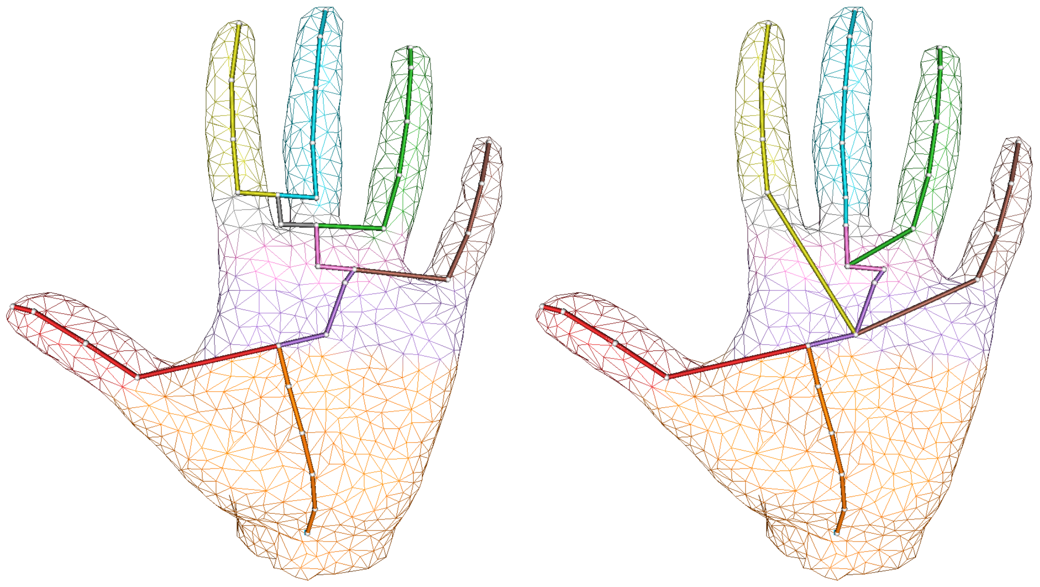

Figure 4.2: The left image shows the raw skeleton. On the right, the out-of-mesh connections are fixed, and leaf skeleton nodes of non-leaf layers have been removed.

4.5 Reverse growing

When a connected component splits into multiple new ones, a large jump is performed from the previous skeleton point to the first skeleton point of the new layers. This problem can be seen at the transition from the base of the hand to the thumb in the left image of Figure 4.3. A general way to avoid this non-optimal connection is to look at the n most recent skeleton points of the parent layer instead of always selecting the last one. Desired properties like distance, connection angle, and whether the connection leaves the mesh, can be used to select the best candidate. Choosing a different skeleton point may improve the skeleton, but it does not solve the problem of having any useful information exposed by the scalar function of this region. Instead, we modify the scalar function, which is implicitly defined by Layered Fields, by growing in the reverse direction after a split has occurred. This is achieved by removing the parent layer to allow the new layers to grow into the reverse

direction. The bones connecting the parent layer with the child layers are removed since the objective is to improve these connections. During the reverse growing, skeleton points are computed for the advancing contour lines. If the most recent skeleton point is close to an already existing skeleton point of the parent layer, the reverse growing is stopped for this layer. A connection to this close point is formed. Reverse growing is not guaranteed to find a suitable connection to an existing skeleton point. Therefore, the erased connection is restored.
To improve the success rate, only one child layer is allowed to reverse grow at a time, This is achieved by manipulating the Laplacian matrix for the seed vertices of the child layers for all except one. This is then repeated for all other child layers. Afterward, the parent layer may have a branch, which results in a leaf skeleton point, which should not be present in the final skeleton. Such a branch is removed, by continuously removing the last point of the parent layer, until it is connected to a child layer.

Reverse growing, as described above, does not work for small parent layers. The child layers do not have enough space to find a better connection. To give the reverse growing layer more space, additional parent layers may be removed. The reverse growing layer may then also create a connection to those additional layers. This gives the children more freedom during reverse growing but also decreases the chance of finding a good skeleton point to form a connection. Choosing the number of parent layers involved in reverse growing is a difficult problem to solve without user input. This is due to the missing information about the shape of the parent layers, which would require expensive shape analysis to obtain. However, the lifetime of the parent layer can provide information about the space provided.

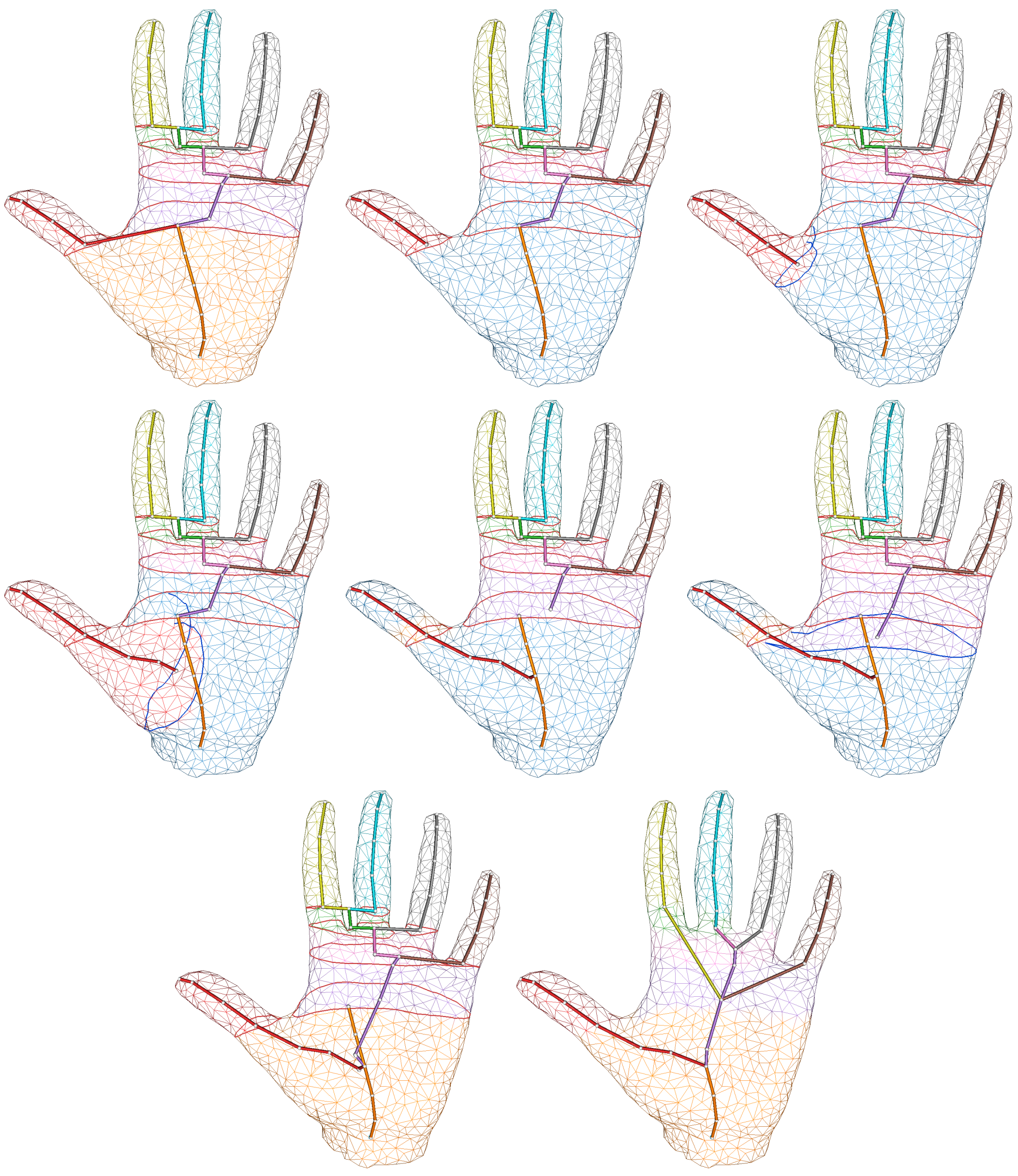

Figure 4.3: Example reverse growing process. The first image is the raw skeleton without optimizations. Image 2-7 show the reverse growing process and the new connection to the parent layer. The last image shows the final skeleton after removing the parent leaf skeleton points and all other mentioned optimizations.

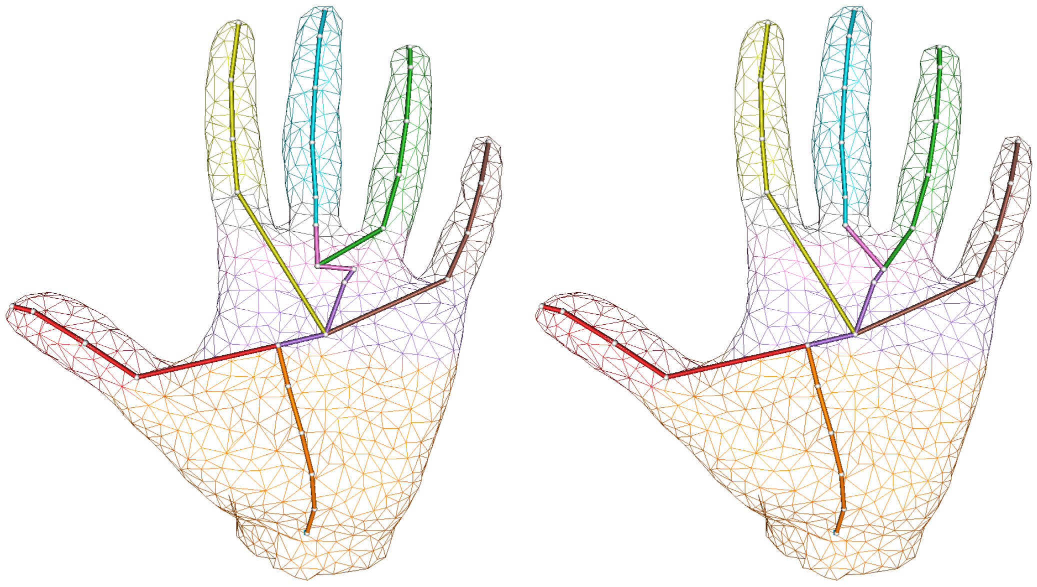

Figure 4.4: The left image shows the result after the out-of-mesh optimization. The right image shows the result of the support layer removal. The point of interest is the purple layer connecting the middle finger and the ring finger.

4.6 Removing of support layers

Multiple branches close to another may produce very small layers which do not offer important information for the skeleton. Furthermore, the bones produced by those small support layers are often very jaggy and have sharp angles which are not present the shape of the object. In Figure 4.3, the pink layer, connecting the middle and ringer finger, is classified as a support layer. It only has one additional skeleton point. The first, connected to the purple layer, and the last, connected to the turquoise layer, are generated by Reeb graph events. Our method tries to optimize such layers away to reduce the complexity of the skeleton. This is done by looking at the angles of bones leaving this layer. For the example above, the two bones connected to the middle and ring fingers are subject to optimization. If the overall connection angle is reduced by reconnecting these bones to an earlier one of the support layer, the old connection point is removed. The result of this optimization can be seen in Figure 4.4. This optimization is only performed if the new connections do not intersect with the mesh.

5 Results

The test system for the experiments consists of: a NVIDIA Geforce GTX 1080 TI with 11 GB of GDDR5X memory and 3584 compute cores, an Intel(R) Core(TM) i7-6850K, and 64 GB of DDR4 system memory. The Δt for the Layered fields update was set to 5.0.

5.1 Skeletons

Skeletons, shown in this section, were created without any user input or fine-tuned parameters. The only parameter of our method is the number of iterations tracking is not performed after a split happened, which is set to 20 across all examples. All optimizations previously mentioned, except reverse growing, were applied during the generation of the skeletons. Reverse growing was excluded because it requires user input to work reliably. One start layer with one randomly chosen start seed is used in the experiments, except for Figure 5.2, which aims to show the differences produced by different initial seeds. Figure 5.1 shows that our method does include the essential features of the stag mesh. All branchings and endpoints of the antlers, nose, and ears are well preserved. The front feet are well connected to the base of the stag. The connection of the back feet, on the other hand, shows some differences between the left and the right foot. The sharp boundary which is advancing towards the back feet was still a bit skewed caused by the starting seed lying on the left side of the mesh, which caused the difference in branching to both back feet. Detecting, not to mention

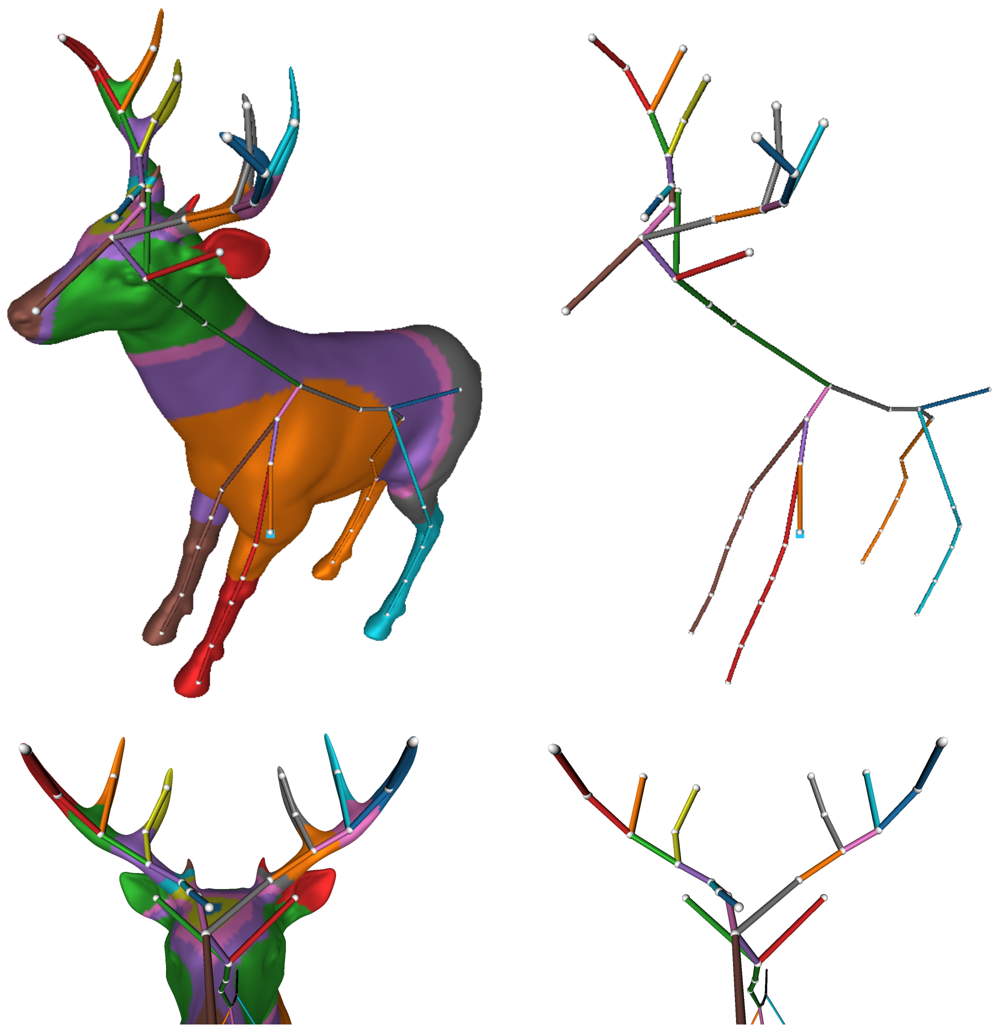

Figure 5.1: Skeleton of a deer mesh. The initial seed was located on the left front leg.

fixing, such difference is a challenging problem without having a global view of the mesh.

If the same starting seeds are selected, our method is invariant to affine transformation. Furthermore, Figure 5.2 shows that the start seeds impact only the starting region. Skeletons for parts of the mesh farther away from the starting seeds look very similar. For example, the skeletons for the head of the human are nearly identical even though the starting seeds were located on entirely different parts. The fourth image has a different skeleton in the head region because the start seed was located at the top of the head. These similarities, produced by our method, can be observed on all limbs too.

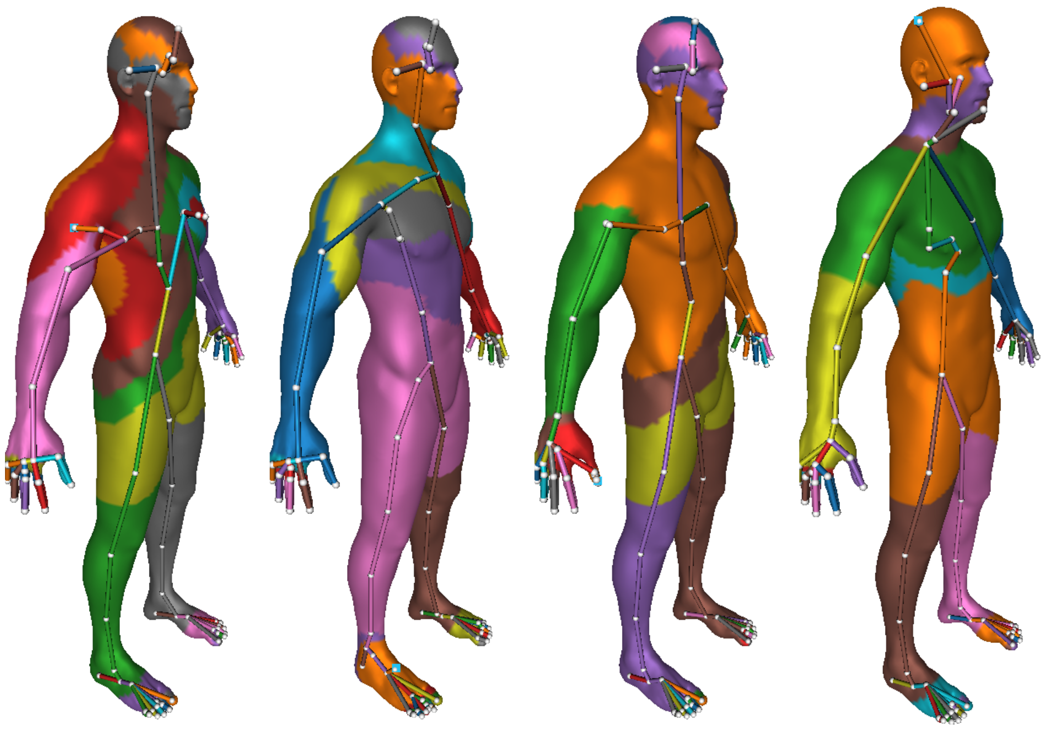

Figure 5.2: The skeleton generated for four different starting seeds. The locations from left to right are: left shoulder, left foot, left thumb, and the top of the head.

The tail of the camel in Figure 5.3 shows an example where the skeleton

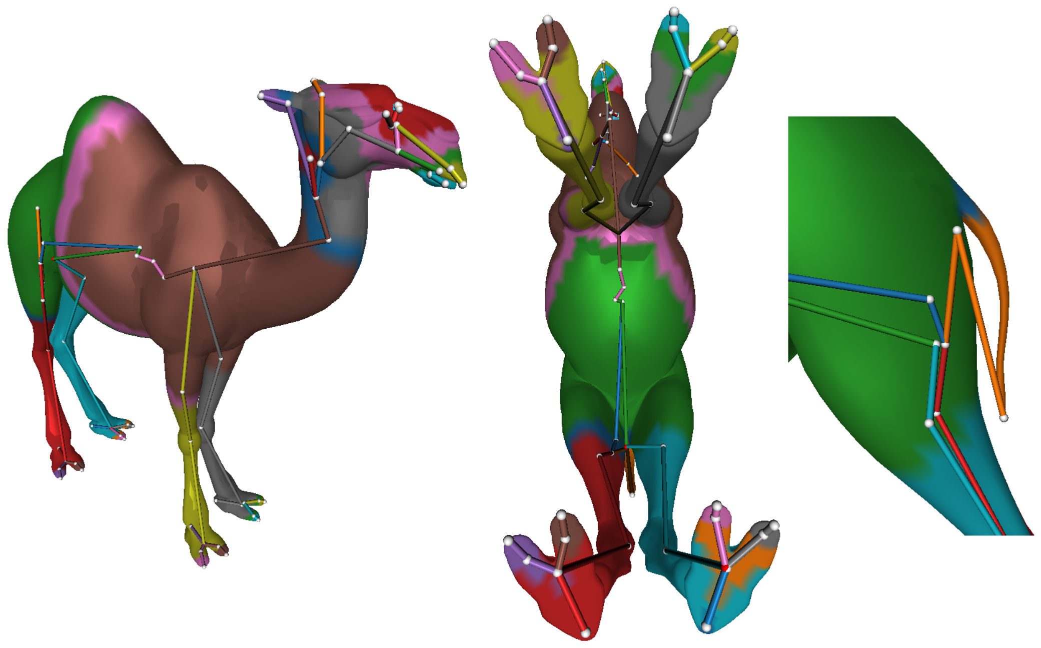

Figure 5.3: Skeleton generated by our method for the camel mesh. The tail features are out-of-mesh connection.

leaves the mesh where no applied optimization can solve this out-of-mesh connection. The diameter analysis fails because the tail has no rapid changes. As a post-process fix for the tail, reverse growing could be used, starting from the last point the contour line visited on the tail. As a fix during simulation time, the skeleton point for all sharp boundaries could be created in every iteration and checked if the connection to the previous point leaves the mesh. If the ray between those two points did not leave the mesh, the point is discarded. In the event that it intersects, the skeleton point of the last iteration is added as a new skeleton point. The disadvantage of this fix is that the skeleton will not be centered and a performance penalty because the skeleton point is needed in every iteration.

The dragon [Lab96] is a challenging mesh to produce a skeleton without any user input. It has a lot of small features, for example, the spikes along the back of the dragon, which should not be part of the skeleton. The tail is connected to the back, which forms a loop the skeleton has to represent. Our method produces a good looking skeleton shown in Figure 5.4. The open mouth and also the more significant spikes on the head are accounted for by our generated skeleton. The body shape is also well modeled. However, the front feet are not really captured. Our method cannot handle features that form tubes with a small elongation compared to the diameter. In general, for such features, like the front claws, the sharp boundary of the parent layer (yellow) does not grow around a single claw, and therefore no split will happen. The same can be seen in the image on the right bottom, where the upper claws were detected, but the lower claws did not. The worst part of the dragon is clearly the area marked by the white rectangle. This part is shown from another angle in the picture on the right bottom. The dragon is a 3D scanned model and has a hole at the bottom. The triangulation in this

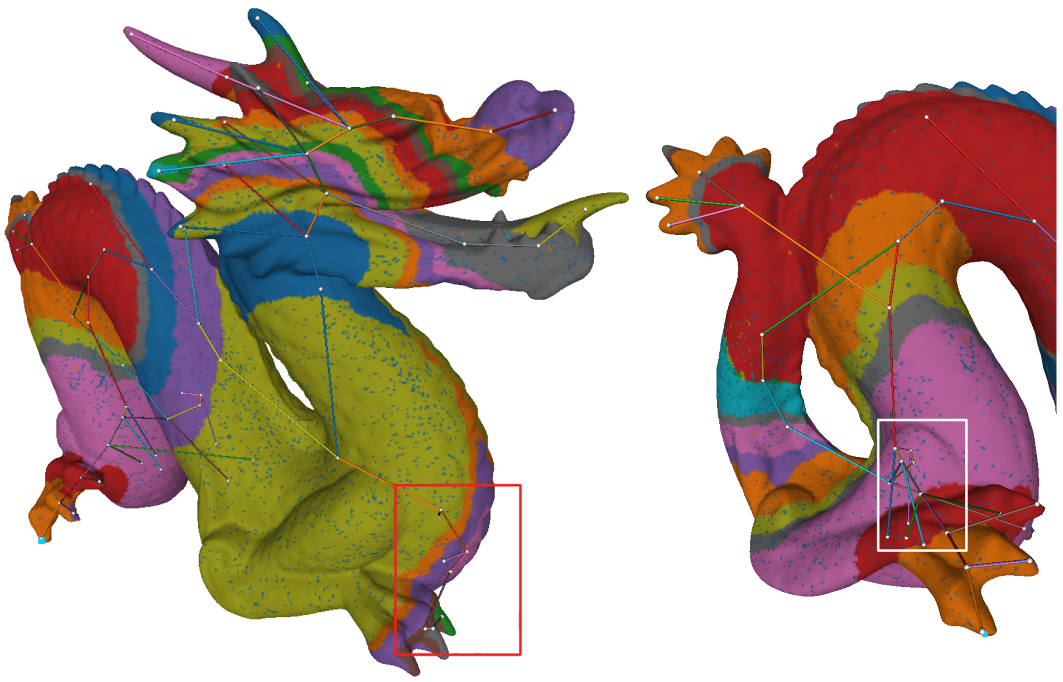
Figure 5.4: Skeleton generated by our method for the dragon mesh. The red and white rectangles highlight problematic areas.

area is very irregular, and faces with a tiny area are present. The advancing fronts in this region show some numerical problems which create a bad skeleton for this region.

5.2 Runtime

This section analyses the runtime for each step of our method to generate skeletons based on Natural Reeb graphs. The performance overhead of creating skeletons additional to Natural Reeb graphs is minimal, and therefore no separate discussion for creating only Natural Reeb graphs is provided.

5.2.1 Contour Extraction

We expected the runtime of the contour extraction to be linear to the number of active layers during an update step due to the serial implementation, which processes layer after layer. Figures 5.5, 5.7 and 5.9 show exactly the expected scaling in runtime. Between iterations 2600 and 4000 in Figure 5.5 for the dragon mesh, there was only one active layer that was growing alongside the body. In this time frame, the runtime for the contour extraction stayed nearly constant, which supports our initial expectation. The number of contour triangles did increase by about 600% in this time frame shown in Figure 5.6. This increase resulted in more writes to global memory to store the result and did increase the time needed in each iteration. However, these writes to store the contour triangles are highly regular and are efficiently executed by the memory controller of the GPU. The increased runtime for those stores is overshadowed by the reads of the energy values for each edge. Obtaining the energy values from the sparse system matrix requires irregular memory access patterns and depends only on the number of triangles of the mesh.

5.2.2 Connected Component counting/labeling

Connected component labeling has the same linear scaling with the number of layers as Contour Extraction. The serial implementation again causes this scaling. Additionally, the connected component labeling scales by $O(m \log m)$ with the number of contour triangles. Between iteration 2600 and 4000 in Figure 5.5, the increase in runtime caused by the increase of contour triangles is visible. However, none of the tested meshes had a large enough number of triangles forming the contour to impact the performance of the algorithm. When looking at Figures 5.6, 5.8, and 5.10 the peak for the contour line sizes lies at around $0.5\% - 1.5\%$ of the triangle count of the whole meshes.

5.2.3 Skeleton Point Calculation

The reported runtime for skeleton point calculation includes the runtimes for creating a skeleton point based on Natural Reeb graph events or the perimeter analysis. It is important to note that in case the perimeter analysis is disabled, there would be zero time spent for iterations where no layer splitting happens. Perimeter analysis depends on extracting the perimeter and skeleton point in each iteration to make a decision about changes in the perimeter for a segment of the mesh. To calculate the skeleton point, the perimeter is needed for the weighted sum. This is the reason why both values are reported as one. The runtime for these steps scales only with the number of layers for our experiments. This is due to the low utilization of the GPU. Calculating the skeleton point only involves iterating two times over the contour triangles and reading vertex positions and energy values, which both involve irregular memory access patterns. Furthermore, the number of contour triangles is too little to hide these memory access latencies.

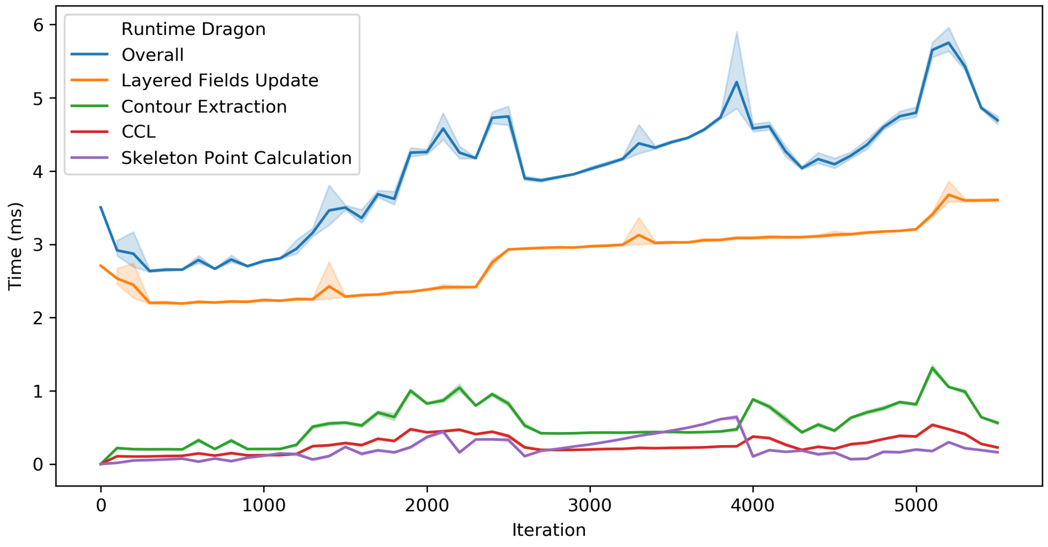

Figure 5.5: Average runtime over five runs for the dragon mesh. The variance of each iteration is shown as the area around the line.

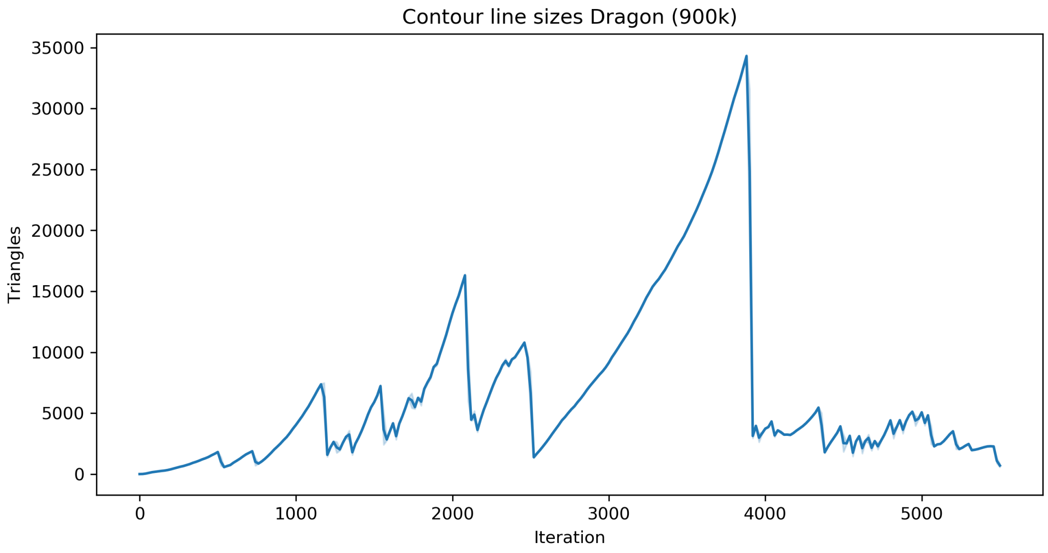

Figure 5.6: The number of contour triangles in each iteration for the simulation on the dragon mesh.

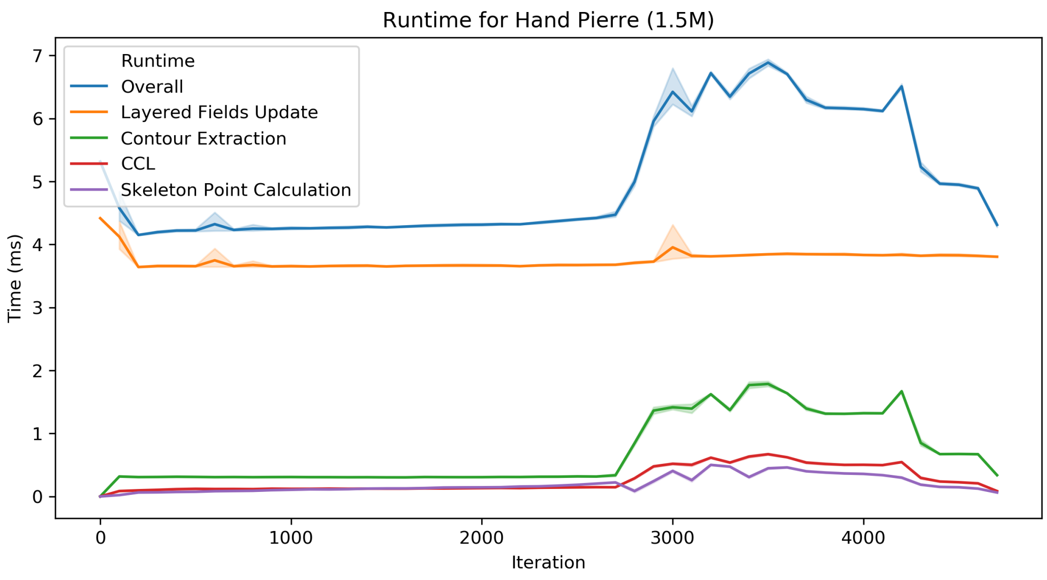

Figure 5.7: Average runtime over five runs for the hand Pierre mesh. The variance of each iteration is shown as the area around the line.

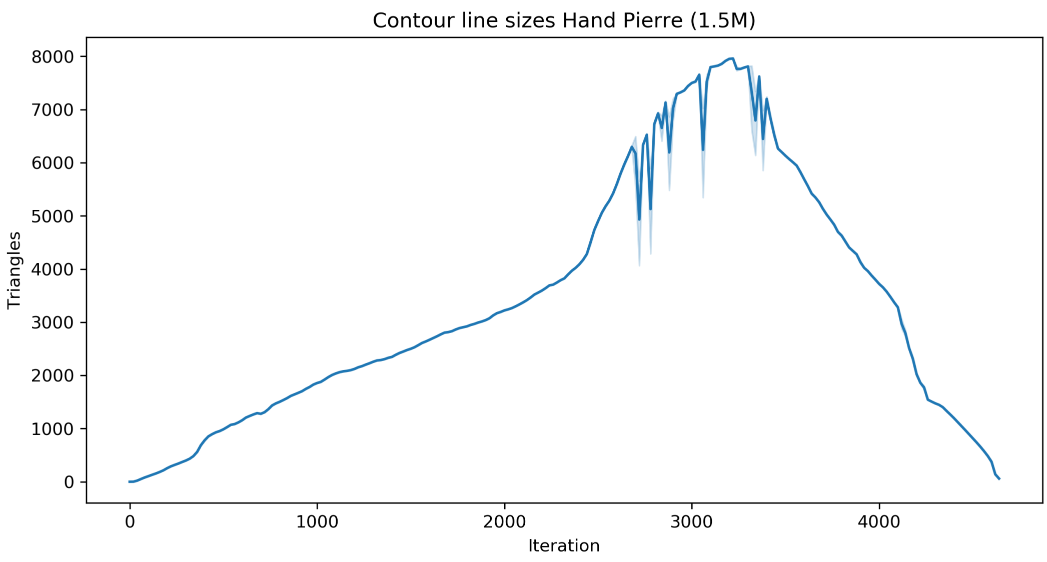

Figure 5.8: The number of contour triangles in each iteration for the simulation on the hand Pierre mesh.

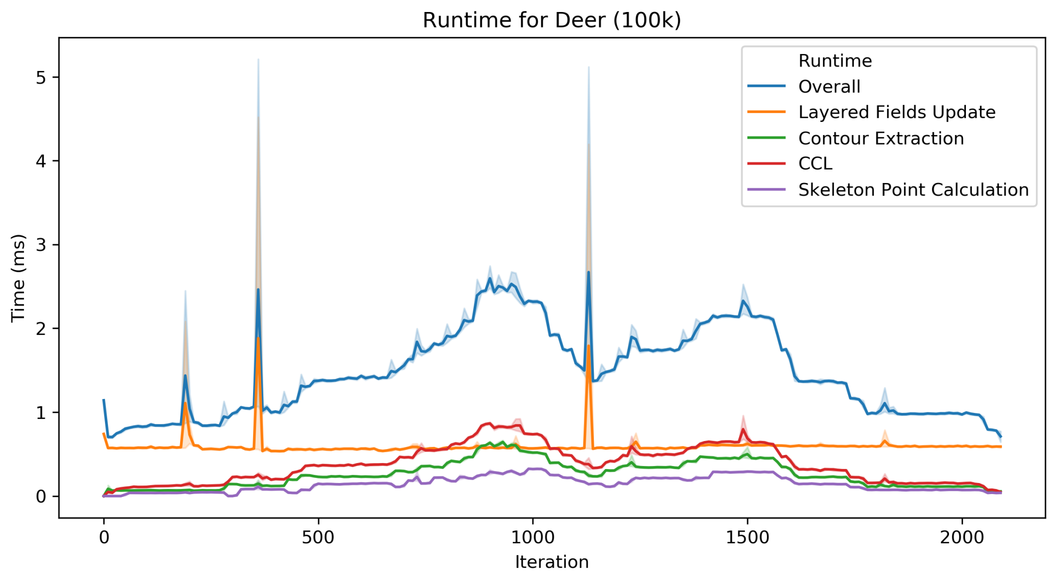

Figure 5.9: Average runtime over five runs for the deer mesh. The variance of each iteration is shown as the area around the line.

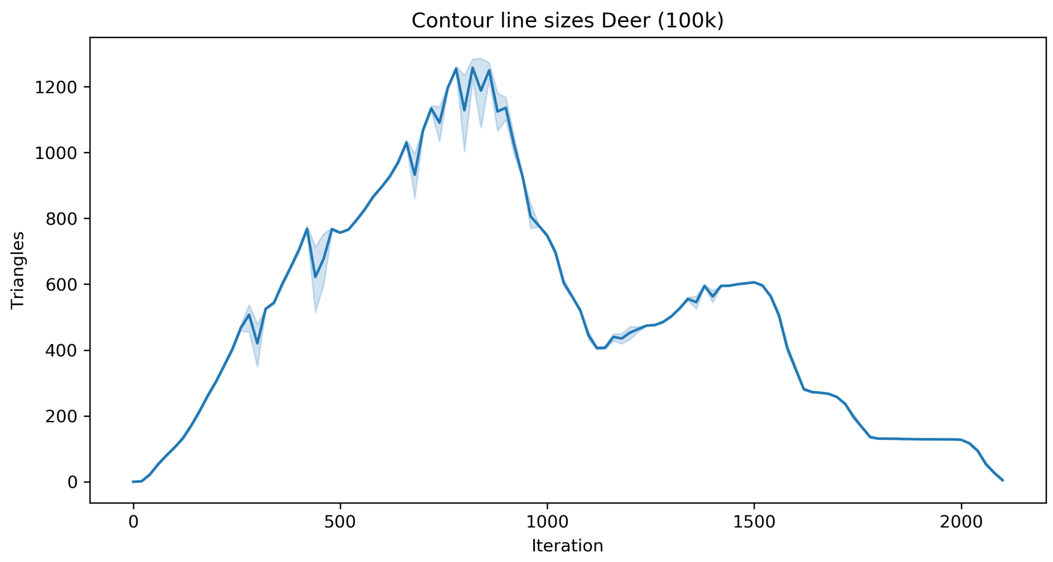

Figure 5.10: The number of contour triangles in each iteration for the simulation on the deer mesh.

5.2.4 Overall

Table 5.1 shows the total runtime and detailed information about each step of the method for various meshes. As shown in Figure 5.11, when looking at the runtime composition, the steps which need to be executed in each simulation iteration take the most time: contour creation, connected component counting/labeling, and skeleton point creation. This confirms our initial expectations. An important observation is that the Natural Reeb graph extraction and skeleton creation only takes about one-quarter of the runtime the update step of Layered fields needs. This lowers the possible gains in runtime when further optimizing the implementation of our method.

5.2.5 Comparison

We achieve a performance that is comparable with the mesh contraction method shown by Bajaj et al. [BPS97]. The most complex mesh covered in their work has 70k triangles with a runtime of 2.25 seconds for the parallel implementation and 5.27 for the single-threaded implementation. Our method needs for a similar-sized mesh (100k) around 3 seconds. The initial contraction based mehtod [Au+08], which is used by Bajaj et al. [BPS97], reported 240 seconds for a dragon mesh on a very outdated CPU. However, even with the runtime halved to account for a faster, more recent CPU, it is still by far slower than our method, which needs 21 seconds for 900k triangles.

Mesh(Triangles)	Total	Fields update	Contour extraction	CCL	Skeleton point
Hand Pierre (1.5M)	23.183	17.332	32.87	1.232	0.924
Male (1.4M)	40.604	28.657	7.222	2.434	1.616
Dragon (900K)	21.973	15.319	3.075	1.445	1.184
Hand arc [han19] (800K)	15.147	11.009	2.086	1.170	0.615
Deer (100K)	3.079	1.240	0.546	0.789	0.301
Human (50k)	1.373	0.384	0.262	0.467	0.141

Table 5.1: Runtime in seconds for different sized meshes. Only important steps of the method, in terms of runtime, are listed.

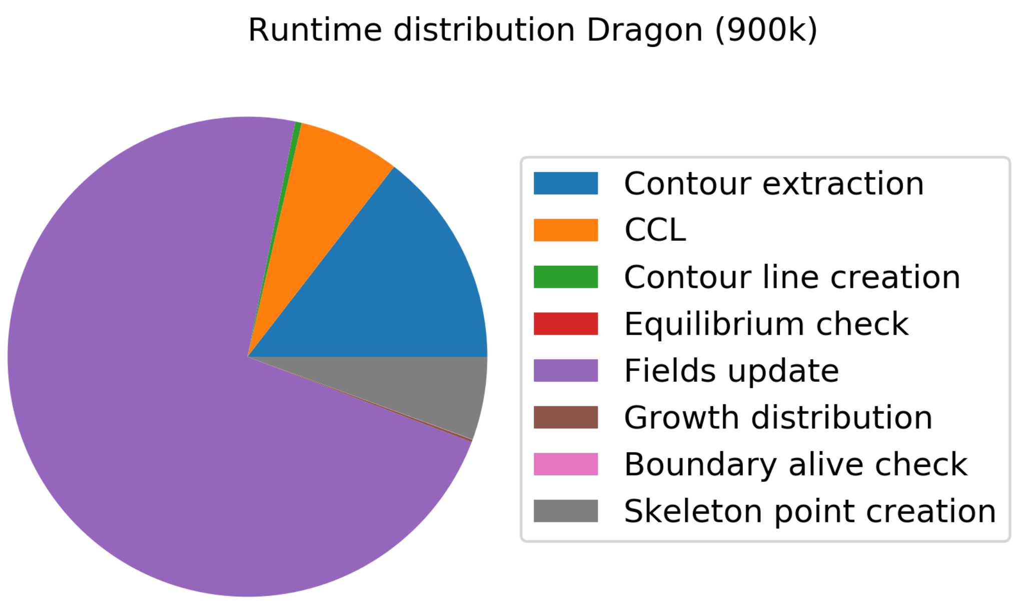

Figure 5.11: The contribution of each step to the overall runtime during the whole simulation of the dragon mesh.

5.3 Memory consumption

The reported numbers in Table 5.2, do contain the complete memory usage on the GPU, including Layered fields and our method. The system memory usage stays below the usage on the GPU and is therefore not reported. As expected, the memory consumption grows linearly with the number of triangles. Our method only creates the edge list for contour extraction, which scales linearly with the number of triangles. The capacity needed for other helper structures depends on the contour line size, which stays well below the number of faces of the whole mesh.

Mesh	Memory consumption	
	Total	Per triangle
Hand Pierre (1.5M)	424MB	0.26kB
Male (1.4M)	416MB	0.29kB
Dragon (900K)	294MB	0.32kB
Hand arc (800K)	262MB	0.32kB
Deer (100K)	86MB	0.86kB
Human (50k)	72MB	1.44kB

Table 5.2: Memory consumption on the GPU for various meshes. This includes memory allocated by the Layered fields implementation and our method.

6 Conclusion

To conclude, in this thesis, we developed Natural Reeb graphs, which avoid difficulties existing Reeb graph methods are facing. By replacing the explicit mapping function of traditional Reeb graphs, Natural Reeb graphs can be used in an automatic application setting because no user input is required. Natural Reeb graphs include all important aspects like branchings or holes of a surface to get a high-level explanation of the 3D shape.

As an application, a skeleton extraction method is presented, which is competitive compared to existing methods. Skeletons produced by our method are transformation invariant as well as centered, which are two main features required by further applications of mesh skeletons. The GPU friendly design of both, the Natural Reeb graph algorithm, and the skeleton extraction method, results in fast execution on the GPU. Even topological complex models like the Chinese dragon with around 900k triangles can be handled in about 20 seconds on a single consumer GPU. Furthermore, reverse growing is an optimization enabled by the implicit mapping function of the Natural Reeb graphs. It improves the skeleton in branching areas, which are difficult to handle for existing Reeb graph-based methods. Reverse growing redefines the mapping function for those areas on-line, which provides additional information to improve the skeleton.

Bibliography

[AH12] Waleed Abbas and A. Hamza. "Reeb graph path dissimilarity for 3D object matching and retrieval." In: *The Visual Computer* 28 (Mar. 2012), pp. 305–318. DOI: 10.1007/s00371-011-0640-5 (cit. on p. 1).

[AN15] A. Acharya and V. Natarajan. "A parallel and memory efficient algorithm for constructing the contour tree." In: *2015 IEEE Pacific Visualization Symposium (PacificVis)*. Apr. 2015, pp. 271–278. DOI: 10.1109/PACIFICVIS.2015.7156387 (cit. on p. 7).

[Au+08] Oscar Kin-Chung Au et al. "Skeleton Extraction by Mesh Contraction." In: *ACM Trans. Graph.* 27.3 (Aug. 2008), 44:1–44:10. ISSN: 0730-0301. DOI: 10.1145/1360612.1360643. URL: http://doi.acm.org/10.1145/1360612.1360643 (cit. on pp. 8, 43).

[Bia+03] Silvia Biasotti et al. "An overview on properties and efficacy of topological skeletons in Shape Modelling." In: June 2003, pp. 245–254. ISBN: 0-7695-1909-1. DOI: 10.1109/SMI.2003.1199624 (cit. on p. 3).

[BPS97] C. L. Bajaj, V. Pascucci, and D. R. Schikore. "The contour spectrum." In: *Proceedings. Visualization '97 (Cat. No. 97CB36155)*. Oct. 1997, pp. 167–173. DOI: 10.1109/VISUAL.1997.663875 (cit. on pp. 2, 7, 24, 43).

[CSM07] Nicu D. Cornea, Deborah Silver, and Patrick Min. "Curve-Skeleton Properties, Applications, and Algorithms." In: *IEEE Transactions on Visualization and Computer Graphics* 13.3 (May 2007), pp. 530–548. ISSN: 1077-2626. DOI: 10.1109/TVCG.2007.1002. URL: https://doi.org/10.1109/TVCG.2007.1002 (cit. on p. 8).

[han19] hand. *"3D Hand scan" by artec3d.com is licensed under CC BY 3.0*. 2019. URL: https://www.artec3d.com/3d-models/hand (visited on 12/31/2019) (cit. on p. 44).

[Hil+01] Masaki Hilaga et al. "Topology matching for fully automatic similarity estimation of 3D shapes." In: Jan. 2001, pp. 203–212. DOI: 10.1145/383259.383282 (cit. on p. 1).

[Lab96] Stanford University Computer Graphics Laboratory. *3D-scan of a chinese dragon*. 1996. URL: http://graphics.stanford.edu/data/3Dscanrep/ (visited on 12/31/2019) (cit. on p. 36).

[MLM18] I. Manolas, A. S. Lalos, and K. Moustakas. "Parallel 3D Skeleton Extraction Using Mesh Segmentation." In: *2018 International Conference on Cyberworlds (CW)*. Oct. 2018, pp. 172–175. DOI: 10.1109/CW.2018.00041 (cit. on p. 8).

[Pas+07] Valerio Pascucci et al. "Robust On-line Computation of Reeb Graphs: Simplicity and Speed." In: *ACM SIGGRAPH 2007 Papers*. SIGGRAPH '07. San Diego, California: ACM, 2007. DOI: 10.1145/1275808.1276449. URL: http://doi.acm.org/10.1145/1275808.1276449 (cit. on p. 7).

[REE46] G. REEB. "Sur les points singuliers d'une forme de Pfaff completement integrable ou d'une fonction numerique [On the Singular Points of a Completely Integrable Pfaff Form or of a Numerical Function]." In: *Comptes Rendus Acad. Sciences Paris* 222 (1946), pp. 847–849. URL: https://ci.nii.ac.jp/naid/10024635920/en/ (cit. on p. 1).

[Sha+07] Andrei Sharf et al. "On-the-fly Curve-skeleton Computation for 3D Shapes." In: *Comput. Graph. Forum* 26 (Sept. 2007), pp. 323–328. DOI: 10.1111/j.1467-8659.2007.01054.x (cit. on p. 8).

[SKK91] Y. Shinagawa, T. L. Kunii, and Y. L. Kergosien. "Surface coding based on Morse theory." In: *IEEE Computer Graphics and Applications* 11.5 (Sept. 1991), pp. 66–78. ISSN: 1558-1756. DOI: 10.1109/38.90568 (cit. on p. 7).

[Tag+12] Andrea Tagliasacchi et al. "Mean Curvature Skeletons." In: *Computer Graphics Forum* 31.5 (2012), pp. 1735–1744. DOI: 10.1111/j.1467-8659.2012.03178.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2012.03178.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2012.03178.x (cit. on p. 8).

[WL08] Y. Wang and T. Lee. "Curve-Skeleton Extraction Using Iterative Least Squares Optimization." In: *IEEE Transactions on Visualization and Computer Graphics* 14.4 (July 2008), pp. 926–936. ISSN: 2160-9306. DOI: 10.1109/TVCG.2008.38 (cit. on p. 8).

[WXS06] Naoufel Werghi, Yijun Xiao, and Jan Siebert. "A functional-based segmentation of human body scans in arbitrary postures." In: *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* 36 (Mar. 2006), pp. 153–165. DOI: 10.1109/TSMCB.2005.854503 (cit. on p. 1).