Eurographics Italian Chapter Conference (2008)
V. Scarano, R. De Chiara, and U. Erra (Editors)

Topological Operations on Triangle Meshes Using the OpenMesh Library

Fabio Guggeri, Stefano Marras, Claudio Mura, and Riccardo Scateni

Dipartimento di Matematica e Informatica, Università di Cagliari, Cagliari, Italy

Abstract

Recent advances in acquisition and modelling techniques led to generating an exponentially increasing amount of 3D shapes available both over the Internet or in specific databases. While the number grows it becomes more and more difficult to keep an organized knowledge over the content of this repositories. It is commonly intended that in the near future 3D shapes and models will be indexed and searched using procedure and instruments mimicking the same operations performed on images while using algorithms, data structures and instruments peculiar to the domain.

In this context it is thus important to have tools for automatic characterization of 3D shapes, and skeletons and partitions are the two most prominent ones among them. In this paper we will describe an experience of building some of this tools on the top of a popular and robust library for manipulating meshes (OpenMesh). The preliminary results we present are promising enough to let us expect that the sum of the tools will be a useful aid to improving indexing and retrieval of digital 3D objects.

The work presented here is part of a larger project: Three-Dimensional Shape Indexing and Retrieval Techniques (3-SHIRT), in collaboration with the Universities of Genoa, Padua, Udine, and Verona.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Geometric algorithms, languages, and systems

1. Introduction

In this work we show some preliminary results we obtained in extracting topological features from triangle meshes using a generic and efficient data structure for representing and manipulating polygonal meshes. We will present how several known algorithms were implemented and integrated using the library and the improvements and additions we did and we are planning to do. The aim of this paper is to present in a common framework several tools developed for the common problem of characterizing 3D shapes, indexing them and compare different objects among them for retrieval purposes. It is part of a larger two-years project to which people coming from different fields (interactive computer graphics, image processing, computer vision) are collaborating for building a system for indexing and retrieval of 3D objects.

The rest of the paper is organized as follows: in section 2 we will tell why it is important to extract and classify topological characteristics from triangle meshes, and briefly recall the work upon which our system is built over, in section 3 there will be a short description of OpenMesh and its features, sections 4 and 5 will be dedicated to describe the results we obtained up to now, and in section 6 we will draw our conclusions and explain how the work will continue in the future.

2. Topological features of triangle meshes

Skeletonization and 3D segmentation are both useful tools for representing 3D meshes for indexing and retrieval proposals.

Skeletons are compact representations of the models they are extracted from and carry both topological and geometrical information in a simple and exhaustive form. Medial axis and topological skeleton are very interesting features of three-dimensional objects that can be used in many application fields ranging from shape description to medical imaging. While defining what the skeleton is in 2D is pretty

© The Eurographics Association 2008.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

simple (the geometric locus of the centers of maximal disks contained in the original object [Blu67]), extending the definition to 3D is surprisingly difficult since it is formed by surfaces and medial curves (thought as an extension of medial axes). Restricting the skeleton to 1D brings it to be formed only by curves. Since there is a strict correlation between medial surfaces and skeleton, in literature the two terms are often used interchangeably. Once extracted one can use the skeleton both to define the object shape [SMD*05,OFCD02] and to help in performing the segmentation; they are particularly useful in describing complex models.

A working definition *skeleton* is given in [CSM07]: the locus of centers in maximal inscribed balls. Formally, let $X \subset R^3$ a 3D shape; a *ball* of radius r centered at x is defined as $r(x) = \{y \in R^3, d(x,y) < r\}$ where $d(x,y)$ is the distance between x and y. A ball is maximal if it is not included in any other ball. The set including all the centers x of maximal balls is the skeleton of X. It is possible to give also the definition of skeleton in terms of properties it holds [ZT99]:

homotopy the skeleton must be topologically equivalent to the original 3D shape, that is it should have the same number of connected components and holes;

thinness it is, in fact, desirable that it is just a set of curves to make it possible an easy transformation in a graph where each curve segment is an arc and each junction is a node;

centeredness each point of the skeleton is, theoretically equidistant from the two closest points of the mesh;

robustness with respect to noise on shape surface;

smoothness it should be as smooth as the original surface and, possibly, catch the discontinuities and reduce them.

Different algorithms for skeleton extraction have been proposed and discussed [SLSK07]. In [GS99], D. Silver and N. Gagvani present an algorithm for parameter controlled volume thinning that, starting from a volumetric representation of a 3D object, uses a combination of distance and a user-defined parameter, referred as *thinness parameter*, to detect skeleton voxels.

Segmentation is a vast and complex domain, both in terms of problem formulation and resolution techniques. It consists in formally translating the delicate visual notions of homogeneity and similarity, and defining criteria which allow their efficient implementation. The goal is to partition the source data into meaningful pieces, i.e. those parts corresponding to the different entities, in the physical and semantic sense of the application envisioned. In the realm of 3D data, several surveys report interesting approaches for different data representations such as unorganized points, range image, or 3D polygonal meshes [AA93, HJBJ*96, Pet02, AKM*06]. Roughly speaking, the segmentation methods can be categorized into two main classes: edge-based and region-based [Pet02]. In the former, features corresponding to part boundaries are first detected and then regions are built, each one formed by sets of points delimited by the same boundary. In the latter, points sharing the same similarity property are grouped together. In particular, three are the most popular approaches to region-based segmentation: split-and-merge methods, identified by a top-down paradigm; region-growing methods, that adopt a bottom-up paradigm, and clustering-based methods, based on the projection of the points onto a higher dimensional space where the clusters (i.e., segments) are recovered by defining some particular distance functions [JMF99].

3. The OpenMesh library

OpenMesh is a data structure, developed by the Computer Graphics Group RWTH (University of Aachen) [Ope], for representing and manipulating meshes. OpenMesh is able to handle and manipulate mesh composed by arbitrary polygons, for general cases, but it's obviously specialized in triangle meshes. For each triangle, information about vertices, half-edges, edges and faces are stored; also information about connectivity (half-edge incident to a face, vertices of an half-edge, etc) and topology are stored. Each element contains also specific information (normal for vertices, color for faces, etc), but the entire structure is customizable, so new information can be added to an element when needed. Spatial coordinate system is the standard $x-y-z$, and coordinates can be represented using float precision or double precision.

The elements of the mesh (vertices, half-edges and faces) form the kernel of the mesh; element of the same type (for example, faces) are stored in a doubly-linked list, implemented using array. For each element, a data structure called *handle* is specified; the handle is used as an index of the array, but it is also used in order to access information about the element. Attributes of the elements (predefined or user-defined) form the *traits* of the mesh. More in details, basic operations on mesh element (add/remove/access) are defined in the `BaseKernel` class; attributes are added using the `AttribKernelIt` class, and, finally, the class `ArrayKernelIt` performs basic operations directly on the array. Each element is defined in its specific class (`HalfEdgeT`, `FaceT`, `VertexT`); particular data structures, called *iterators*, are used in order to pass from an element to another according to the connectivity and the topology of the mesh. For example, the Vertex-Edge iterator is used to visit all edges (half-hedges) incident in a vertex. User-defined features can be added modifying `Property` of elements.

The particular structure of OpenMesh has the advantage of keeping memory usage as low as possible (no dynamic allocation, no virtual function tables, etc) and, at the same time, OpenMesh tries to be easy to use and to manipulate, and it is highly adaptable for multiple uses and algorithms.

OpenFlipper is a framework offering great support for the development of OpenMesh-based applications: it provides classes for loading `.off` meshes into a scenegraph and for

© The Eurographics Association 2008.

performing the most common operations on them. It also offers an intuitive and clear graphical interface, which results particularly handy when the user wants to interactively apply some effects on the loaded meshes, such as lighting and coloring effects. Appropriate classes and methods allow to translate, rotate and scale meshes, to select single mesh items (or a subset of them), to obtain information about them and to apply several different visualization options; a special option allows the user to get snapshots of the working environment and to save them in standard formats. Another relevant feature is an embedded text console which can be used to show a textual output during computation, particularly useful for application debugging.

It is worth to notice that this operations can be performed both through the GUI and by using C++ code inside one's own application: this is probably the most important aspect for programmers who might want to provide a graphical front-end to their programs. In particular, OpenFlipper was designed to simplify its extension by creating plug-ins, since there is a special set of classes that provides a standard interface for those who want to create their own plug-in. Plug-ins consist of dynamic libraries, are highly independent and make OpenFlipper a highly modular environment. Each user can personalize the set of plug-ins he wants to use, by simply selecting and copying them in the appropriate application folder.

OpenFlipper can be run under Linux systems, it is written in C++ and uses QT4 libraries [Qt] for the graphical interface, while OpenGL APIs are used for visualization purposes.

4. Voxelization and skeletonization

Since the algorithm in [GS99] needs a voxel-based representation, a first step to be accomplished is to switch from a mesh-based shape to a volumetric representation of the shape itself. This can be done by putting the mesh into a suitable data structure: using an octree is quite a natural choice. The cells of the octree will be marked as *shape* or *background*, using a flag; background voxels are discarded, since they are useless in the extraction algorithm. While performing the voxelization we keep track of which voxel contains which element of the mesh (e.g., if a vertex is inside a voxel, then the voxel will contain a pointer to the vertex and its incident faces) in order to keep $O(1)$ the order of magnitude of the search for neighbor voxels. In this way, it is possible to pass from the voxel-based structure to the original mesh, and back as needed.

During the implementation of the octree data structure, many technical issues raised. Think, for instance, to the management of an octree that has to handle a set of points in space: it is not a particular complex problem; but things change when the octree has to handle a larger quantities of different data: points and faces, keeping the correct topology and connectivity. Choosing the right, efficient and robust methods for testing intersections between voxels and faces is not an easy task, and, even if in literature there are different solutions proposed, it's not easy to choose the most suitable for our purposes.Another fine technical issue related to the implementation of the data structure is the distinction between *object* voxels and *background* voxels. As a matter of fact, usual meshes don't provide any method for distinguishing between inside and outside, so we developed different approaches to the problem: the main idea was to select a subset of voxels, marked as *internal*, and then use them as seed points to individuate internal cell, searching in the neighborhood of seed voxels. Selection of seed voxels can be done using centroids, for convex meshes, or using a kind of *discrete curvature approximation* to search point on the surface, and then propagate the voxelization inward. Both this approaches were implemented, and the user can choose the best for his needs.

The first step of the skeletonization algorithm consists in computing the minimal distance between each voxel in the interior of the object and the boundary of the object itself, that is the layer of voxels containing the mesh. This distance, called *Distance Transform* ($\mathcal{DT}$), at a voxel $p = x,y,z$ is defined as $DT_p = min\{d((x,y,z),(i,j,k)) : (i,j,k) \in BV\}$ where d is the distance between voxel p and voxel (i,j,k), and BV is the set of boundary voxels. $\mathcal{DT}$ can be computed using several metrics, the simplest one is a standard $< 3-4-5 >$ metric (where 3 is the distance between voxels sharing a face, 4 between voxels sharing an edge, and 5 the distance between voxels sharing only a vertex). A default value of 3, 4 or 5 is assigned to each voxel in BV; then, computation is propagated inward, in order to assign the right $\mathcal{DT}$ value to each voxel of the object. At the end of the first step, every voxel will be labeled with its $\mathcal{DT}$ value indicating how far it is from the boundary. Ideally, skeleton will contain points that are as far as possible from the boundary to respect the centeredness property.

In the second and last step of the algorithm, the voxels belonging to the skeleton are identified using maximal balls. In fact, centers of maximal balls are individuated using relationship between the $\mathcal{DT}$ of the voxel p and the $\mathcal{DT}$ of the 26 neighbors of p. Let S be the skeleton of the object, then $p \in S$ if the mean of neighbors'$\mathcal{DT}$'s (MNT) is lower than the $\mathcal{DT}$ of the voxel p ($\mathcal{DT}_p$). Using only this condition, the skeleton could not be completely thinned, and some surfaces could be still present in the skeleton. To avoid this, the *thinness parameter* $\mathcal{TP}$ is introduced, and a voxel $p \in S$ if the $MNT_p < \mathcal{DT}_p - \mathcal{TP}$. Use of $\mathcal{TP}$ makes it possible to arbitrary set the thinness of the skeleton. Ideally, $\mathcal{TP}$ should be choose in order to complain with the thinness property and obtain a 1-voxel thin skeleton.

In order to correctly implement the algorithm, we ran into some problems. First of all, the OpenMesh library, as the name suggest, does not provide any information about vol-

© The Eurographics Association 2008.

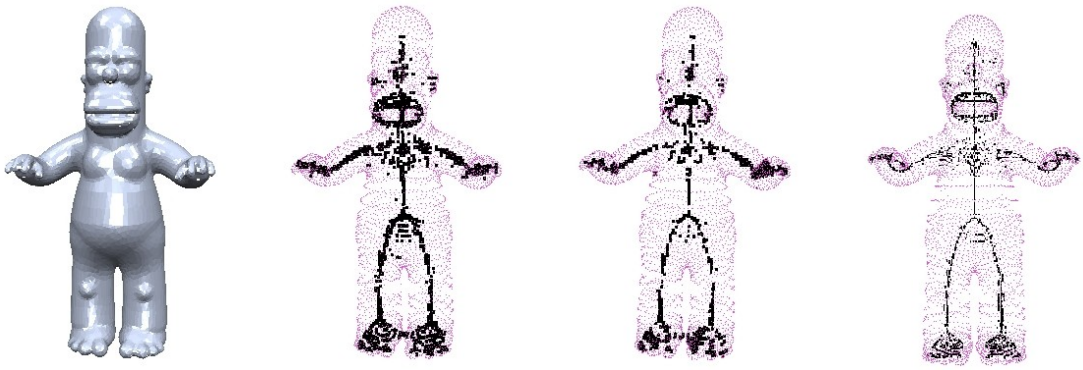

Figure 1: *The leftmost image shows the 3D object under analysis; in the other three images we can see its skeleton extracted using three different parameters setups for the octree and the thinning algorithm. From left to right:* $\mathcal{TP} = 1.5$ *and* $LOD = 8$*;* $\mathcal{TP} = 1.75$ *and* $LOD = 8$*;* $\mathcal{TP} = 1.75$ *and* $LOD = 9$*.*

umetric representations of objects, and has no data structure already implemented, so it was necessary to develop from scratch an octree data structure while preserving the information on mesh topology. The other main issue to face was related to the choice of the thinness parameter $\mathcal{TP}$, which should be small enough to minimize spurious component, but also large enough to preserve the skeleton connectivity. It is recommended to refine the skeleton in a post-processing steps (e.g., connecting components not connected at the end of the computation), or eliminating spurious voxels. Figure 1 shows the results of the computation of the skeleton of a mesh of 9856 faces and 4930 vertices using different values both for the level of refinement of the embedding octree, and $\mathcal{TP}$. The fourth skeleton is computed using a different level of refinement for the octree, while the second uses a different $\mathcal{TP}$.

5. Segmentation

5.1. Fuzzy clustering

The segmentation obtained is based on a strictly geometric criterion: the more distant two faces are, the less likely they are to belong to the same patch, while if two faces are near there exists a high probability that they belong to the same patch. Both angular and geodetic distances are taken into account when computing distance between two faces, according to the following formula: $\alpha * dist_{ang} + (1-\alpha) * dist_{geod}$, where α, the weight in the formula, is tipically quite high to ensure that the segmentation is done according to the minima rule. Once distances between adjacent faces have been computed, an *all-pair shortest paths* (APSP) algorithm is run on the dual graph of the mesh, as described in [KT03]. The core of APSP is a slightly modified version of the original Dijkstra shortest-path algorithm [Dij59], which takes advantage of the fact that the dual graph of a mesh is actually an undirected graph. Once distances have been computed between each pair of faces, a set of representative faces (one for each patch) is chosen and for each face the probability of belonging to each patch is computed. Faces are then clustered, inserting all those faces which are almost as likely to belong to a patch as to the other in a fuzzy region [KT03]. These faces tipically belong to a boundary between two patches. To determine whether a face in the fuzzy region belongs to a patch or to another, a network flow graph is constructed and Edmonds-Karp algorithm [EK72] is run on it. Finally, when segmentation is completed, each patch is given a color, and each face is colored according to the patch it belongs to.

In this context, OpenMesh and OpenFlipper offer a great support to an efficient implementation. First of all, the OpenMesh data structure allows for an easy and quick navigation of the mesh, by providing ad-hoc iterators and circulators: this is particularly useful in this algorithm, since it often implies performing operations on the dual graph of the mesh. Having a simple and efficient way to iterate through faces and edges, there was no need to implement an alternative data structure for the mentioned graph. Since it offers a large number of navigation functions, OpenMesh proves to be very useful for a quick deployment of applications which work on meshes. Moreover, this data structure can be easily extended using *traits* and dynamic properties, which showed particularly useful to store distances and to flag visited faces during the APSP execution, and to store the id of the patch whom a face belongs to during clustering and border refinement steps. Another relevant feature is the possibility of setting a color for mesh items (edges, faces, vertices): this is very useful to obtain a visual result and to show in an intuitive and immediate way the effect of the segmentation.

If OpenMesh is a valid basis for the core of the application, OpenFlipper offers a set of functions that are particularly useful for those programmers who want to embed their applications in an interactive graphical environment, based on QT4 libraries. OpenFlipper can be easily extended by creating personalized plug-ins, which consist of `.so` files.

© The Eurographics Association 2008.

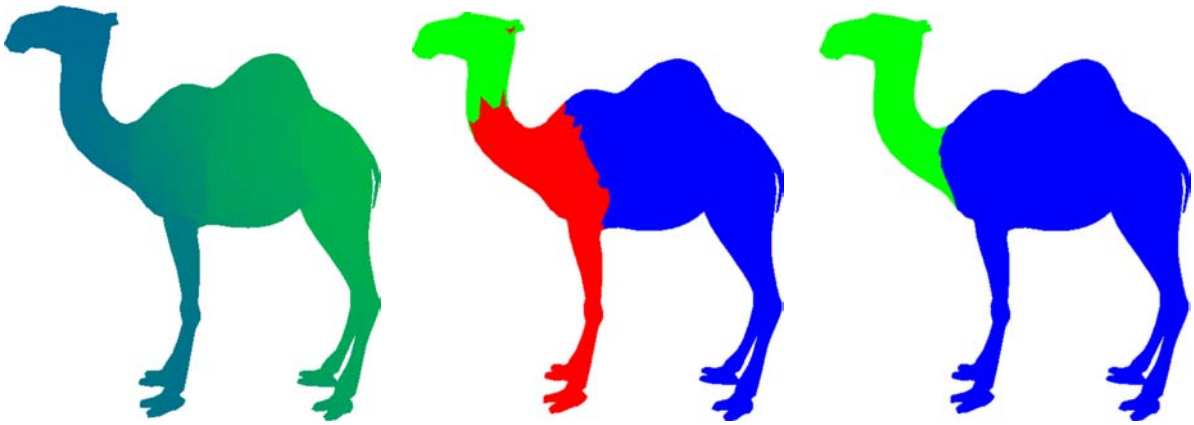

Figure 2: *From left to right: color coding of the probability distribution (green to blue); a red band encodes the uncertain faces; the assignment completely resolved.*

A set of C++ headers containing some interfaces (classes which encompass only virtual functions) to be implemented is available for the programmer, who can thus create a personalized class and insert standard QT4 objects in it. We developed a plug-in, integrating a Tool Box in the global interface: some `QSliders` objects allow users to tune the most important parameters used in the algorithm, (such as the above-mentioned α weight); different `QButtons` are placed on the Tool Box, and each of them is connected to a function that implements a single step of the algorithm, so that the user can see the effects of each phase of the segmentation. Besides, a set of available plug-ins provide an easy way to manually select either single or multiple mesh items, to obtain information about them and to save the selection for further use: this proves useful both to perform a visual debug of the algorithm and to collect data to be used in subsequent phases. In Figure 2 we can see images showing three different stages of the segmentation process: initially we identify the two furthest faces (using geodetic and angular distance) and we compute the probabilities for each face to belong to the segment containing each of the two faces; then we classify the faces in three cluster, one containing the faces with a reasonable probability to belong to one segment, one containing the faces belonging to the other segment using the same criterion, and a third containing the faces which attribution is uncertain (the fuzzy band); at last the we resolve the uncertainty using the min-cut max-flow algorithm.

5.2. Morphing

Defining correspondences between meshes is an important step in the morphing process. In order to obtain a more visually appealing effect for the user, one would like to have some parts of the source mesh to be morphed directly to similar parts of the target mesh, for example, a viewer would expect that the head and paws of a camel would correspond to the head and paws of a cow during a continuous morphing visualization. This means that a morphing process should be aware of which vertices should be constrained to be mapped directly each other. This kind of vertices are called *feature points* (f.p.), and are selected by the user. After selecting the feature points in both meshes and defining the direct correspondences between the f.p.'s in the first (*source*) and the second (*target*) mesh, the goal is to provide a common base domain for the mapping in order to create the morphing function as a combination of the three mapping functions (target to base domain, source to base domain, base domain to base domain) [KS04].

The base domain is defined as a set of topologically equivalent triangular patches whose vertices are the f.p.'s selected by the user. This means that we aim to construct a triangulation of the f.p.'s of both meshes so that these triangulations have the same topology, leading to a simple one-to-one mapping for the "base domain to base domain" mapping function. This triangularization defines a segmentation of the two meshes starting from the feature points: the MatchMaker algorithm [KSG03] is used to obtain this topologically equivalent segmentation, defining patch borders as shortest paths on the mesh between feature points and then removing the paths that don't preserve the topology constraint.

First of all, candidate paths are computed as shortest paths between feature points using Dijkstra's algorithm over the mesh edges. Every new path is then checked to see if it satisfies three conditions:

1. No paths should intersect, except at their end-vertices;
2. The new path does not block any necessary future paths; this means that if a new path generates a closed region, the feature points included in the region should be the same in both meshes;

© The Eurographics Association 2008.

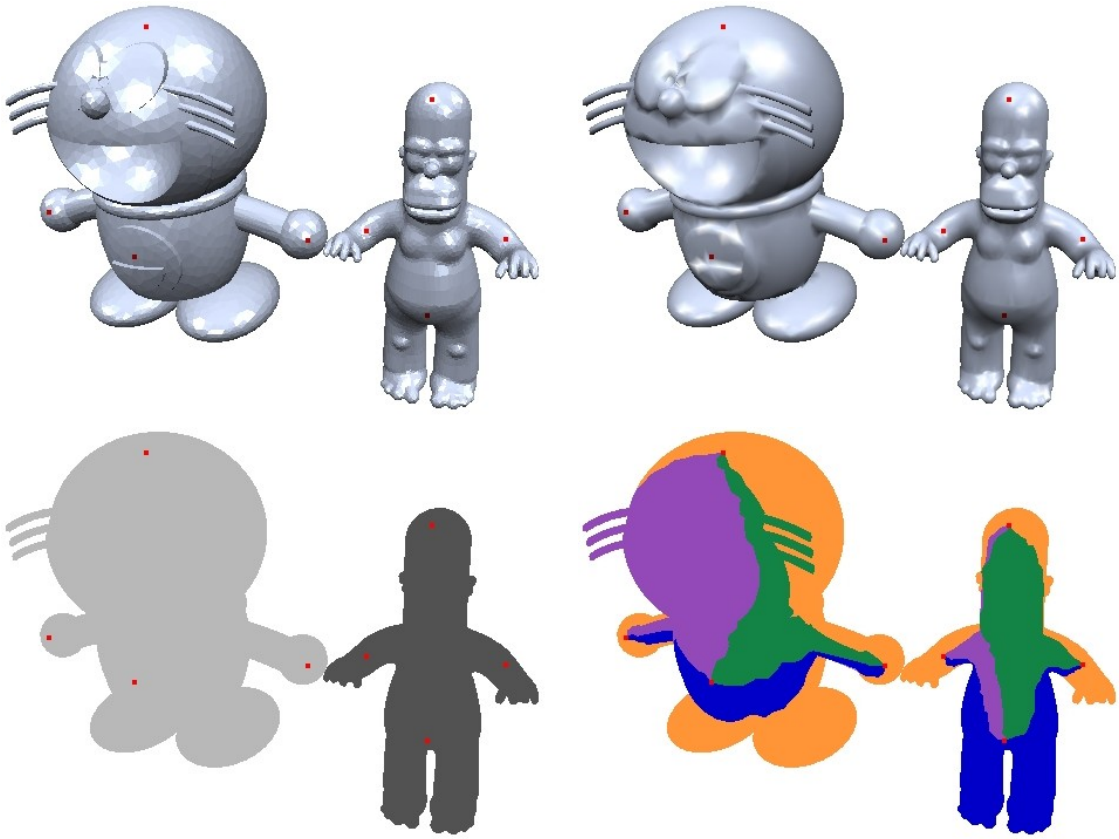

Figure 3: *On the top row the two meshes, flat and Gouraud shaded, with the feature points highlighted; on the bottom row: the meshes before the segmentation starts, a single patch covers the whole figure (left), the segmented meshes: colors code the topological equivalence of the patches.*

3. The regions should have the same orientation on both meshes.

Paths are then added to the final segmentation until no more paths can be added. If a triangular segmentation is obtained, the common base domain construction is finished. If not, further steps of refinement are needed.

Implementation of the MatchMaker algorithm using OpenMesh is simplified by the ability of the data structure to dynamically store properties relative to elements of the mesh (vertices, edges and faces). This feature is useful when running algorithms on the mesh such as Dijkstra's shortest path search, where the distance for each vertex to the source vertex can be easily recalled as a dynamic property created with OpenMesh. Feature points are user-selected using the OpenFlipper picking interface, and stored in two vectors depending on the mesh they belong to. A boolean property, created for each vertex, tells whether the vertex is a feature point or not. When the user is done with feature points selection (see first three images of Figure 3), selecting more than three f.p.'s on each mesh (and the same number between the meshes), the algorithm computes the shortest paths between those vertices using a modified version of Dijkstra's algorithm in order to force a path between two feature points not to step over a third feature point, a fact that would cause a degenerate topology in later computations. To avoid that, feature points are considered as end-nodes of a graph and the outgoing edges aren't added in the cut-set, so that a path that reaches a feature points can't go further to reach a second one.

Similarly, when recomputing a candidate path that overlaps a definitive one, we use another version of the shortest path search where vertices that are already part of a patch border are avoided and are not stored in the queue, so that the recomputed path cannot intersect any of the definitive patch borders. It's easy to differentiate between *free* vertices and vertices which form a border thanks to the previously mentioned dynamic properties; a boolean property that acts as a flag for each component is set to show whether it has a special role in the topology construction or not. The same is done for edges that form a border, faces that belong to a

© The Eurographics Association 2008.

patch (where the property is an integer that represents the index of the patch it belongs to), and any other information directly connected with mesh's elements.

After the computation of the shortest paths, on each iteration, the pair of corresponding paths whose sum is the smallest among every pair is analyzed to see if it can become a definitive border satisfying the topology constraints. The second constraint is satisfied if the vertices found inside the patch at the right (or left) side of a path correspond to the ones found at the right (or left) side of the corresponding path on the other mesh. This control is implemented as a front propagation from the right side of each path and storing every feature point encountered by the front inside the starting patch. Comparison between the two encountered sets is made easier and faster by storing an *user insertion index* on each feature point and, then, implementing the sets as priority queues based on such index, checking if the top value on each queue is the same and popping and re-checking until the queues are empty or the condition fails. If this is the case, then one of the paths would be blocking paths not yet computed and, then, the pair cannot be accepted as definitive. Elsewhere, the pair is stored as a definitive border on each mesh and overlapping candidate paths must be recomputed.

If it's impossible to find a new path during recomputing, the pair is simply removed from the pool. When a new path is added, if it creates a closed loop with previous definitive borders, it creates a new patch and the mesh is updated consequently. A new color is randomly chose in order to visually underline the different segments that the algorithm is creating. The third topology constraint is checked by a cyclical ordering around the endpoints of the new definitive path, that is, candidate paths that share an endpoint with the new path must start in the same segment on both meshes. For example, if on the source meshes the candidate path from 1 to 2 is in the segment limited by the definitive paths from 1 to 3 and from 1 to 4, the corresponding path should be in the segment limited by the two paths corresponding to the ones said above. If not, one of the paths is recomputed with this constraint. As in overlapping paths recomputing, paths that cannot be found after the constraint are removed from the candidate set.

When the set of paths is empty, the algorithm is over. If the patches obtained after the computation are triangular, we can start the mapping part of the morphing algorithm. If not, some steps of refinement are needed. If a patch isn't triangular it means that some of the vertices have no free edges on which construct a new path. So, triangularization of the patches is done creating *face paths*, meaning that a new border is created searching for a path over the faces and then creating new vertices on the mesh when the path can't step over the existing ones. The last image of Figure 3 shows the final result of the segmentation of the two meshes.

6. Conclusions and future work

We have presented here a suite of tools, developed upon a free library, to extract topological information from triangle meshes.

Regarding the skeletonization we have completely implemented the voxelization step, and we have implemented a skeleton extraction algorithm, without any post-processing refinement tool. Skeleton extraction still have to be further developed. We only have reached some partial result, and a first approximation of the skeleton has been successfully extracted from the mesh, but there are still some issues to face: in most cases the skeleton is not thin and it is not completely connected. The implementation of a refinement process is, thus, the first thing to do. Another unsolved question is related to how formatting the output of the algorithm: a graph-based skeleton description, or an approximation with equations or splines are the possible choices.

Regarding the segmentation we have completely implemented the fuzzy clustering algorithm and we are trying to speed it up using graph cuts to accelerate the APLP step. Initial results are very promising and we plan to be able to work even on larger meshes using such acceleration techniques. In the morphing field we are planning to change the strategy of triangularization still for speeding up the shortest paths identification. Another thing to be done is the actual morphing between the two meshes which we consider a relative easy task to tackle.

All the meshes used for the experiments were created with OpenMesh.

Acknowledgements

This work has been partially financed by the PRIN grant 3-shirt, (n. 2006010149_003, year 2006) of the Italian Ministry of University and Research. We are indebted to Leif Kobbelt and Ian Moebius of the Computer Graphics & Multimedia Group of the University of Aachen for helping us in becoming familiar with OpenMesh.

References

[AA93] ARMAN F., AGGARWAL J. K.: Model-based object recognition in dense-range images: a review. *ACM Comput. Surv. 25*, 1 (1993), 5–43.

[AKM*06] ATTENE M., KATZ S., MORTARA M., PATANE G., SPAGNUOLO M., TAL A.: Mesh segmentation - a comparative study. In *Proc. Shape Modeling and Applications 2006 (SMI'06)* (June 2006), IEEE Computer Society Washington, DC, USA, pp. 7–18.

[Blu67] BLUM H.: A transformation for extracting new descriptors of shape. In *Proc. Models for the Perception of Speech and Visual Form* (Nov. 1967), pp. 362–380.

© The Eurographics Association 2008.

[CSM07] CORNEA N. D., SILVER D., MIN P.: Curve-skeleton properties, applications, and algorithms. *IEEE Transactions on Visualization and Computer Graphics 13*, 3 (May/June 2007), 530–548.

[Dij59] DIJKSTRA E. W.: A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271.

[EK72] EDMONDS J., KARP R. M.: Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM 19*, 2 (1972), 248–264.

[GS99] GAGVANI N., SILVER D.: Parameter-controlled volume thinning. *Graphical Models and Image Processing 61*, 3 (May 1999), 149–164.

[HJBJ*96] HOOVER A., JEAN-BAPTISTE G., JIANG X., FLYNN P., BUNKE H., GOLDGOF D., BOWYER K., EGGERT D., FITZGIBBON A., FISHER R.: An experimental comparison of range segmentation algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 18 (1996), 673–689.

[JMF99] JAIN A. K., MURTY M. N., FLYNN P. J.: Data clustering: a review. *ACM Comput. Surv. 31*, 3 (1999), 264–323.

[KS04] KRAEVOY V., SHEFFER A.: Cross-parameterization and compatible remeshing of 3d models. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 861–869.

[KSG03] KRAEVOY V., SHEFFER A., GOTSMAN C.: Matchmaker: Constructing constrained texture maps. *ACM Transactions on Graphics 22*, 3 (July 2003), 326–333.

[KT03] KATZ S., TAL A.: Hierarchical mesh decomposition using fuzzy clustering and cuts. *ACM Transactions on Graphics 22*, 3 (July 2003), 954–961.

[OFCD02] OSADA R., FUNKHOUSER T., CHAZELLE B., DOBKIN D.: Shape distributions. *ACM Transactions on Graphics 21*, 4 (Oct. 2002), 807–832.

[Ope] OpenMesh. **www.openmesh.org**.

[Pet02] PETITJEAN S.: A survey of methods for recovering quadrics in triangle meshes. *ACM Comput. Surv. 34*, 2 (2002), 211–262.

[Qt] Qt 4.3 Whitepaper. **www.trolltech.com**.

[SLSK07] SHARF A., LEWINER T., SHAMIR A., KOBBELT L.: On-the-fly curve-skeleton computation for 3d shapes. *Computer Graphics Forum 26*, 3 (Sept. 2007), 323–328.

[SMD*05] SHOKOUFANDEH A., MACRINI D., DICKINSON S., SIDDIQI K., ZUCKER S. W.: Indexing hierarchical structures using graph spectra. *IEEE Transactions on Pattern Analysis and Machine Intelligence 27*, 7 (July 2005), 1125–1140.

[ZT99] ZHOU Y., TOGA A. W.: Efficient skeletonization of volumetric objects. *IEEE Transactions on Visualization and Computer Graphics 5*, 3 (1999), 196–209.

© The Eurographics Association 2008.