Natural Reeb Graphs

Alexander Weinrauch*
Supervised by: Markus Steinberger†

University of Technology
Graz / Austria

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. The proposed method to generate Natural Reeb graphs is designed to be executed efficiently on the graphics processing unit (GPU).

Keywords: Reeb Graph, Shape analysis, GPU

1 Introduction

The Reeb graph [8] represents the topological skeleton of an n-dimensional object and provids 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 [2, 5], surface understanding [5] or shape segmentation [14], data skeletonization [4], topological morphing [6].

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.

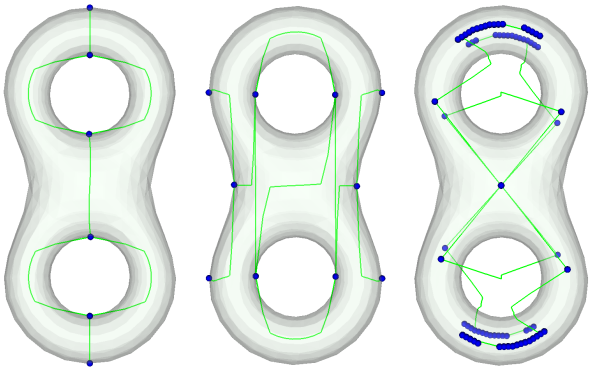

Figure 1: Three Reeb graphs defined by the y-axis (left), x-axis (middle) and z-axis (right) as mapping function.

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 does not produce the Reeb graph based on a user-defined mapping function f, but it is motivated by a recently published method called Layered Fields [15], which simulates a natural diffusion process.

The scalar mapping function f on the topology is implicitly extracted from the natural diffusion process on the surface of a triangulated mesh, therefore, we deduced the name Natural Reeb graphs. The level sets are extracted from the energy states of each vertex during the simulation. 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 three examples with different mapping functions are shown. The Reeb graph shown in the left image of Figure 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

*alexander.weinrauch@gmail.com
†steinberger@icg.tugraz.at

not help to get a global view. The Reeb graph will, therefore, have a low semantic value for further applied algorithms. Shape matching based on the Reeb Graph would be a good example for the low sematic value of the thrid example compared to the example shown on the left. As described by Bajaj et al. [13], 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, which is a key feature required in skeleton creation and shape matching based on the Reeb graph [13].

2 Background and Related Work

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

2.1 Layered Fields

The goal of the Layered Fields paper [15] 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, 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.

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.

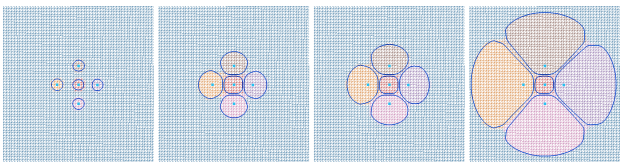

Figure 2: Growth of 3 layers on a plane. The light blue dots represent the start seeds and the blue lines the sharp boundary used to create Natural Reeb Graphs.

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 [3]. The first algorithm to correctly compute the Reeb graph dates back to 1991 [10] 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. [7] 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 [13] 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.

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.

$$L_c(f) = \{(x_1, ..., x_n) | (f(x_1, ..., x_n) = c)\} \quad (1)$$

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.

$$f(t,v,i) = f_i(t,v) = e, \{e \in \mathbb{R} \mid 0 \leq e \leq 1\} \qquad (2)$$

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.

The tracking is performed 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 embedding the Natural Reeb graph into the mesh. For the calculation of the Natural Reeb graph alone, 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 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 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 4 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.

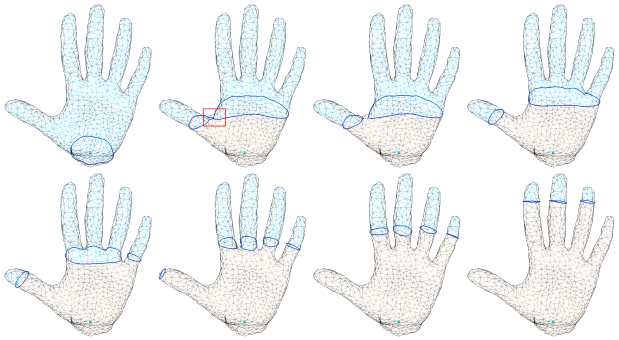

Figure 3: Sharp boundaries (dark blue lines) traced over the evolution of one cell. All brown vertices have higher energy than the threshold c, light blue vertices have less energy than c.

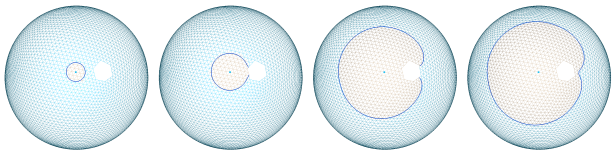

Figure 4: Shows the behaviour of a sharp boundary on a sphere with a hole in it. Note that the circle degenerates to a path while the hole is part of the sharp boundary.

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 a 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 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 performance. To avoid these expensive computations during all iterations, we separate the advancing fronts into different diffusion layers.

3.3 Layer Splitting

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 5 with the evolution shown in Figure 3, 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.

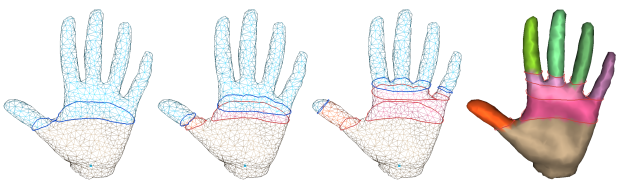

Figure 5: The evolution of a cell with layer splitting enabled. The face colors represent the layer index. The red contour lines show iterations where more than one contour line was present on a single layer.

3.4 Vanishing Components

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 connected component counting and is therefore available without a runtime cost.

3.5 Merging Components

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 natural diffusion process has completed. 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.

3.6 Method summary

Figure 6 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 removed, and also two new nodes were added to the Reeb

graph due to merging regions. 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 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.

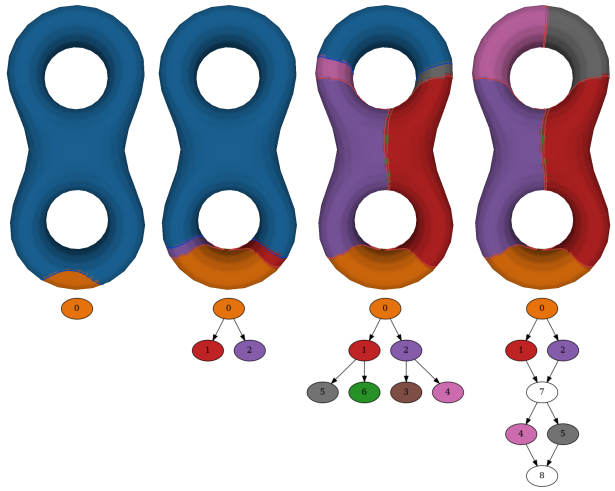

Figure 6: System state at different simulation stages combined with the Reeb graph. Layer seven and eight are caused by the detection of merging layers.

4 Implementation

For simplicity and readability, the discussion in this section does only describe the method for one layer. To support multiple layers, which is mandatory for the algorithm, one can use a serial or a parallel approach. In the serial approach, each layer is processed one after another in a loop. In the parallel approach, all active layers are processed at the same time. The parallel approach induces a higher memory usage but may improve the utilization of more compute units on a GPU. Especially if many small contour lines are present on the layers, the graph during connected component labeling may not be large enough to utilize all multiprocessors available on the GPU. A pseudo code of the complete Natural Reeb Graph extraction algorithm is provided in Algorithm 1.

4.1 Contour edges

The first step in our method is to create a list with all faces, which are part of the contour lines on a specific layer of the given system state. As precomputation, a list of all edges of the mesh containing the start and end vertices and the two adjacent faces is created and transferred to the GPU. To flag all contour triangles, an array (neighbor array) with n items is created, where n is the number of vertices of the unstructured grid. Each item can store two indices of neighbor triangles, which are also part of the contour line. Storing only up to two neighbors assumes that each triangle on the contour has no more than two neighbor faces, which are also part of the contour. The assumption holds for manifold meshes with the addition of borders, because for this kind of meshes, the contour lines form circles or paths where no vertex is used more than once. Initially, each neighbor index is set to zero, which indicates that no neighbor lies on the contour.

For each edge, the energy value of the two vertices is read from the system matrix. If precisely one vertex has a higher value than the given threshold c, both faces of the edge are part of the contour. In that event, the index of one face is written into the neighbor list entry of the other face and vice versa. To avoid race conditions during the writing process of the neighbour index, an atomic compare and swap (CAS) is used, where the new value is only written if the old value was zero. In case the CAS operation on the first item did not succeed, a regular write operation can be performed into the second item.

After the iteration over all edges, every triangle which does have a neighbour entry set in the neighbour list is part of the contour. To generate a dense list of all contour triangles, a linear array is allocated. The length of the array is equal to the number of contour triangles. The contour triangle indices are then written into the dense array using an atomic counter to deduce the index into the dense list.

It is important to note that all helper structures, such as the neighbor list or the contour triangle list, can be reused in future iterations. This keeps the number of costly allocations on the GPU to a minimum. Furthermore, the capacity of most helper structures is determined by the number of triangles on the contour, which stays relatively low in comparison to the number of vertices. To even further minimize the number of allocations during runtime, we increase the size of the helper structure by additional 20% in the case we need to enlarge them. This lowers the number of allocations if the contour triangle count does raise again in future iterations.

4.2 Component counting-labeling

Connected component labeling is a well-researched problem both on the CPU and GPU. However, most research for algorithms suited for the GPU has focused on structured grids like a 2D image. Algorithms for the CPU for unstructured grids are mostly based on breadth-first search (BFS) or depth-first search (DFS), which are highly serial approaches that do not perform well on the GPU. One algorithm designed for the GPU was published by J. Soman, K. Kothapalli, and P. Narayanan [11], which we used for the implementation of our method. The algorithm is based on previous work [9] for the CPU, but by reducing the irregular memory accesses, it improves the runtime on the GPU.

4.3 Layer splitting

The goal of this step is to create a new layer that will start at the location of exactly one connected component. This step requires the full connected component labeling.

Adding additional layers during runtime was not covered by Zayer et al. [15]. However, it turns out that increasing the number of rows in the system matrix is sufficient to create a new layer. Since the system matrix is stored in the CSC representation, the linear data array does not need any modification, only a single integer representing the number of rows needs to be updated. The fields contain a utility layer as the last row of the system matrix. This layer has to be moved up by n rows to still be the last layer of the system matrix. This is done by iterating over the whole linear data array and just incrementing the row index for elements of the old utility row index.

4.3.1 Transfer discrete boundary

The next step is to set the starting vertices for the new layers. Each new layer represents one detected connected component. The vertices of the triangles forming the component are the starting points of the layer. Despite the simplicity of changing the dimensionality of the system matrix, inserting new data into a CSC matrix is an expensive operation. It involves moving all data tied to a higher column index or tied to the same column but a higher row index. All contour line vertices must have energy on the parent layer, or otherwise, they would not be part of the contour line. This implies that an entry for the parent layer index in the system matrix has to be present for contour vertices. Our implementation avoids insertion by shifting the present energy values on the parent layer to the child layer index. The correct new row index depends on the membership to a connected component. The connected component labeling provides this information.

This procedure only moved the sharp boundary defined by the constant c to the child layers. There will still be energy smaller than c on the parent layer behind the sharp boundary. Those energy levels limit the growth of the child layers and therefore have to be removed.

4.3.2 Transfer smooth boundary

After splitting a layer, contour tracking cannot be performed for new layers, because the boundary will be unstable. To reduce the time where tracking cannot be performed, the energy levels smaller than c of the parent layer are transferred to the child layers instead of setting them to zero. The simple change in row index can be performed again to avoid insertion, however, selecting the right new layer index is not trivial. Vertices with smaller energy levels than c were part of one smooth boundary forming the connected component. The contour lines are build by using the sharp boundary defined by the constant c, so there is no mapping between those vertices of the smooth boundary and the connected components. Therefore, a controlled flooding algorithm [12] is used to select the correct child layer for the vertices of the smooth boundary. It starts from the vertices forming one connected component and looks at all neighbor vertices. If a neighbor vertex has an energy level smaller than c, it is considered as part of the same connected component and is transferred to the new layer of this connected component. Transferred vertices are the start vertices of the next iteration. Vertices with the energy level of zero are not part of the boundary and are ignored. Those signal the end of the smooth boundary. This is repeated until no more start vertices are marked in the last iteration.

4.3.3 Removal of border between parent and children

After splitting, the child layer has two smooth boundaries. One boundary will advance further on the mesh, and the second one will stay at the initial start vertices, representing the border between the parent and the child layer. This goes against the idea of having only one boundary/connected component on a layer. This is solved by modifying the Laplacian matrix to prohibit the diffusion of energy over faces connecting the parent layer with the child layers. By erasing all values corresponding to the contour triangles and vertices, which have a higher value than c, the diffusion is stopped between those layers. Additionally, the edges going across these regions are removed from the edge list used to extract the contour lines by using the same controlled flooding algorithm. This results in not detecting the sharp boundary between the child and parent 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;
        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

Mesh(Triangles)	Total	Fields update	Contour extraction	CCL
Hand Pierre (1.5M)	23.183	17.332	32.87	1.232
Male (1.4M)	40.604	28.657	7.222	2.434
Dragon (900K)	21.973	15.319	3.075	1.445
Hand arc [1] (800K)	15.147	11.009	2.086	1.170
Deer (100K)	3.079	1.240	0.546	0.789
Human (50k)	1.373	0.384	0.262	0.467

Table 1: Runtime in seconds for different sized meshes. Only important steps of the method, in terms of runtime, are listed.

5 Results

To illustrate the Natural Reeb graphs more clearly, they are embedded into the mesh by sampling the location for a critical point as the barycenter of the respective contour line. Additionally to the figures presented in this section, a video[1] demonstrating this process is also provided. Figure 7 shows such embeddings for different meshes. The start vertex was selected randomly. The effect of different starting points is visualized in Figure 8. In each example in Figure 8 the start vertex is highlighted with a red sphere. These examples show that the Natural Reeb graph is only influenced in the local neighborhood of the start vertex. For parts of the mesh further away, the Natural Reeb graph looks exactly the same as for different starting starting vertices.

[1] https://youtu.be/tsX8iV1RNEU

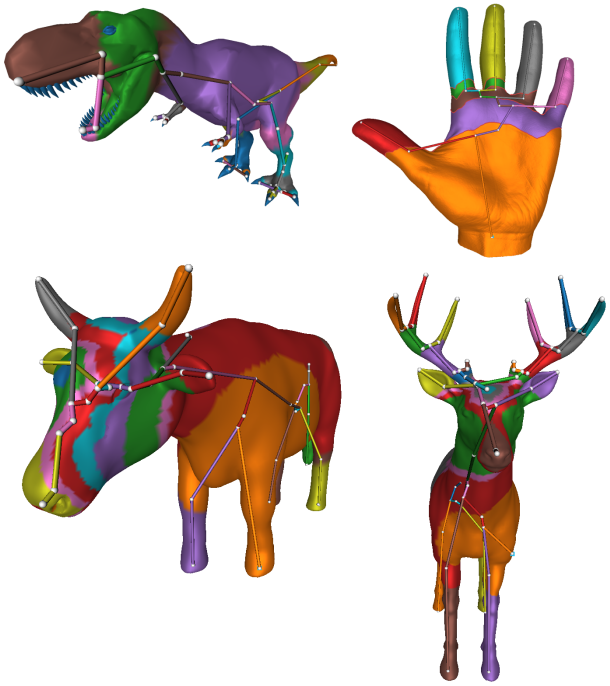

Figure 7: Natural Reeb graphs embedded into the mesh. For each splitting nodes are inserted for the splitting layer and all child layers.

Figure 8: Natural Reeb graphs embedded into the mesh. Each example has a different start vertex highlighted in red.

5.1 Runtime

Table 1 shows the total runtime and detailed information about each runtime intesive step for various meshes. An important observation is that the Natural Reeb graph extraction 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.

6 Memory consumption

The reported numbers in Table 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 2: Memory consumption on the GPU for various meshes. This includes memory allocated by the Layered fields implementation and our method.

7 Conclusion

To conclude, in this paper we introduce Natural Reeb graphs, which avoid difficulties existing Reeb graph based applications 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. Different start vertices only influence the local neighborhood. Therefore, Natural Reeb graphs provide a meaningful representation of the mesh independent of the exact start vertex. For future work we want to further explore the indepence of Natural Reeb Graphs to their starting position. Combining multiple Natural Reeb Graphs from different starting regions to remove the areas influenced by the starting positions might improve our results.

This combination might be especially useful for shape matching, because the produced shape description would be invariant to the orientation and trasformation of the mesh.

References

[1] "3d hand scan" by artec3d.com, licensed under cc by 3.0, 2019.

[2] Waleed Abbas and A. Hamza. Reeb graph path dissimilarity for 3d object matching and retrieval. *The Visual Computer*, 28:305–318, 03 2012.

[3] A. Acharya and V. Natarajan. A parallel and memory efficient algorithm for constructing the contour tree. In *2015 IEEE Pacific Visualization Symposium*, pages 271–278, April 2015.

[4] Xiaoyin Ge, Issam I. Safa, Mikhail Belkin, and Yusu Wang. Data skeletonization via reeb graphs. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 837–845. Curran Associates, Inc., 2011.

[5] Masaki Hilaga, Yoshihisa Shinagawa, Taku Komura, and Tosiyasu Kunii. Topology matching for fully automatic similarity estimation of 3d shapes. pages 203–212, 01 2001.

[6] P. Kanonchayos, T. Nishita, S. Yoshihisa, and T. L. Kunii. Topological morphing using reeb graphs. In *Proceedings of the First International Symposium on Cyber Worlds (CW'02)*, CW '02, page 0465, USA, 2002. IEEE Computer Society.

[7] Valerio Pascucci, Giorgio Scorzelli, Peer-Timo Bremer, and Ajith Mascarenhas. Robust on-line computation of reeb graphs: Simplicity and speed. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, 2007.

[8] Georges 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]. *Comptes Rendus Acad. Sciences Paris*, 222:847–849, 1946.

[9] Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

[10] Y. Shinagawa, T. L. Kunii, and Y. L. Kergosien. Surface coding based on morse theory. *IEEE Computer Graphics and Applications*, 11(5):66–78, Sep. 1991.

[11] Jyothish Soman, Kishore Kothapalli, and P J Narayanan. Some gpu algorithms for graph connected components and spanning tree. *Parallel Processing Letters*, 20(04):325–339, 2010.

[12] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1981.

[13] J. Tierny, J. Vandeborre, and M. Daoudi. Invariant high level reeb graphs of 3d polygonal meshes. In *Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*, pages 105–112, 2006.

[14] Naoufel Werghi, Yijun Xiao, and Jan Siebert. A functional-based segmentation of human body scans in arbitrary postures. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 36:153 – 165, 03 2006.

[15] Rhaleb Zayer, Daniel Mlakar, Markus Steinberger, and Hans-Peter Seidel. Layered fields for natural tessellations on surfaces. *ACM Trans. Graph.*, 37(6), December 2018.