Robust On-line Computation of Reeb Graphs: Simplicity and Speed*

Valerio Pascucci
CASC - LLNL / CS - UC DAVIS
pascucci@acm.org

Giorgio Scorzelli
CASC - LLNL
scrgiorgio@cogesic.it

Peer-Timo Bremer
CASC - LLNL
ptbremer@acm.org

Ajith Mascarenhas
CASC - LLNL
mascarenhas1@llnl.gov

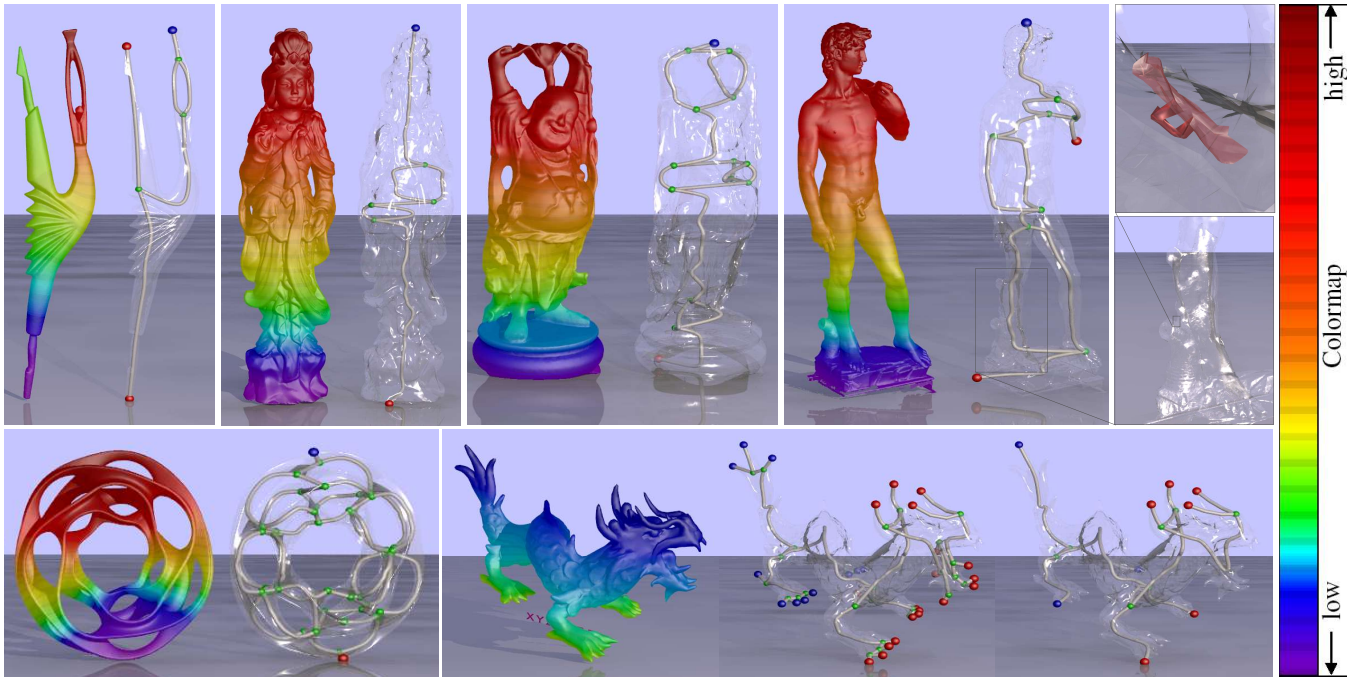


Figure 1: (Top row) Simplified Reeb graphs of the Dancer, Malaysian Goddess, Happy Buddha; and David together with two close-ups showing a tiny tunnel at the base of David's leg. The pseudo-colored surfaces show the function used for computing the Reeb graph. The transparent models show the structure of the Reeb graph and its embedding. (Bottom row) The Heptoroid model and two levels of resolution for the Reeb graph of the Asian Dragon model.

Abstract

Reeb graphs are a fundamental data structure for understanding and representing the topology of shapes. They are used in computer graphics, solid modeling, and visualization for applications ranging from the computation of similarities and finding defects in complex models to the automatic selection of visualization parameters.

We introduce an *on-line* algorithm that reads a stream of elements (vertices, triangles, tetrahedra, etc.) and continuously maintains the Reeb graph of all elements already read. The algorithm is *robust* in handling non-manifold meshes and *general* in its applicability to input models of any dimension.

Optionally, we construct a skeleton-like embedding of the Reeb graph, and/or remove topological noise to reduce the output size. For interactive multi-resolution navigation we also build a hierarchical data structure which allows real-time extraction of approximated Reeb graphs containing all topological features above a given error threshold.

Our extensive experiments show both high performance and practical linear scalability for meshes ranging from thousands to hundreds of millions of triangles. We apply our algorithm to the largest, most general, triangulated surfaces available to us, including 3D, 4D and 5D simplicial meshes. To demonstrate one important application we use Reeb graphs to find and highlight topological defects in meshes, including some widely believed to be "clean."

*For more information about the project see:
http://pascucci.org/research/topology/reeb-graph/

1 Introduction

The Reeb graph [Reeb 1946] is a fundamental data structure that encodes the topology of a shape. It is obtained by contracting to a point the connected components of the level-sets (also called contours) of a function defined on a mesh. The Reeb graph has been used extensively in a wide range of applications such as shape matching [Hilaga et al. 2001] and encoding [Shinagawa et al. 1991; Lazarus and Verroust 1999; Takahashi et al. 1997], compression [Biasotti et al. 2000], surface parameterization [Steiner and Fischer 2002], and iso-surface remeshing [Wood et al. 2000] and simplification [Wood et al. 2004]. Reeb graphs can determine whether a surface has been reconstructed correctly, indicate problem areas, and can be used to encode and animate a model. Topological concepts such as the Reeb graph are especially useful in processing massive models like those generated by high-resolution laser range scans. However, we are aware of only two gener-

ically applicable algorithms [Shinagawa and Kunii 1991; Cole-McLaughlin et al. 2003] to compute Reeb graphs neither of which is able to handle even moderately size surfaces. Both algorithms require a global sorting step, expensive temporary data structures, and most importantly demand the input to be a manifold surface. None of these requirements can be fulfilled for models such as the David shown in Figure 1(top-right).

This paper presents a new algorithm to compute Reeb Graphs in an on-line fashion. The algorithm is fast, scales linearly in practice, and handles virtually any simplicial mesh without restrictions on dimension or complexity. When available, the algorithm takes advantage of coherency in the input (even one not as perfect as the one constructed in [Isenburg and Lindstrom 2005]), and computes Reeb graphs of massive models using a small memory footprint. Furthermore, our algorithm naturally handles non-manifold surfaces. As we prove in Section 3 this property is sufficient to compute Reeb graphs for meshes of any dimension and we show three, four, and five-dimensional examples.

During the on-line computation we can create an embedding of the Reeb graph similar to the one described in [Lazarus and Verroust 1999] which provides a medial-axis-type skeleton representation of the surface. The skeleton can be used to model or animate the surface as well as to display the Reeb graph itself. If necessary, we also remove topological noise on-the-fly to avoid creating unnecessarily large graphs. Finally, we process the Reeb graph to create a progressive layout that provides real-time access to the graph in a coarse to fine manner. This allows the handling of massive graphs on moderate hardware and makes them available for various applications such as topological simplification [Wood et al. 2004].

Prior work. Reeb graphs, and especially their loop-free specialization called contour trees, are popular in computer graphics and visualization. In [1991] Shinagawa et al. first introduced Reeb graphs to the computer graphics community using them to construct surfaces. In a later related paper Takahashi et al. [Takahashi et al. 1997] enhance the Reeb graph with certain flow curves on the surface to create a control net used to encode smooth surfaces. Biasotti and others [Biasotti 2001; Attene et al. 2001; Biasotti et al. 2000] reconstruct, compress, and analyze surfaces based on related concepts. Along a slightly different line, Hilaga et al. [2001] use Reeb graphs of an approximated all-pairs-shortest-distance function on a surface to find shapes in a data base. A more extensive collection of similar ideas can be found in [Funkhouser and Kazhdan 2004]. Steiner and Fisher [2002; 2001] are interested in cutting an arbitrary surface into a disk and to this end store selected level-sets along edges of the Reeb graph. A handle in the surface is opened by cutting along the level-set stored in the loop of the Reeb graph corresponding to the handle. Ni et al.[2004] solve the same problem by computing a "fair" Morse function on the surface and use its Morse complex to define the cuts.

To compute the Reeb graph all papers mentioned so far use variants of Shinagawa and Kunii's [1991] original algorithm. However, instead of computing the correct Reeb graph in $O(n^2)$ time (where n is the number of edges in the triangulation) they choose a sample based algorithm running in $O(n \log n)$ which may return incorrect results. The only correct $O(n \log n)$ algorithm we are aware of is due to Cole-McLaughlin et al. [2003]. They sweep the surface in order of function value maintaining fronts using dynamic balanced search trees. As mentioned above both algorithms require manifold surfaces and neither of them is able to handle modern size data sets. Wood and others [Wood et al. 2000; Wood et al. 2004] compute Reeb graphs of iso-surfaces of volumetric functions using concepts similar to those of [Shinagawa and Kunii 1991]. Their algorithm is streaming in the sense that they only need to keep a single slice of the volume in memory. Besides being limited to iso-surfaces their algorithm implicitly uses the sweep direction as function value as thus is restricted to one of the three coordinate functions.

Contour trees are widely used in visualization and even a limited survey of related work is beyond the scope of this paper. Standard algorithms to compute contour trees in any dimension can be found in [Carr et al. 2003] and [Chiang et al. 2005] and an extension which augments the tree with information about the genus of the level-sets in [Pascucci and Cole-McLaughlin 2003]. Some of the more popular applications include the acceleration of level-set extraction [van Kreveld et al. 1997], simplification of level-sets [Carr et al. 2004], and the automatic design of transfer functions [Weber et al. 2002].

Contributions. One fundamental novelty of our algorithm is the ability to achieve practical efficiency while eliminating the need to sort the mesh. In particular, this paper provides contributions in four areas. We:

(i) Introduce a robust algorithm to compute Reeb graphs for simplicial meshes using a stream of local updates. The algorithm naturally takes advantage of coherency in the input and makes effective use of auxiliary information such as knowing that the input is a manifold or explicit vertex finalization;

(ii) Compute a simplification and a good embedding of the Reeb graph with a small performance overhead of less than 5% of the overall running time;

(iii) Prove and demonstrate how the same algorithm can be used to compute Reeb Graphs of tetrahedral and higher dimensional meshes; and

(iv) Provide extensive empirical evidence showing high performance (processing 500K to 800K triangles per second) and practical linear scalability up to the largest models available to us. In particular we compute the Reeb graph of the St. Matthew model, with 372M triangles, in 486 seconds using less than 20MB of main memory.

2 Reeb Graphs

Given a simplicial mesh M with a piecewise linear (PL) function F sampled at its vertices, the *level-set* at a value s is defined as the set of points in M with function value equal to s. We analyze the evolution of the connected components of the level-sets of F called *contours*.

The *Reeb graph* `RG` of F is obtained by contracting the contours of F to points. An example of this construction is shown in Figure 2 for the triple torus model with function F equal to the z coordinate of its vertices.

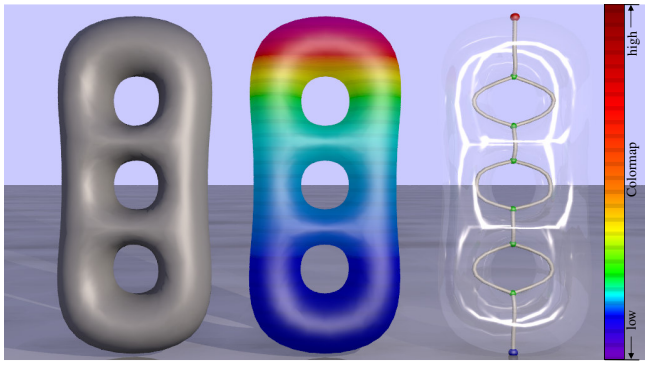


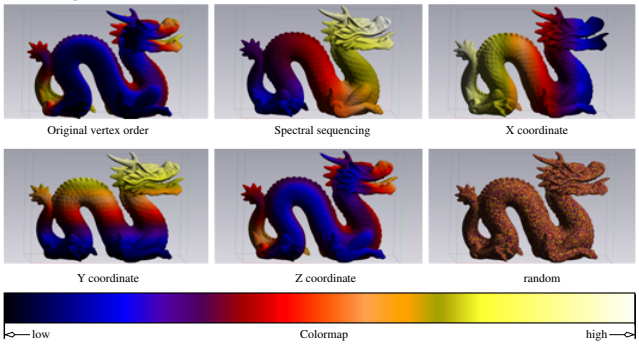
Figure 2: Reeb graph of the height function on the triple torus. The three tunnels of the model are mapped to three loops in the graph.

Points on the Reeb graph that correspond to contours passing through critical points of F (maxima, minima, and saddles) are called *nodes*. The rest of the Reeb graph consists of *arcs* connecting the nodes. Since, contours change topology only at critical points an arc represents a family of contours that do not change topology. For a PL function the critical points lie on the vertices of M and

we can associate nodes with vertices of M. We call a monotonic sequence of arcs in RG a *path*.

After describing the input mesh, the input function, and the data structures to represent the Reeb graph, we will describe a simple, robust algorithm that computes Reeb Graphs for triangulated (not necessarily manifold) surfaces. Then, we describe modifications to this algorithm to (i) increase performance, (ii) compute an embedding of the Reeb graph, and (iii) to simplify Reeb graphs. Finally, we show in Section 3 that the identical algorithm also computes Reeb graphs of functions defined over simplicial complexes of arbitrary dimensions.

Input Mesh. We assume the input is a sequence of vertices and triangles; a triangle refers to its vertices by a unique index given by their order of appearance. Triangles and vertices do not need to be in any particular order, except that all vertices of a triangle must appear before the triangle. We handle the Alias/Wavefront OBJ format, the Stanford PLY format, and the streaming mesh format by Isenburg and Lindstrom [2005], in addition to other formats.



Input Function. As discussed above, F is defined using values at vertices of the input mesh. These are provided either in an additional file, given implicitly (e.g. one of the vertex coordinates), or computed at run-time. The images above show several functions as a pseudo-coloring on the dragon mesh. Naturally, the choice of a function is dependent on the particular target application.

Data Structure. We maintain two tightly coupled data-structures: The input mesh M and the Reeb graph RG. We use a standard graph data structure to represent the Reeb graph. With each node of RG we maintain the index of its corresponding vertex. With each arc of RG we maintain a list of pointers to edges of M that intersect the contours represented by the arc. With each edge $e \in M$ we maintain a pointer to the highest (in F) arc in RG that has a pointer to e. An arc of RG represents a family of contours that do not change topology, each of which intersects a set of edges in M. An edge of M intersects a family of contours that may span several arcs in RG. In particular, an edge corresponds to a path in RG. In Figure 3(a), for example, we see arc a_0 has pointers to edges e_1 and e_4, because a_0 represents contours that intersect edges e_1 and e_4. Edge e_4 is pointed to by arcs a_0 and a_2, because it intersects contours that are represented by both the arcs. Edge e_4 stores a pointer to arc a_0 because it is the highest arc with a pointer to e_4. To ease traversal between M and RG we store extra information with each edge to jump to the entry in an arc's list that refers to that edge. In Figure 3(a), the tuple $(a_0, 1)$ stored with edge e_4 tells us that the entry in a_0's list at index 1 points back to e_4. Similarly, to traverse a path in RG corresponding to edge e we store a pointer with each reference to e to the reference in the next arc in the path. In Figure 3(a), we show these as downward pointing arrows in the Reeb graph.

We will use the following operations to modify the Reeb graph:
- CREATENODE**(**i, w**)** Add a new node with index i, and function value w to RG, see Figure 3(a).

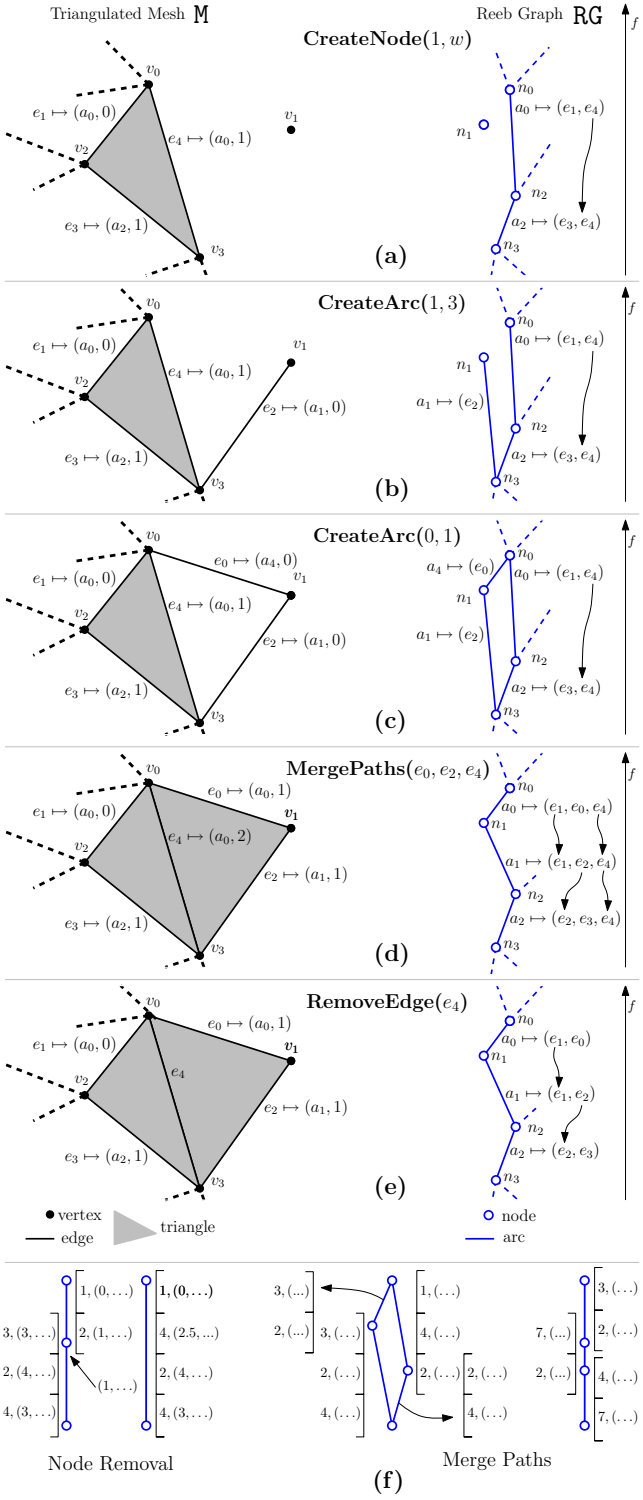


Figure 3: Incremental update of the Reeb graph after inserting triangle (v_0, v_1, v_3). (a) When inserting vertex v_1 we insert a new node into RG (Because v_0 and v_3 are part of an existing triangle their nodes already exist). (b-c) When inserting the edges of the triangle we insert corresponding arcs in RG. (d) When inserting the interior of the triangle we merge the path a_4, a_1 with the path a_0, a_2 to form the path a_0, a_1, a_2. (e) We optimize the data structure by removing any reference to the manifold edge e_4. (f) Update of the buckets with embedding information when merging arcs of RG.

```
1  Function ComputeReebGraph(InputFile):
2     ForEach element in InputFile Do:
3        If (element is vertex v)
4           CreateNode(v);
5        ElseIf (element is triangle t)
6           ForEach edge e in t Do:
7              If (e is new)
8                 CreateArc(e);
9              EndIf
10          EndFor
11          // Call e0,e1,e2 the edges of t, with e0,e1
12          // sharing the maximum and e0,e2 the minimum.
13          MergePaths(e0,e1,e2);
14          ForEach edge e in t Do:
15             If (all vertices in e are finalized)
16                RemoveEdge(e);
17             EndIf
18          EndFor
19       EndIf
20    EndFor
```

```
1  Function MergePaths(Edges e0,e1,e2):
2     a0 = Arc(e0); a1 = Arc(e1); a2 = Arc(e2);
3     GlueByMergeSorting(a0,a1,e0,e1);
4     GlueByMergeSorting(a0,a2,e0,e2);
```

```
1  Function GlueByMergeSorting(Arcs a0,a1, Edges e0,e1):
2     While (a0 and a1 are valid arcs) Do:
3        n0=BottomNode(a0);
4        n1=BottomNode(a1);
5        If (F(n0) > F(n1))
6           MergeArcs(a0,a1);
7        Else
8           MergeArcs(a1,a0);
9        EndIf
10       a0 = NextArcMappedToEdge(a0,e0);
11       a1 = NextArcMappedToEdge(a1,e1);
12    EndWhile
```

Table 1: Pseudocode for the core functions of the Reeb Graph computation algorithm. The entry point is the function ComputeReebGraph().

- CREATEARC(i, j) Add a new arc connecting node i and node j to RG, see Figure 3(b-c).
- MERGEPATHS$(e_\alpha, e_\beta, e_\gamma)$ Merge paths e_α, e_β ,and e_γ in RG, see Figure 3(c-d).

The MERGEPATHS operation will be used to remove loops in the Reeb graph, as will be described shortly.

General Algorithm. Our algorithm proceeds by reading vertices and triangles processing them in the order of appearance. Initially, RG is empty and each time we read a new vertex or triangle we make an update that maintains the correct Reeb graph of all simplices seen so far, possibly with addition of degree-two nodes (removed by the optimizations described below). Figure 4 shows this idea for triangles that create/connect connected components or create/destroy loops. The pseudocode for the main functions of our algorithm is provided in Table 1.

For each new vertex, we invoke CREATENODE to create a new node in the Reeb graph. For each triangle, we first invoke CREATEARC for each of its edges that has not already been seen. Inserting the interior of a triangle could connect disjoint contours inducing a merging of the paths in RG that correspond to the edges of the triangle; we invoke the MERGEPATHS operation to process the interior of the triangle.

Referring to Figure 3(a-c), we see that (starting with a correct Reeb graph) inserting node n_1 and arcs a_1, a_4 correctly modifies the graph to reflect inserting vertex v_1 and edges e_0, e_2 into the mesh. Sweeping a level-set from bottom to top a new contour (consisting of a single point) splits at v_3, evolves through edges e_2, e_0, and merges again at v_0. Inserting the interior of the triangle connects these new contours (represented by arcs a_1, a_4) with those that intersect triangle (v_0, v_2, v_3) (represented by the arcs a_2, a_0). This induces the removal of a loop in RG, as shown in Figure 3(c-d). The value of F at the vertices determines which edge, here e_4, directly connects the highest vertex, v_0, to the lowest vertex, v_3, and we call the corresponding path $p_1 = a_0, a_2$. The paths corresponding to the remaining two edges, e_0 and e_2, are concatenated to form a second path $p_2 = a_4, a_1$. In RG, both paths connect the highest node, n_0, to the lowest node, n_3. Merging p_1 with p_2 removes the loop from RG and creates a single path $p_3 = a_0, a_1, a_2$, as shown in Figure 3(d). At all times, this simple sequence of operations maintains the complete Reeb Graph of all simplices seen so far.

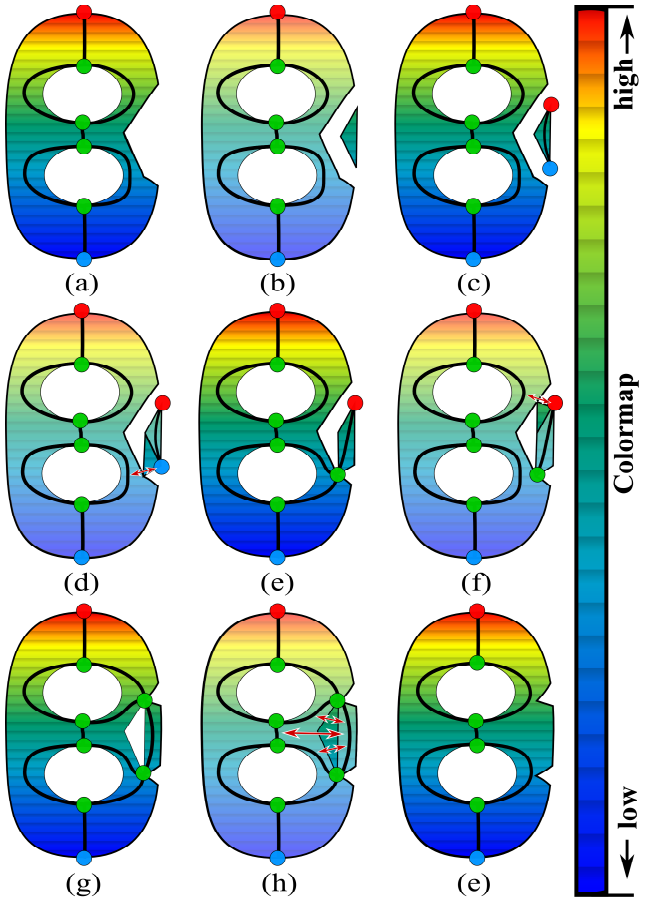


Figure 4: On-line update of a Reeb graph. (a) Initial mesh with its Reeb graph and critical points (red for max, green for saddle and blue for min). (b-c) Triangle adding a new component. (d-e) Triangle connecting two components. (f-g) Triangle creating a loop. (h-e) Triangle destroying a loop.

At the end, we make a pass through RG and remove degree-two nodes by merging their adjacent arcs. This last pass through the data is not necessary if one of the optimizations below is applicable.

Correctness. The natural way to show correctness of an on-line algorithm is by induction.

We first assume to have a 2D simplicial complex M with known Reeb graph RG. The central step of our algorithm is to simply add a new triangle t to M and update RG accordingly (as shown in Figure 4). This operation is performed correctly by adding first the

boundary of t and then merging the arcs of RG corresponding to contours of M that get connected by the interior of t. The result is a new mesh M' with correct Reeb graph RG'.

Formally, given the Reeb graph of a simplicial complex M with n triangles we can compute the Reeb graph of the simplicial complex $M+t$ having $n+1$ triangles. Moreover, the Reeb graph for any M having 0 triangles is the empty graph. Using this as the base for the induction, we can compute the Reeb graph of any 2D simplicial complex (no restriction to manifold or other special cases). In Section 3 we also show how this algorithm can be used for higher dimensional mesh.

Optimizations. We take advantage of additional information provided with the input mesh to reduce the storage requirements of our algorithm and improve performance. To this end, we introduce a new operation REMOVEEDGE(e_α) that follows the path corresponding to e_α and removes all occurrences of e_α from RG. REMOVEEDGE is used, for example, when the input model is known to be a two-manifold. In this case, an edge whose two adjacent triangles have already been processed cannot be involved in any subsequent updates of RG. Therefore, its reference can be removed from the graph as shown in Figure 3(e).

Similarly, it is often possible to determine that all triangles incident to a vertex have been read in input. With this information one can remove an edge when both its vertices have been finalized (even if the mesh is non-manifold). Notice that one can process meshes in streaming mode even if they are read in the original format and no preprocessing has been done to reorganize the data [Wu and Kobbelt 2003]. Moreover recent work [Isenburg et al. 2006] has shown that finalization information can be guessed when not available with little performance penalty if the heuristic is wrong.

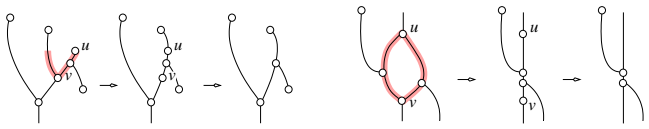
To further reduce the dynamic storage requirements we keep in memory only the portions of M and RG that may be traversed or modified in the future. Therefore, we remove from memory any triangle whose edges have become manifold, and any vertex that has become a manifold (is attached to a circular fan of triangles). If node n_i has degree two in RG, then we remove it from the graph and merge its two adjacent arcs into one. In this way, many nodes of RG can be removed without altering its structure. After several applications of REMOVEEDGE some arcs of RG may have empty edge pointer lists. Such arcs can be removed from memory as well since they will never be modified again (they correspond to closed contours that have been completed).

Embedding. While constructing the Reeb graph, we also compute its embedding based on the coordinates of the input vertices. We store with each arc an ordered sequence of buckets which partition the range of function value spanned by the arc.

Assume that the overall range of the function for a give model is $[0,100]$ and we want to subdivide it into 200 sections. With an arc a that spans values from 53.7 to 71.2, we store buckets with separating values $(53.5, 54, 54.5, \ldots, 71.5)$. Each bucket contains an integer NV to count the number of vertices accumulated and a vector P of the their average coordinates. When an arc is first created by CREATEARC the first and last buckets contain one vertex (the endpoints of the edge), while the rest of the buckets contain no vertices. For a short arc that spans only one bucket and one edge, its initial value is $NV = 2$ and P is the average of the two endpoints of the edge.

The basic algorithm modifies arcs in two ways (see Figure 3(f)): (i) by a MERGEPATHS operation, and (ii) when removing a node of degree two. In both cases we concurrently update the corresponding buckets merging them whenever necessary.

When we merge two buckets (NV_1, P_1) and (NV_2, P_2) with the same range but from two distinct arcs we obtain a bucket (NV, P) with



(a) (b)

Figure 5: (a) Maximum u is canceled with up-forking saddle v by first gluing the path (shaded) from u to v onto the other path starting at v, and then deleting nodes u and v. (b) Down-forking saddle u is canceled with up-forking saddle v by gluing two paths and deleting nodes u and v.

$NV = NV_1 + NV_2$ and $P = (P_1 NV_1 + P_2 NV_2)/NV$ (unless $NV_1 = NV_2 = 0$ in which case $NV = 0, P = (0,0,0)$). If two buckets contain some of the same vertices (because their arcs start and/or end at the same node), we subtract the contribution of such vertices from P and the decrement the value of NV accordingly. This avoids counting vertices multiple times.

In a MERGEPATHS operation two paths with the same endpoints are merged into one. In this procedure we traverse the two paths and their buckets from top to bottom and create a unique sequence of arcs with buckets as in a merge-sort algorithm, see Figure 3(f).

When we remove a degree two node n, it has node n_1 adjacent from above and node n_2 adjacent from below. We replace arcs (n_1, n) and (n, n_2) with arc (n_1, n_2). We obtain the sequence of buckets for (n_1, n_2) by concatenating those in (n_1, n) with those in (n, n_2) except for the two buckets containing n which we merge into one if they span the same range of function values.

The update operations are more complicated if we cannot estimate the range of the function before loading the mesh. Still, we want to estimate the embedding of RG using between k and $2k$ buckets. In this case we dynamically maintain the range $R = f_{max} - f_{min}$ while loading the mesh. Initially, the range is set to the function range of the first triangle or set of triangles we load in memory. The size of a bucket is set to 2^h, where h is the largest signed integer such that $k2^h \geq R$. The bucket containing the real value w has range $[i2^h, (i+1)2^h]$, where i is the largest signed integer such that $i2^h < w$.

Once we set the current parameter h we update the value of R each time we read a new input vertex. Whenever R surpasses $2k2^h$ the value of h is incremented until R is again between $k2^h$ and $2k2^h$. Thus, buckets periodically double in range while their separating values remain stable. The adjustment of buckets happens only in an amortized manner, immediately before an arc undergoes either a merge or a node removal. This strategy avoids repeated traversals of the graph.

On-line noise removal. Real-world functions are often noisy and one can expect their Reeb graphs to be noisy as well. Any practical use of such Reeb graphs requires noise removal. We use persistence based simplification [Edelsbrunner et al. 2000] to eliminate insignificant extremum-saddle pairs from the Reeb graph while it is being computed. The *persistence* of a pair of critical points is the absolute difference in their function values. As shown in Figure 5(a), we remove extremum-saddle pairs that are connected by a sequence of finalized arcs and whose persistence is less than a user-specified value. In our experience a large portion of most Reeb graphs consists of noise. Removing features with persistence less then 0.1% of the function range can shrink a graph by orders of magnitude without loosing any significant information. The denoised Reeb graphs are then subjected to a more extensive analysis by building a coarse-to-fine hierarchy as described next.

Simplification and Hierarchy. Simplifying the Reeb graph and storing it in a coarse-to-fine hierarchy is a requirement for any subsequent progressive visualization and/or processing. We simplify

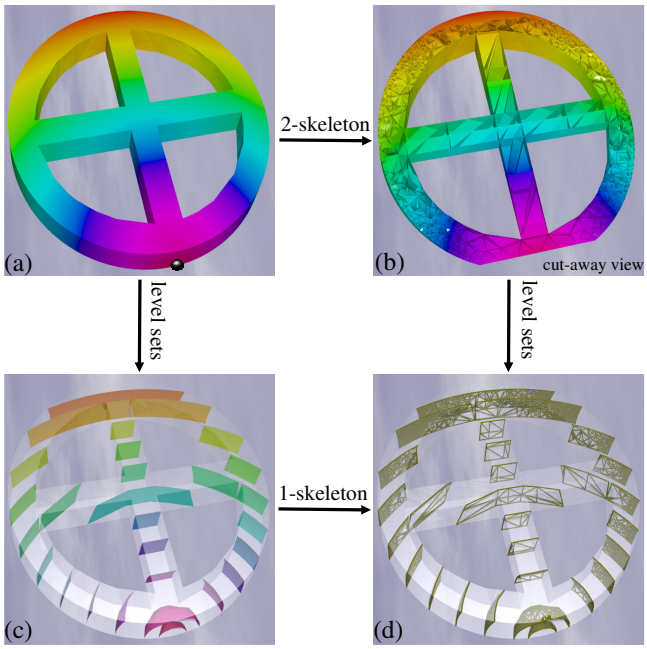


Figure 6: (a) Boundary of a tetrahedral mesh M pseudo-colored based on the distance F from the marked vertex. (b) Cut-away view of the two-skeleton $\hat{M}$ of M with the restricted function $\hat{F}$ as pseudo-color. (c) Several level-sets of F. (d) One-skeleton of the level-sets of F which are also the level-sets of $\hat{F}$.

the Reeb graph using the extended persistence algorithm [Agarwal et al. 2004] which generalizes the notion of persistence to loops. In addition to pairing maxima with *split-saddles* and minima with *merge-saddles* (as in regular persistence) their algorithm pairs split-saddles with merge-saddles which form a loop in the Reeb graph. This allows us to measure the persistence of loops and rank them uniformly with the other topological features.

We compute a persistence hierarchy for a Reeb graph in two phases: First, we compute all possible persistence pairs. Second, we cancel pairs of critical points in increasing order of persistence while maintaining the sequence of simplified Reeb graphs in a partially persistent data-structure [Driscoll et al. 1989]. This data-structure requires space proportional to that used by the original Reeb graph plus the number of updates to the graph, and allows accessing the Reeb graph simplified to any persistence threshold in time proportional to the size of the output. Figure 5 shows an extremum-saddle cancellation (a), and a saddle-saddle cancellation (b). Both simplifications glue together the paths from a saddle towards its paired node and remove both nodes (which now have valence two).

Combined with the de-noising step described above we generate the persistence hierarchy of all models using moderate resources. For example, we process the St. Matthew using a maximum of 150MB of memory. Once computed, we store the hierarchy on file in a coarse-to-fine order to enable progressive processing. In the file header, we provide a look-up table of memory usage of the Reeb graph at each simplification level to guide the user on how fine a persistence level one can load with fixed memory resources.

3 Higher Dimensional Meshes

The algorithm described above is very general and applies to simplicial meshes of any dimension on the basis of the following theorem:

Theorem. *The Reeb graph of a function F, defined on the simplicial complex M, is identical to the Reeb graph of a function $\hat{F}$, the restriction of F to the 2-skeleton $\hat{M}$ of M.*

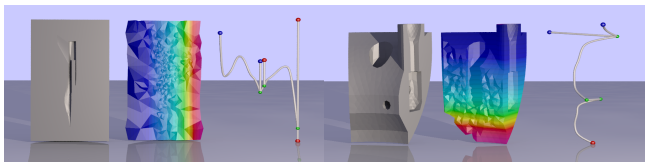


Figure 7: Two examples of Reeb graph of tetrahedral meshes often used in scientific visualization. On the left the fighter model with 70K tetrahedra. On the ritgh the SPX model with 20K tetrahedra.

Proof. By definition, the Reeb graph is obtained by contracting the connected components of level-sets to points. Consider a component of the level-set of function F defined on a d-dimensional simplicial complex M. This component is itself a $(d-1)$-dimensional complex (not necessarily simplicial) made up of $(d-1)$-cells (derived from d-simplices of M), whose boundary are $(d-2)$-cells (derived from the $(d-1)$-simplices of M), and so on. E.g: In 3D, a level-set component is a 2-dimensional complex made up of triangles or quads (derived from tetrahedra in M), bounded by edges (derived from triangles in M), and vertices (derived from edges in M). The connectedness of a level-set component is determined solely by its 1-skeleton, which is derived from the 2-skeleton $\hat{M}$ of M. Figure 6 shows a diagram that depicts how to get to the 1-skeleton of a level-set component of function F defined on a tetrahedral mesh. We can extract a level-set of F defined on M and then extract its 1-skeleton, or we can restrict F to $\hat{M}$ and extract the level-set of $\hat{F}$ defined on $\hat{M}$. Both paths arrive at the same 1-skeleton of the level-set component and thus produce the same Reeb graph. This proves the theorem. □

In practice, we process higher dimensional simplicial complexes by reading each simplex and applying the algorithm directly to its two-skeleton. Figure 7 shows the Reeb graph for two tetrahedral meshes commonly used for testing scientific visualization codes. Figure 8 shows the Reeb graphs of the Sierpinski simplex subdivided two times for dimensions two through five using one of the coordinates as function. We use the `dot` graph drawing package to compute the layout and even though the layout is certainly not optimal one can distinguish the $(d+1)$ clusters expected in dimension d.

4 Results and Conclusions

We have tested our algorithm on various models using for each a range of functions including x, y, and z coordinates and some eigenvectors of the Laplacian matrix of the input mesh (such as the one used for building streaming meshes). Because the choice of function impacts the running time we have also tested with random functions that yield very large, possibly impractical Reeb graphs, but constitute a very good test for performance and robustness. Figure 10 shows timing results for the worst case with random functions and for the average running time over the other functions. For comparison we also plot the mesh load times and the ideal linear scalability line. All tests were performed on a MacBookPro 2.3Ghz Intel Dual Core processor, with 2GB of main memory.

Our experiments show that, in practice, our new algorithm scales linearly both in the average and in the worst case where the running time doubles. At the high end we can compute the Reeb graph of the St. Matthew model with roughly 372 million triangles in just over eight minutes on average, see Table 2. Given the theoretical lower bound of $O(n \log n)$ for the computation of the Reeb graph on 2-manifolds these results are very encouraging.

Furthermore, we have the option of running in "out-of-core" mode in which the finalized parts of the data structures are removed on-the-fly and the results written to disk immediately. The last two columns of Table 2 and Figure 10(in red) show the memory used when running out-of-core. We can compute the Reeb graph of the

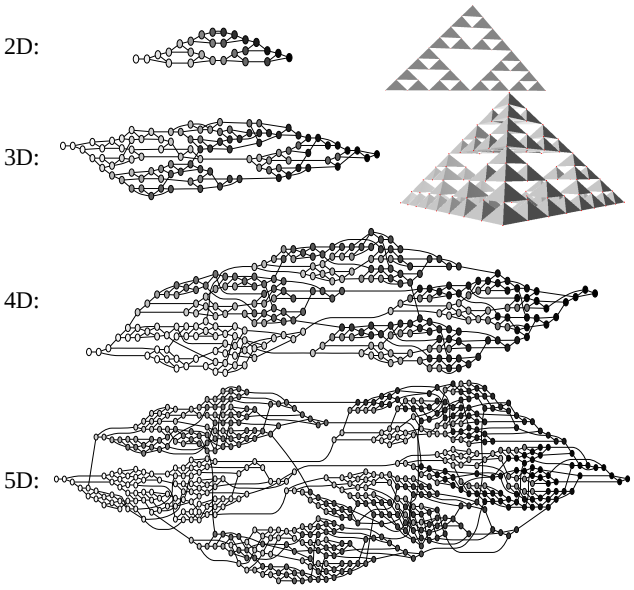


Figure 8: Reeb graphs of the Sierpinski simplex of dimension two to five subdivided twice. We use one of the coordinates as function value, shown here in grayscale.

Model name	number of triangles	RG compute time max	RG compute time avg	Load time	Mem. usage min	Mem. usage max
cow	5.8Kt	0.01s	0.01s	0.01s	23KB	33KB
feline4k	8.3Kt	0.02s	0.01s	0.02s	66KB	94KB
indian goddess	19Kt	0.13s	0.10s	0.28s	47KB	47KB
turtle	38Kt	0.08s	0.06s	0.07s	4.1MB	4.1MB
heptoroid	40Kt	0.08s	0.06s	0.08s	1.5MB	1.5MB
dancer	50Kt	0.24s	0.11s	0.10s	1.5MB	1.8MB
elephant	50Kt	0.10s	0.05s	0.05s	0.1MB	0.1MB
ganesh	413Kt	1.1s	0.85s	0.8s	0.2MB	0.3MB
dragon	871Kt	1.8s	1.0s	0.8s	0.1MB	0.1MB
goddess2	1.0Mt	3.1s	2.3s	1.9s	0.3MB	0.3MB
happy-buddha	1.1Mt	2.8s	1.4s	5.2s	0.3MB	0.3MB
gargoyle	1.7Mt	4.2s	3.2s	3.5s	1.0MB	1.0MB
malay. goddess	3.6Mt	14s	9.5s	7.0s	0.5MB	0.7MB
asian-dragon	7.2Mt	23s	8.7s	7.4s	0.5MB	0.5MB
thai-statue	10Mt	34s	13s	23s	1.0MB	1.0MB
Lucy	28Mt	101s	48s	47s	2.1MB	2.1MB
David	56Mt	218s	108s	94s	2.1MB	2.1MB
StMatthew	371Mt	1448s	486s	311s	8.4MB	8.4MB

Table 2: Performance results for a variety of models. Running times for in-core and memory usage for out-of-core computation.

St. Matthew model using between 8.3MB of main memory in the best case and 376MB in the worst case. This makes Reeb graph computations for even the largest available models feasible on commodity hardware. Surprisingly, due to the streaming nature of the algorithm the out-of-core computation has nearly no impact on the overall run-time. For example, the David computes in 218 seconds in-core versus 220 seconds out-of-core and Lucy in 102 seconds in-core versus 105 seconds out-of core. Coupled with the on-line noise removal described in Section 2 we process the St. Matthew model (using a reasonable function) from original surface to progressively stored Reeb graph using never more than 150MB of memory.

To study the dependency of the performance of our algorithm on the layout of the input data we report in Table 3 the Reeb graph computation time and memory usage for the meshes in their original (.PLY) order after adding vertex finalization. Table 3 also reports the preprocessing time to compute this finalization. We observe that in most cases the running times and memory usage increase

Model	Performance of RG computation for different functions $f=x$ time / mem	$f=y$ time / mem	$f=z$ time / mem	$f=random$ time / mem	conv. time
cow	0.1s / 33KB	0.1s / 23KB	0.1s / 33KB	0.1s / 33KB	0.1s
feline4k	0.1s / 66KB	0.1s / 66KB	0.1s / 66KB	0.1s / 94KB	0.1s
indian goddess	0.1s / 47KB	0.1s / 47KB	0.1s / 47KB	0.1s / 47KB	0.3s
turtle	0.2s / 4.1MB	0.2s / 4.1MB	0.2s / 4.1MB	0.2s / 4.1MB	0.4s
heptoroid	0.2s / 1.5MB	0.2s / 1.5MB	0.2s / 1.5MB	0.2s / 1.5MB	0.4s
dancer	0.2s / 1.5MB	0.2s / 1.8MB	0.2s / 1.8MB	0.2s / 1.8MB	0.3s
elephant	0.2s / 0.1MB	0.2s / 0.1MB	0.2s / 0.1MB	0.2s / 0.2MB	0.4s
ganesh	1.1s / 0.4MB	1.1s / 0.4MB	1.3s / 0.3MB	1.1s / 0.4MB	1.9s
dragon	2.2s / 1.0MB	2.2s / 1.0MB	2.3s / 1.0MB	2.2s / 1.0MB	4.5s
goddess2	3.8s / 0.4MB	3.7s / 0.4MB	4.2s / 0.4MB	3.8s / 0.4MB	3.6s
happy buddha	2.7s / 1.0MB	2.7s / 1.5MB	2.9s / 1.0MB	2.7s / 1.0MB	5.6s
gargoyle	4.6s / 1.0MB	4.3s / 1.0MB	4.4s / 1.0MB	4.3s / 1.0MB	5.6s
malay. goddess	9.8s / 0.7MB	9.7s / 0.7MB	11s / 0.5MB	9.9s / 0.7MB	9.6s
asian dragon	16s / 16MB	16s / 16MB	16s / 16MB	16s / 16MB	28s
thai-statue	31s / 16MB	30s / 16MB	31s / 16MB	30s / 16MB	40s
Lucy	1.8m / 94MB	1.8m / 94MB	1.8m / 94MB	1.7m / 94MB	72s
David	2.1m / 8.3MB	2.2m / 8.3MB	14m / 528MB	2.1m / 8.3MB	2.6m
StMatthew	15m / 8.3MB	3.8h / 376MB	16m / 16MB	15m / 8.3MB	25m

Table 3: Performance of the Reeb graph computation for models stored in the original triangle order. For each model we report computation time and memory usage for four functions. We report also the preprocessing time to add the finalization information to the file.

but remain comparable to those obtained for the optimal streaming meshes re-arranged in spectral order [Isenburg and Lindstrom 2005]. This is due to the fact that most meshes are in fact created with some degree of coherency and our algorithm is able to exploit such coherency when present.

Notice, that computing the finalization information is not an expensive operation. Given a mesh without finalization we perform a preliminary pass counting the number of triangles that use each vertex. During the second read the vertices are finalized when all their triangle references have been counted. Naturally, this operation is independent of the input function and it is usually better to add finalization information to the file to avoid re-computation. For completeness we report in Table 3 the finalization times for each mesh in the original triangle order. Notice that the finalization time is often dominated by the file I/O given the massive sizes of the original ply files and uncompressed output obj file. For example the StMatthew model has a 7.3GB input and 16GB output, the David model has a 1.0GB input and 2.3GB output, and the Lucy model has a 0.5GB input and 1.0GB output. Still, we are now able to process all these models on a commodity laptop computer.

Application. To demonstrate one application of our algorithm we use Reeb graphs to find and highlight small defects in geometric models. In particular, we can detect small manifold handles and tunnels and are not restricted to just finding non-manifold or missing triangles. The former defects are global in nature and therefore much harder to detect than the latter which can be found by checking a local neighborhood.

For each model we plot the number of loops above a certain persistence (as a percentage of function range) creating one-dimensional graphs, see Figure 9. These plots indicate both the topological complexity of a model as well as a threshold separating noise from features. Figure 9 shows three different examples: The plot for the Botijo on the left shows one very small and five larger features. A closer inspection reveals that, although completely manifold, the model contains some folded triangles creating a hidden tunnel. This flaw is not visible in standard rendering yet will adversely impact most geometric processing, e.g. simplification or parametrization. The plot for the Malaysian Goddess in the middle of Figure 9 shows a large flat region that separates the topological noise from the two desired tunnels. The close-ups above the plot

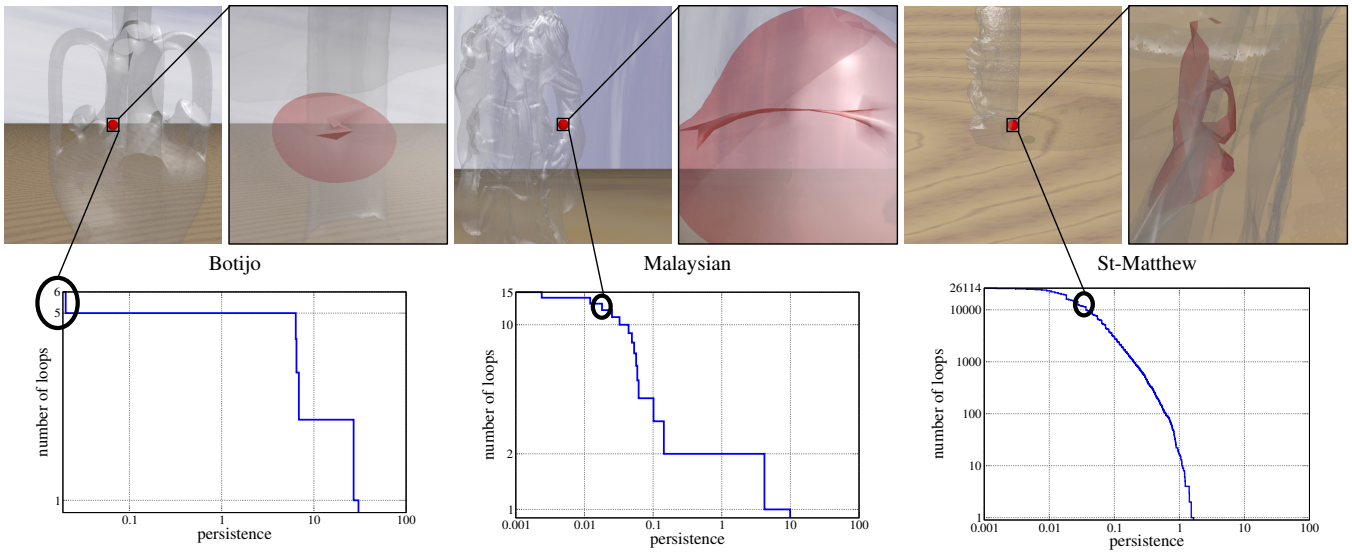


Figure 9: Finding small defects in surfaces:(Bottom) Plots of the number of loops in the Reeb graph above a given persistence for the Botijo, Malaysian Goddess, and St. Matthew model. The persistence is a percentage of the range of the function used to compute the Reeb graph. Both axis use a logarithmic scale and span several orders of magnitude. (Top) Close-up views of the neighborhood around small topological defects detected using the Reeb graph.

highlight the neighborhood around one small loop revealing a very thin handle, likely an artifact of the surface reconstruction process. The plot for St. Matthew on the right of Figure 9 shows no flat region and its largest loop has a comparatively low persistence. This is expected as the model ideally has genus zero and all tunnels and handles are in fact noise. The close-up view shows a small tunnel at the back of the statue.

Conclusions. In conclusion, we have presented a novel algorithm to compute, simplify, and hierarchically re-organize Reeb graphs of very large data sets without restrictions on dimension or complexity of the input. This represents a significant improvement over of the state-of-the-art relevant to a broad range of applications. In particular, when combined with the embedding discussed above, our algorithm immediately increases the range of applicability of most algorithms mentioned in Section 1 by several orders of magnitude. Despite its broad impact, our algorithm is easy to implement, robust, and can be run on commodity hardware.

Acknowledgments We thank P. Lindstrom and M. Isenburg for giving us access to their streaming meshes. The video associated with this paper was edited by E. Cronshagen and R. Gaunt.
This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.
This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET).

References

AGARWAL, P. K., EDELSBRUNNER, H., HARER, J., AND WANG, Y. 2004. Extreme elevation on a 2-manifold. In *SCG '04: Proceedings of the ACM Symp. on Computational Geometry*, 357–365.

ATTENE, M., BIASOTTI, S., AND SPAGBUOLO, S. 2001. Remeshing techniques for topological analysis. In *Proc. of Shape Modeling International*, 142–153.

BIASOTTI, S., MORTARA, M., AND SPAGNUOLO, M. 2000. Surface compression and reconstruction using Reeb graphs and shape analysis. In *Proc. of Spring Conf. on Comp. Graph.*, 175–184.

BIASOTTI, S. 2001. Topological techniques for shape understanding. In *Central European Seminar on Computer Graphics, CESCG 2001*.

CARR, H., SNOEYINK, J., AND AXEN, U. 2003. Computing contour trees in all dimensions. *Comput. Geom. Theory Appl. 24*, 3, 75–94.

CARR, H., SNOEYINK, J., AND VAN DE PANNE, M. 2004. Simplifying flexible isosurfaces using local geometric measures. In *VIS '04: Proceedings of the IEEE Visualization 2004* , 497–504.

CHIANG, Y.-J., LENZ, T., LU, X., AND ROTE, G. 2005. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Comp. Geom.: Theory and App. 30*, 2, 165–195.

COLE-MCLAUGHLIN, K., EDELSBRUNNER, H., HARER, J., NATARAJAN, V., AND PASCUCCI, V. 2003. Loops in reeb graphs of 2-manifolds. In *Proceedings of the 19th Annual Symposium on Computational Geometry*, ACM Press, 344–350.

DRISCOLL, J. R., SARNAK, N., SLEATOR, D. D., AND TARJAN, R. E. 1989. Making data structures persistent. In *J. Comput. Sys. Sci*, vol. 38, 86–124.

EDELSBRUNNER, H., LETSCHER, D., AND ZOMORODIAN, A. 2000. Topological persistence and simplification. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, 454.

FUNKHOUSER, T., AND KAZHDAN, M. 2004. Shape-based retrieval and analysis of 3D models. In *SIGGRAPH '04: Proceedings of the conference on SIGGRAPH 2004 course notes*, ACM Press, New York, NY, USA, 16.

HILAGA, M., SHINAGAWA, Y., KOHMURA, T., AND KUNII, T. L. 2001. Topology matching for fully automatic similarity estimation of 3D shapes. In *Proceedings of ACM SIGGRPAH 2001*, E. Fiume, Ed., ACM, 203–212.

ISENBURG, M., AND LINDSTROM, P. 2005. Streaming meshes. In *Proceedings of the IEEE Visualization 2005 (VIS'05)*, IEEE Computer Society, 231–238.

ISENBURG, M., LINDSTROM, P., GUMHOLD, S., AND SHEWCHUK, J. 2006. Streaming compression of tetrahedral volume meshes. In *Proceedings of Graphics Interface 2006*, 115–121.

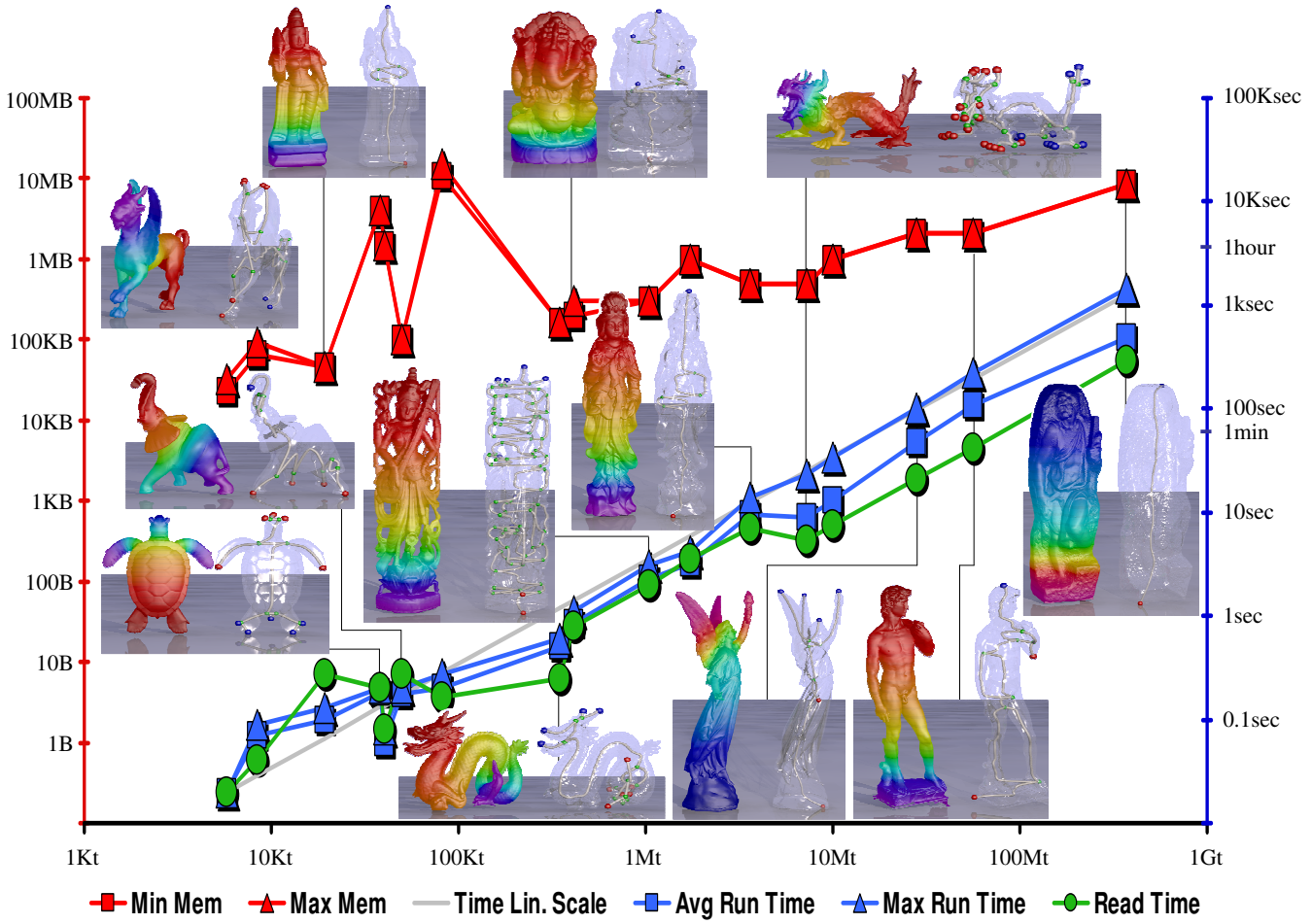


Figure 10: Performance measurements for the Reeb graph computation algorithm. We consider models from 6K triangles (Cow) to 372M triangles (St. Matthew statue). For reasonable input functions (blue squares) we achieve consistent processing speeds of about 800K-900K triangles per second. In stress tests with a random function (blue triangles) the performance decreases to about 400k-500k triangles per second. The running time is roughly twice the time needed for loading the input mesh (green circles) in compressed streaming format. The red plots show minimum (squares) and maximum (triangles) memory usage among all tests.

LAZARUS, AND VERROUST, A. 1999. Level set diagrams of polyhedral objects. In *Proceedings of the fifth ACM Symposium on Solid mMdeling and Applications*, ACM Press, 130–140.

NI, X., GARLAND, M., AND HART, J. C. 2004. Fair morse functions for extracting the topological structure of a surface mesh. *ACM Trans. on Graphics (TOG)* , 613–622.

PASCUCCI, V., AND COLE-MCLAUGHLIN, K. 2003. Parallel computation of the topology of level sets. *Algorithmica 38*, 1 (Oct.), 249–268.

REEB, G. 1946. Sur les points singuliers d'une forme de pfaff completement intergrable ou d'une fonction numerique [on the singular points of a complete integral pfaff form or of a numerical function]. *Comptes Rendus Acad.Science Paris 222*, 847–849.

SHINAGAWA, Y., AND KUNII, T. 1991. Constructing a Reeb graph automatically from cross sections. *IEEE Computer Graphics and Applications 11*, 44–51.

SHINAGAWA, Y., KUNII, T., AND KERGOSIEN, Y. L. 1991. Surface coding based on Morse theory. *IEEE Computer Graphics and Applications 11*, 66–78.

STEINER, D., AND FISCHER, A. 2001. Topology recognition of 3D closed freeform objects based on topological graphs. In *SMA '01: Proceedings of the sixth ACM symposium on Solid modeling and applications*, ACM Press, New York, NY, USA, 305–306.

STEINER, D., AND FISCHER, A. 2002. Cutting 3D freeform objects with genus-n into single boundary surfaces using topological graphs. In *SMA '02: Proceedings of the seventh ACM symposium on Solid modeling and applications*, 336–343.

TAKAHASHI, S., SHINAGAWA, Y., AND KUNII, T. L. 1997. A feature-based approach for smooth surfaces. In *Proc. of the Fourth Symp on Solid Modeling and Applications*,97–110.

VAN KREVELD, M. J., VAN OOSTRUM, R., BAJAJ, C. L., PASCUCCI, V., AND SCHIKORE, D. 1997. Contour trees and small seed sets for isosurface traversal. In *Symposium on Computational Geometry*, 212–220.

WEBER, G., SCHEUERMANN, G., HAGEN, H., AND HAMANN, B. 2002. Exploring scalar fields using critical isovalues. In *Proc. IEEE Visualization '02*, IEEE Computer Society Press, IEEE, 171–178.

WOOD, Z. J., DESBRUN, M., SCHRODER, P., AND BREEN, D. E. 2000. Semi-regular mesh extraction from volumes. In *Proc. IEEE Visualization '00*, IEEE Computer Society Press, Los Alamitos California, 275–282.

WOOD, Z., HOPPE, H., DESBRUN, M., AND SCHRÖDER, P. 2004. Removing excess topology from isosurfaces. *ACM Trans. Graphics (TOG) 23*, 2, 190–208.

WU, J., AND KOBBELT, L. 2003. A stream algorithm for the decimation of massive meshes. In *Proc. Graph. Interf.*, 185–192.