2013

Automatic Segmentation Of Teeth In Digital Dental Models

David Anton Mouritsen
University of Alabama at Birmingham

AUTOMATIC SEGMENTATION OF TEETH IN DIGITAL DENTAL MODELS

by

DAVID A. MOURITSEN

CHUNG H. KAU, COMMITTEE CHAIR
ANDRE FERREIRA
JOHN K. JOHNSTONE
CHRISTOS VLACHOS

A THESIS

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Master of Science

BIRMINGHAM, ALABAMA

2013

AUTOMATIC SEGMENTATION OF TEETH IN DIGITAL DENTAL MODELS

DAVID A. MOURITSEN

CLINICAL DENTISTRY

ABSTRACT

Introduction: Within the last decade, there has been an increase in the use of digital records in orthodontics. An orthodontic diagnostic setup is a simulated treatment outcome prepared from dental models. To create a diagnostic setup from digital dental models, the first step is to segment the models, or identify the boundaries of each tooth. In the past, a human operator would manually segment the models using a computer program to draw the boundary that separates the teeth from each other and the gingiva. The ideal, fully automatic, segmentation algorithm would not require a user to interact with the digital model. The purpose of this research is to design, develop, and test a computer algorithm to automatically segment digital dental casts and to release the project as open-source software to stimulate future research. **Methods:** A tooth segmenting algorithm was conceptualized and implemented as a C++ plug-in for Meshlab, an open source system for the processing of 3D triangular meshes. The C++ source code for the project was written using Qt Creator 2.2.1. The only input required for the algorithm, other than the model file, is the number of teeth for which to search. As test cases for the algorithm, six maxillary and six mandibular OrthoCad models were converted from .3DM to .STL format. The accuracy of the segmentation, measured by visual inspection, and the runtimes for each test model were recorded. **Results:** The 12 models (6 mandibular and 6 maxillary) had 160 teeth total to segment. The segmentation algorithm correctly segmented 133 of the 160 teeth for an 83.13% success rate overall.

The algorithm correctly segmented 70 of 80 maxillary teeth for an 87.5% maxillary success rate, and 63 of 80 mandibular teeth for a 78.8% mandibular success rate. The average runtime was 32 seconds on a 3.0GHz Intel® CoreTM2 Duo CPU with 4 GB RAM. **Conclusions:** With overall segmentation success at 83.13% and average runtime of 32 seconds, future research will focus on improving accuracy and speed, while maintaining usability. More research needs to be done before efficient, fully-automated segmentation of digital dental casts becomes a reality.

Keywords: segment, segmentation, dental, model, cast, algorithm

ACKNOWLEDGMENTS

I extend thanks to Dr. Chung H. Kau, the Orthodontic Department Chair; to Dr. Christos Vlachos, the Orthodontic Department Program Director; and to the members of my thesis committee for their time and guidance. Also, I thank all the faculty and residents in the Orthodontic Department at the University of Alabama School of Dentistry in Birmingham, Alabama.

Special thanks go to Lauren, my beautiful wife, for her never-ending optimism, encouragement, and support. In addition, thank you Anders, Hyrum, Jedidiah, and Julie for making me laugh and inspiring me with your imaginations and creativity.

TABLE OF CONTENTS

Page

APPENDICES

LIST OF FIGURES

INTRODUCTION

Every day you easily recognize thousands of objects by sight: a stapler, a car, a mug, a phone, etc. Donald Hoffman et al. described the complexity of the visual recognition process as follows: "Billions of neurons labor in concert to transform, step by step, the shower of photons hitting each eye into recognized objects. . . . The ease of recognition, like the ease of an Olympic skater, is deceptive. Recognizing [objects] from photons is no small task."[1] This truth becomes apparent when attempting to program a computer to automatically recognize objects. For example, even a young child can easily and quickly identify teeth on a dental model; as digital dental models have become more prominent in dentistry, researchers have spent many years trying to replicate this feat of recognition using computer algorithms.

Up until the 1990s, all dental and orthodontic study models were physical casts poured in dental stone.[2] In 2002, 6.6% of orthodontists used digital study models for initial records and 2.7% used them for diagnostic setups.[2] In 2008, 18% of orthodontists used digital study models for initial records and 2.5% used them for diagnostic setups.[2] The American Board of Orthodontics (ABO) now allows digital pretreatment models in the board certification process.[3] In addition, the ABO Directors are actively pursuing the implementation of an automatic scoring system for all digital casts.[3] There are many companies today that offer digital dental model services, such as: GeoDigm, Incognito, Invisalign, Motion View Software, OrthoCad, OrthoPlex, and Suresmile.[4-10] These companies are quick to advertise the benefits of digital models: they are easier to store, to

retrieve, and send than their physical counterparts. Although many orthodontists still use physical study casts today, the trend is clear: each year more clinicians are embracing the benefits of digital dental models.

Even though more orthodontists are using digital models each year, only a small percentage are using the digital models for diagnostic setups.[2] An important treatment planning step in many orthodontic cases is a diagnostic setup, or a treatment simulation performed by arranging the teeth in the ideal post-orthodontic treatment position.[11] Diagnostic setups using physical models are performed by carefully sawing each tooth from the cast by hand.[11] Then, the teeth are arranged in wax in an ideal position. If different extraction patterns are in question, multiple setups are needed. This process is time consuming and labor intensive. Due to the limitations of current digital model technology; a significant amount of time and cost are still required to perform a digital diagnostic setup. Once the process becomes cost effective and efficient, orthodontists will be more likely to use digital models for simulations of treatment.

The first step in a digital diagnostic setup is segmentation, or the process of separating the teeth from the gums and from each other. Most digital model companies restrict the segmentation of digital models to proprietary digital laboratories. The process is expensive, tedious and time-consuming because human operators are employed to manually segment the digital teeth on the models. In contrast, Motion View Software has developed proprietary software that allows the orthodontist to segment digital models in-office by clicking on each tooth.[7] Although this is a significant improvement over past segmentation algorithms, the method still requires significant user interaction with the model.

The ideal, fully automatic, segmentation algorithm would not require a user to interact with the digital model. Preferably a clinician, or researcher, would batch process thousands of casts and easily gain access to useful information such as the arch form, arch length, or Bolton tooth size discrepancy. At the time of writing, no such software exists and there are relatively few published articles that describe automated techniques to segment digital dental casts. The source-code used in nearly all published research papers is proprietary and not available for review, even though some now suggest that "anything less than the release of source programs is intolerable for results that depend on computation."[12] To date there is no open-source implementation of a tooth segmenting algorithm. The purpose of this study is to design, develop, test, and release an open-source implementation of a novel, automated tooth segmenting algorithm. Special emphasis is placed on minimal user interaction with the digital model.

Segmentation in General

The segmentation of 3-dimensional digital models, or the partitioning of these models into meaningful parts, is a critical step in many computer-aided design, manufacturing, and engineering applications.[13] There are many different approaches to segmenting 3-dimensional digital objects, and each method has advantages and disadvantages. A method that works well to segment one type of digital model may fail completely when used on a different type of model. In 2007 A. Agathos et al. organized 3-dimensional digital segmentation methods into the following categories: region growing, watershed-based, reeb graphs, model-based, skeleton-based, clustering, spectral analysis, explicit boundary extraction, critical points-based, multiscale shape descriptors,

markov random fields and direct segmentation.[13] The most common criterion upon which many segmentation methods are based is the negative minima rule, developed by Hoffman and Richards[14], which states that objects are segmented by human perception at regions of concavity.[13]

Negative Minima Rule

The negative minima rule is based on the principle of transversality.[1] D. Hoffman explained transversality as follows: if two separate objects are joined together to form a new single object, the boundary between the old objects is most likely found on a concave crease.[1] The edges of a cube are considered convex creases, while concave creases go into an object.[1] The negative minima rule is an extension of transversality to objects that do not have sharp creases, but have smooth curves. In those cases, the division between parts can be found along the negative minima of the principal curvatures, along their associated lines of curvature.[1] The negative minima rule is important when identifying one object that extrudes from another, for example, when fingers protrude from a hand. It is, therefore, no surprise that many of the published algorithms for the segmentation of teeth rely on the negative minima rule to identify teeth extruding from gingiva on digital models.

Review of Published Tooth Segmenting Algorithms

In 1994 M. Mokhtari et al. created an algorithm to automatically process the imprints of teeth in a wax bite registration.[15] They used a watershed algorithm to detect features of the bite mark such as cusp tips and incisal edges. The algorithm also identified the boundary between teeth in the wax registration. The wax imprints were not an accurate representation of the 3-D anatomy of the teeth and gums; therefore, the algorithm was not designed to identify the boundary between the teeth and gums.

In 2004 T. Kondo et al. propose a method to segment digital dental study models by using 2-D plan-view and panoramic range image projections of the 3-D model.[16] Although this method avoids some challenges of directly processing the 3-D triangular mesh, user interaction is required to manually orient the models and to correct segmentation errors. The algorithm produces poor results in the second molar region in both maxillary and mandibular models, and in cases in which teeth are overlapped, for example, in cases of severe anterior crowding.

In 2005 M. Zhao et al. proposed an interactive tooth segmenting algorithm.[17] First, the user orients the digital model by identifying four reference points. Next, curvature estimates are calculated and the negative minima rule applied to identify the initial feature region. Then, the user deletes unnecessary feature points by drawing boundary lines on the model and by clicking to identify the borders between each tooth. Although this algorithm correctly handles cases of severe malocclusion, it requires more user interaction than previous approaches.

In 2008 Y. Tian-ran et al. described a similar algorithm that required less user interaction.[18] The first part of the algorithm is automated and consists of the following:

estimate curvature of models, extract feature region, remove artifacts, and apply mathematical morphological operations. Then, a user interacts with the model and bridges gaps in the feature region manually. After further processing of the feature region, the segmentation process is complete. The authors concede that future work is required to "remove the user interaction to make the segmenting progress automatically."[18]

In 2010 H. Yau et al. published a paper describing a system to segment digital dental models, align the teeth digitally, and fabricate clear aligners to carry out the desired movements.[19] The segmentation process is not the main emphasis of the paper. Like many previous studies, they method the authors describe relies on automated curvature estimation and feature region extraction. To overcome gaps in the feature region, a user paints on the model with a brush tool. Further processing reduces the feature region to a thin line, and after a region growing step the segmentation is complete. The segmentation method presented in this paper is comparable in efficiency and the required amount of user interaction to the algorithm published in 2008 by Y. Tian-ran.

Also in 2010 N. Wongwaen described a fully-automated tooth segmenting algorithm.[20] The first step in the algorithm is detecting the occlusal plane and approximating the arch form. Next, a 3-D panoramic projection is created from the 3-D model, and then converted to a 2-D panoramic projection. The 2-D panoramic projection is analyzed to identify boundaries between teeth. Then, the 2-D cutting points are mapped back to the 3-D original model. The algorithm only attempts to identify the boundary between teeth, while disregarding the boundary between tooth and gums.

Although the algorithm is completely automated, it frequently divides molars in half and fails to correctly segment overlapping teeth.

In 2010 T. Kronfeld presented a method of segmentation of digital dental casts that uses active contour models, or snakes.[21] First, the curvature of the model is estimated. Then, the negative curvature is used to form a plane representing the boundary between tooth and gum. The plane is then converted into one or more snakes. The boundaries between teeth are discovered by using edge tangent flow. Individual snakes are then started at the crown of each tooth and each tooth is separated from bordering teeth and gums. The values for curvature thresholds are user supplied. When the boundary between the tooth and gums is smooth, or when neighboring teeth overlap each other because of severe malocclusion the segmentation fails.

In 2011 Yokesh Kumar published a paper detailing a near fully-automated algorithm for the segmentation of digital dental models.[22] First, the teeth are segmented from the gums in a technique using the negative-minima rule. The curvature threshold value is determined by user input; the user adjusts a slider while receiving feedback on the model. Then, the teeth are segmented by detecting and connecting corresponding interstitial points on the lingual and labial gumline. The algorithm results in a jagged, inaccurate representation of the tooth gum border, and the border between teeth appears too wide to preserve the original anatomy of each tooth. Also, noise results in areas of the palate being recognized as teeth; these areas must be corrected manually by the user.

In 2011 D. Brunner and G. Brunnett published a paper describing a method using vector fields to close gaps in feature regions to improve the segmentation of digital models.[23] Some of the benefits of the authors' approach are that the segmentation is not

limited to the minima rule, there is no need to change feature regions into feature lines, and the method can be applied to a wide range of model types. The emphasis of the paper was on the segmentation of mechanical parts, but the implications for the segmentation of digital dental models were stated. Errors in the mathematical definitions in the paper were identified which preclude an independent implementation of the algorithm. The authors stated in an email that they would release a revised paper with the corrections.

MATERIALS AND METHODS

A novel algorithm for segmenting teeth on digital dental models was conceptualized and implemented as a C++ plug-in for Meshlab, "an open source, portable, and extensible system for the processing and editing of unstructured 3D triangular meshes."[24] The algorithm combines ideas from two different approaches: Yuan Tian-ran et al.'s "Bio-Information Based Segmentation of 3D Dental Models"[18] from 2008 and D. Brunner and G. Brunnett's "Closing Feature Regions"[23] from 2011. The C++ source code for the project was written using Qt Creator 2.2.1, a cross-platform integrated development environment (IDE).[25] The only input required for the algorithm, other than the .stl model file, is the number of teeth for which to search. A flowchart overview of the algorithm is displayed in Figure 1.

The first step in the algorithm is to estimate the curvature at each vertex in the model. Next an upper boundary of minimum curvature is picked, and an initial feature region is defined. Morphologic operators are used to convert the feature region into a feature line. Then, a distance field is calculated and, small gaps in the feature region are automatically filled. Finally, after small artifacts are removed, the segmentation is achieved by region growing. The number of teeth segmented is compared to the user-input number of teeth, and a new upper boundary of minimum curvature is picked. The algorithm stops when the number of teeth found is closest to the number input by the user.

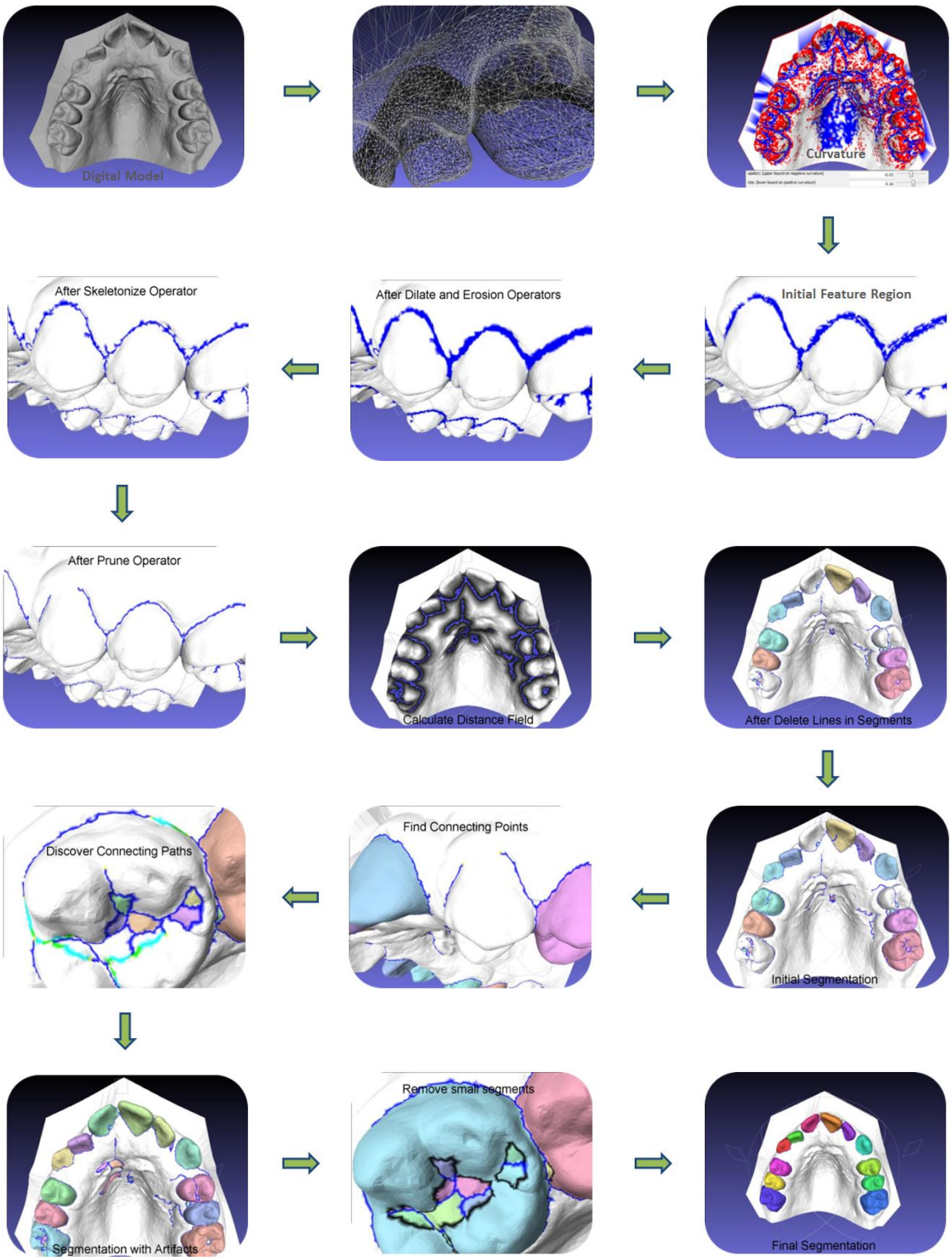

Figure 1. Flowchart of tooth segmentation algorithm.

Mathematical Definitions

In order to communicate clearly about specifics of the algorithm, mathematical definitions are needed to describe the triangle mesh, vertices, edges, the neighbors.[21, 23] A triangle mesh, *M*, is represented by the tuple $M=(V,T)$ where V is the indexed set of vertices and T is the indexed set of triangles. The indexed set of vertices $V=\{v_i : i=1,\ldots,n_V\}$ where each vertex, v_i, represents a unique position in E^3. The indexed set of triangles $T=\{t_j : j=1,\ldots,n_T\}$ and $\forall t \in T: t=\{(v_q, v_r, v_s) \mid v_q, v_r, v_s \in V; v_q \neq v_r \neq v_s\}$. The set of edges $\mathcal{E} =\{e_k : k=1,\ldots,n_E \mid e_k=(v_x, v_y)\}$ where $v_x, v_y \in V; v_x \neq v_y$. The set of neighbors of v, $N(v)$, is comprised of each vertex $n \in V$ and $v \in V$ iff the edge $e=(n,v) \in \mathcal{E}$. The set of n-neighbors of a vertex is defined as the set $N^n(v)$.

Curvature and Feature Region

The curvature at each vertex $v \in V$ is estimated and stored as $k_h(v)$ by using Meshlab's implementation of Mark Meyer's algorithm published in 2002: "Discrete Differential-Geometry Operators for Triangulated 2-Manifolds".[26] Because the range of the curvature values varies widely, the curvature values are mapped from $]-\infty, \infty[$ to $]-1, 1[$ using R. Vergne et al.'s scaling function.[21, 27] The scaling function is defined as follows: $\zeta(x) = tanh(x(K/\alpha)$, with $K = tanh^{-1}(2^B-2/2^B-1)$ where B controls the precision in the number of bits and was set to 8.[27] Curvature threshold variables are defined as follows: ε is the upper bound on the negative curvature and η is the lower bound on the positive curvature. The feature region, *F*, or the set of vertices that separate tooth from gingival and teeth from each other, is defined as follows: $F = \{v \in V \mid k_h(v) < \varepsilon\}$.[21] The feature region can be thought of as a disjoint set because there are patches of vertices on

the digital casts where $k_h(v) < \varepsilon$.[21] Sets that are less than .01 of the total number of vertices in the feature region are treated as unwanted artifacts and deleted. The feature region changes for different values of ε. Even with the best choice of ε, the feature region has gaps and includes extra, unwanted vertices. Major themes in the algorithm are to automatically choose the best ε, remove undesirable artifacts and patch gaps in the feature region.

In Figure 2 the color blue represents negative curvature below ε and the color red represents positive curvature above η. The curvature boundaries in Figure 2 are set as follows: ε = -.02 and η = 0.16. Notice that the blue color outlines the teeth and separates each tooth, and separates the teeth from the gingiva. However, at this threshold value, too much of the palate and occlusal surfaces of the teeth are colored blue to be useful for segmentation. The artifacts, coupled with variations in anatomy, are the main reason why the automatic segmentation of teeth is so difficult. The palatal rugae, for example, have negative and positive curvature that must be identified as non-tooth. When the curvature thresholds values are ε = -.60 and η = 0.60, observe in Figure 3 that there are no extra artifacts. Unfortunately, there are large gaps in the feature region.

Many segmentation algorithms rely on a human user to adjust the curvature threshold values; this algorithm, however, uses trial and error to identify the best curvature threshold values automatically. If ε is too low, not enough teeth will be found. If ε is too high, then extra artifacts will be segmented that are not teeth. The best value of ε is defined as the lowest value for which the number of segments are equal to the number of teeth sought by the user.

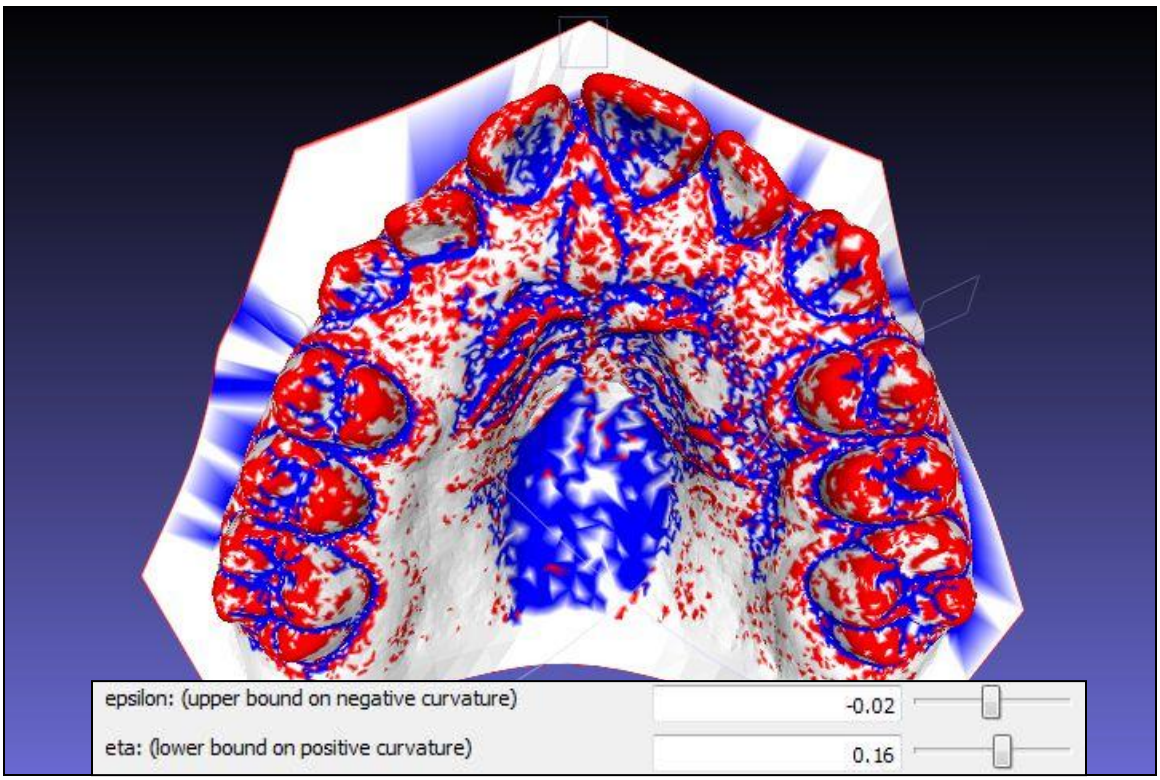

Figure 2. Curvature estimation with negative curvature threshold set too high.

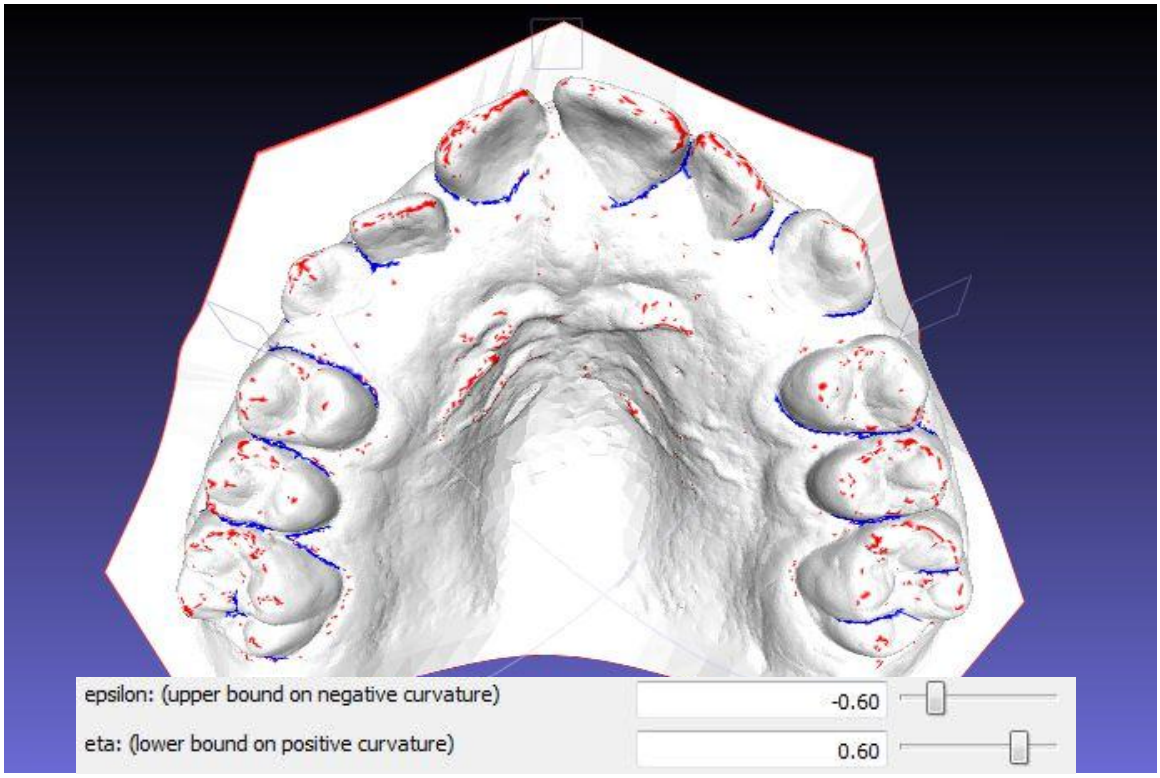

Figure 3. Curvature estimation with negative curvature threshold set too low.

Dilation and Erosion Morphologic Operators

Once a value for ε has been chosen, the next step is to use morphologic operators to clean up the feature region. Figure 4 demonstrates how jagged the feature region is when first defined; it has holes on the interior and bays on the exterior. Also, the feature region is several vertices wide—it's too thick to define an accurate segmentation of the teeth. A wide feature region makes identifying connecting points to close gaps more difficult.

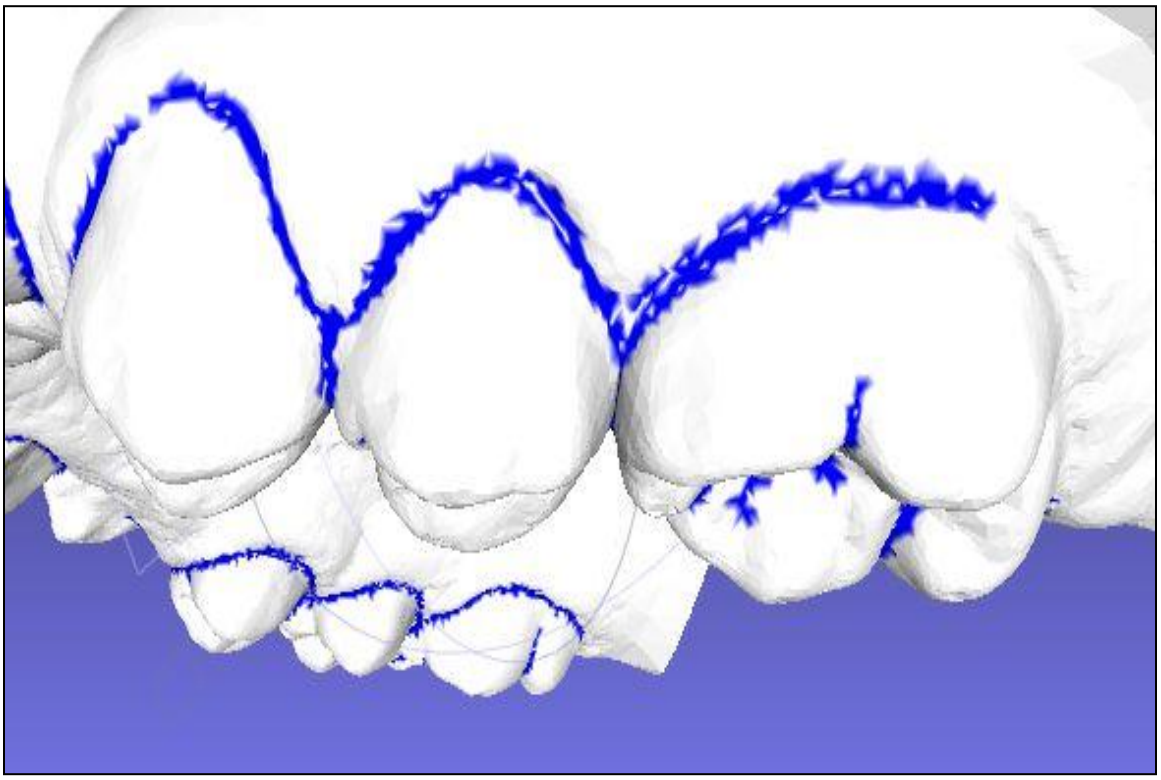

Figure 4. Feature region before morphologic operators.

To fill in the interior holes and exterior bays by growing, or dilating, the feature region while maintaining the general shape of the feature region we apply the dilate operator: $\text{Dilate}^n(F) := F \cup \{j \mid \forall v \in F: j \in N^n(v)\}$.[28] In other words, the neighbors of each vertex in

the feature region are checked to see if they are already part of the feature region. If a neighbor is found that is not a part of the original feature region, it is added to the set.

After the dilation operator, although the feature region has filled in the holes and bays, it has grown larger in size. To regain the general size of the original feature region, the erosion operator is applied: $\text{Erosion}^n(F) := \{j \mid N^n(j) \subseteq F\}$.[28] So, the neighbors of each vertex, *v*, in the feature region are checked to see if they are part of the feature region. If every neighbor of *v* is not found to be part of the feature region, then v is deleted from the set. See Figure 5 for the results of the dilation and erosion morphologic operators.

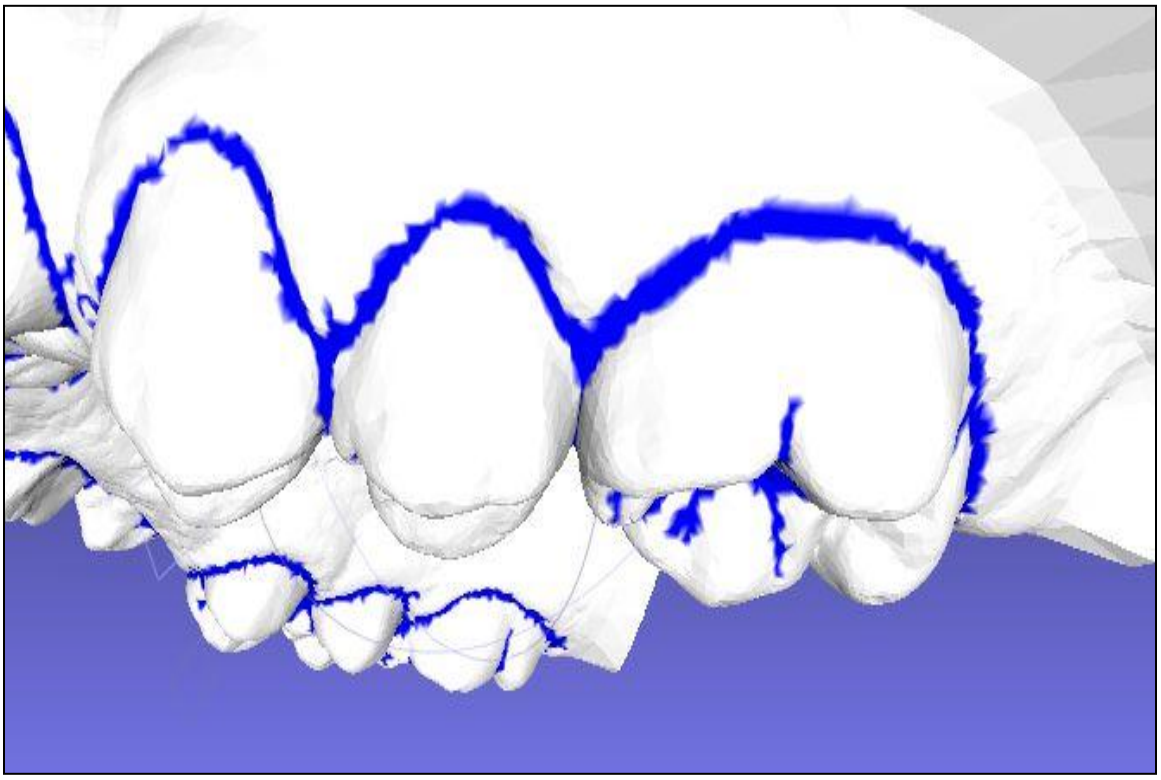

Figure 5. Feature region after dilate and erosion operators.

Skeletonize Morphologic Operator

At this point in the algorithm, the interior holes and some of the exterior bays of the feature region have been filled in. The feature region is still too course—the next step is to iteratively remove layers of the feature region until the feature region is reduced to a one vertex thick skeleton.[28] The feature skeleton must follow the topology of the original feature region.

Before defining the skeletonize operator, first consider the complexity of a vertex. The complexity, c, of a vertex $v \in F$ is defined by traversing the $N(v)$ in a clockwise direction and counting the transitions between vertices that are elements of F and vertices that are not.[28] A vertex is defined as *complex* iff $c \geq 4$.[28] A vertex that is in the middle of a feature line will have $c = 4$, while a vertex that is a node where two or more feature lines intersect will have $c > 4$.[28] The set of all complex vertices in a feature region is $C \subseteq F$.[28] The algorithm must not delete complex vertices, or else new gaps will be introduced in the feature skeleton.

A vertex $v \in F$ is defined as a *center* if every neighbor of v is also in the set F.[28] A *disc* is defined as the set of vertices that are neighbors to a center, not including the center.[28] $\bigcirc \subset F$ denotes the union of all disks and $\odot \subset F$ denotes the union of all centers.[28] The skeletonize operator is defined as skeletonize(F):= $F \setminus (\bigcirc \cap \overline{C \cup \odot})$.[28] The skeletonize operator iteratively removes the outer most layer of the feature region, while maintaining its shape. Each iteration removes vertices that are disks, but are neither centers nor complex. The skeletonize operator is finished when an iteration results in no change to the feature region.

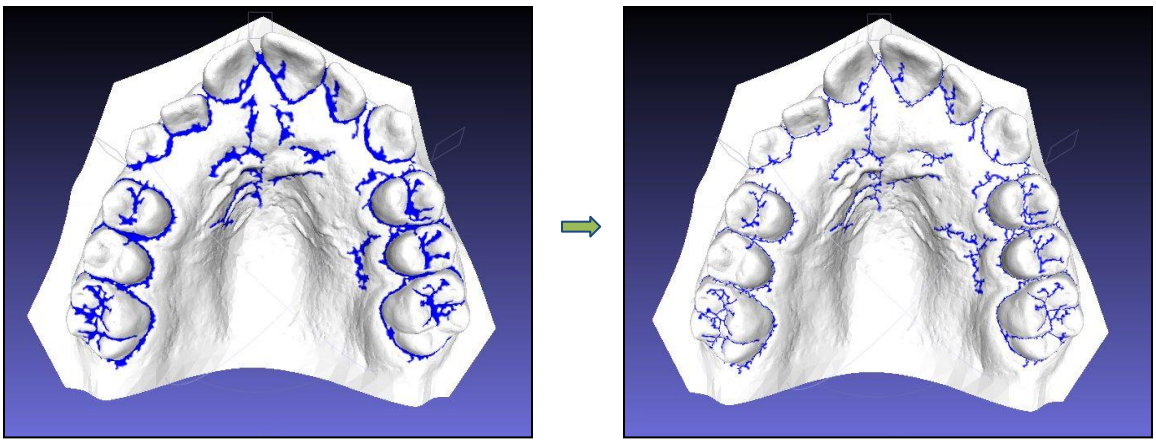

Figure 6. Feature region before and after skeletonize operator (occlusal view).

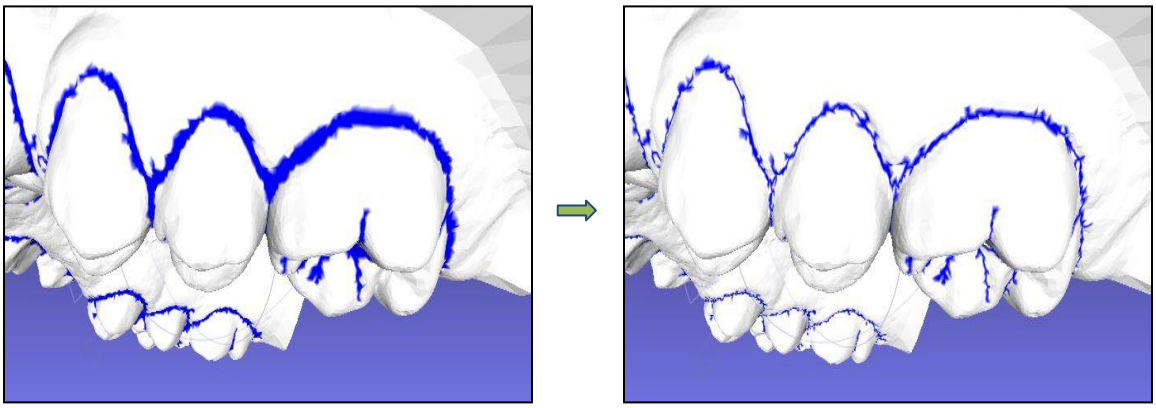

Figure 7. Feature region before and after skeletonize operator (buccal view).

Prune Morphologic Operator

At this point in the algorithm, the feature region has been reduced to a feature line. However, there are still many small branches that are undesirable artifacts. If we were to try to connect gaps in the feature at this point, too many undesirable connections would be made. Therefore, the next step is to prune, or remove, the small branches from the feature line.

The pruning operator iteratively shortens branches by one vertex per iteration. Define *S* as the feature skeleton, and *C* as the set of complex vertices. Remember,

complex vertices are found in the middle portion of a line (not the ends), or at a node where multiple lines intersect. The prune operator is defined as prune(S) = $S \setminus \overline{C}$.[28] Each iteration of the prune operator removes one vertex from the end of a branch. For this algorithm, the prune operator is applied ten times; this is sufficient to remove small branches without excessively widening gaps in the feature line that will need to be connected at a later step.

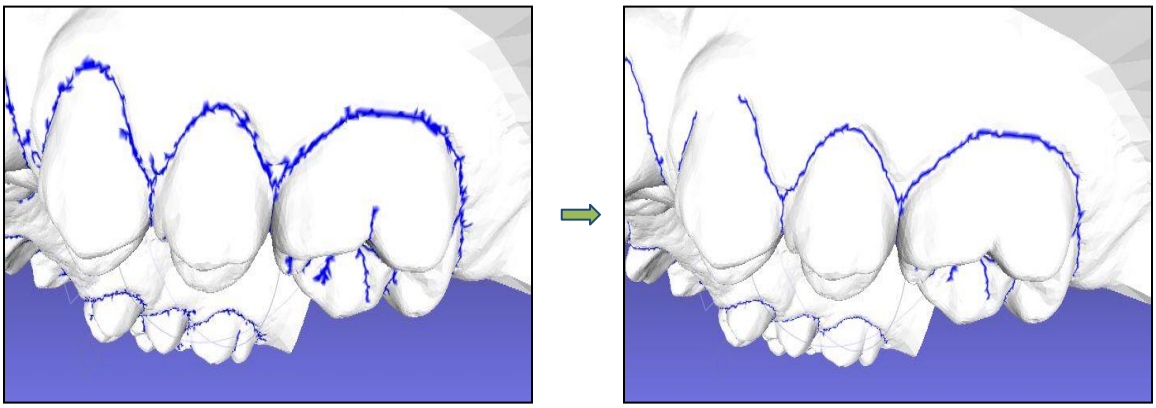

Figure 8. Feature skeleton before and after prune operator (buccal view).

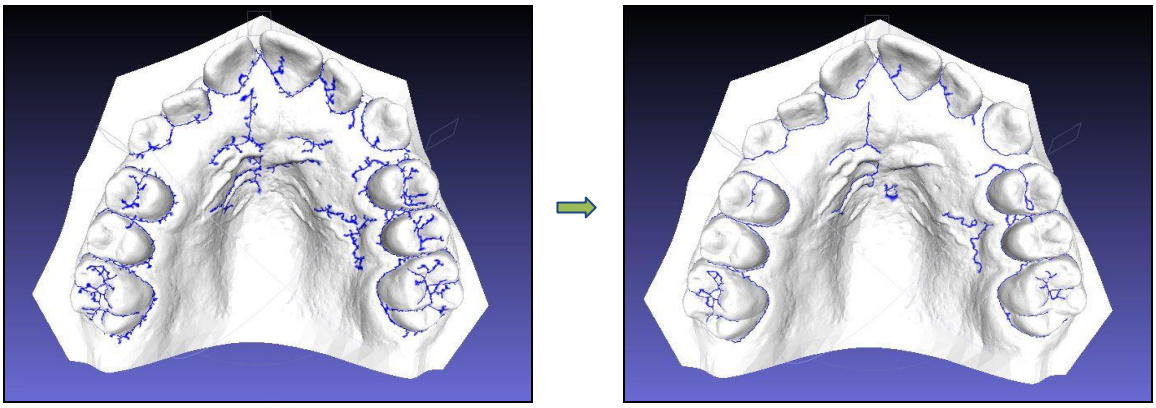

Figure 9. Feature skeleton before and after prune operator (occlusal view).

Initial Segmentation

At this point in the algorithm, the feature region has been transformed into a feature line and a significant number of undesirable artifacts have been removed. Using disjoint sets and region growing, regions that are completely enclosed by the feature line are colored distinct colors. This produces the initial segmentation. Notice in Figure 10 that several teeth were not segmented because there are gaps in the feature line. Also note the remaining undesirable artifacts that must be removed in the palate and inside of the tooth segments.

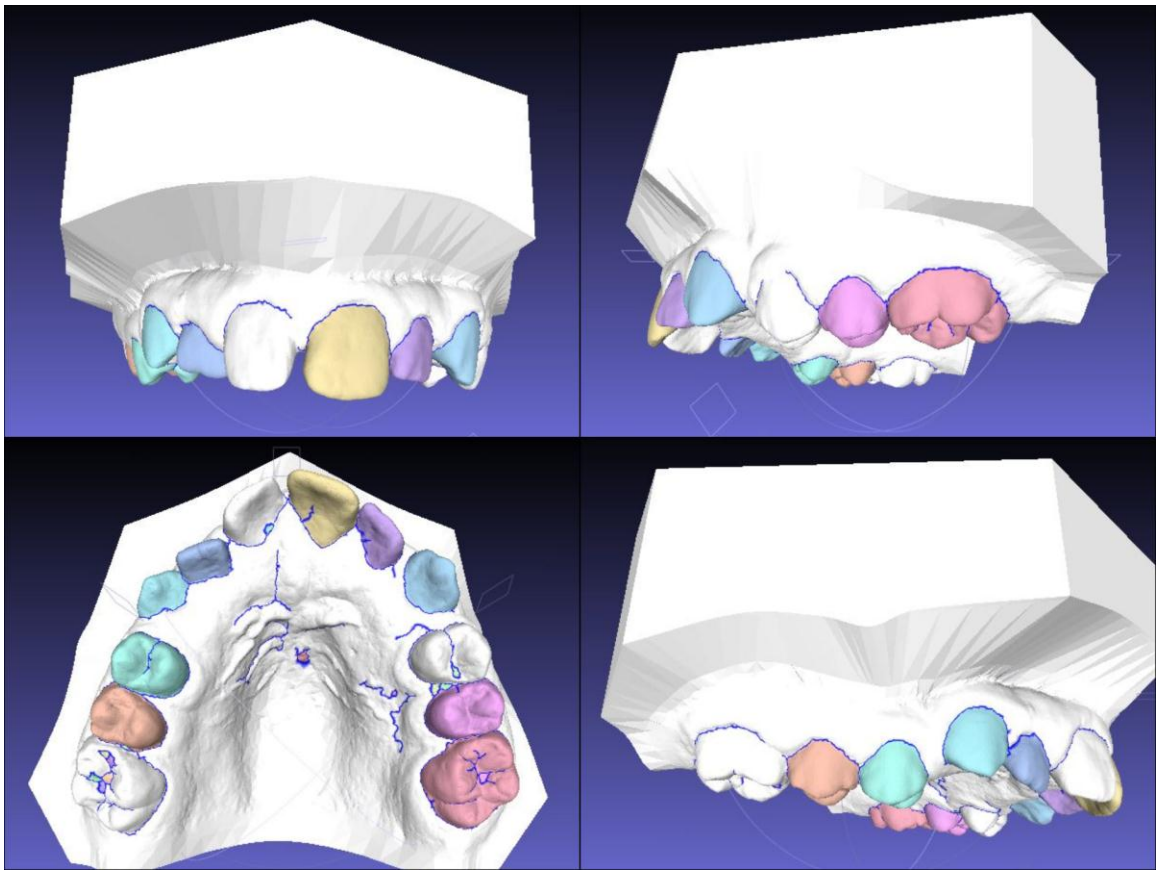

Figure 10. Initial segmentation after morphologic operators.

Delete Branches in Segments

The next major task of the algorithm is to close gaps in the feature region. In order to prepare for this task, as many undesirable artifacts as possible must be removed from the feature skeleton to prevent unnecessary connections. Despite the pruning morphologic operator, note that there are still branches of the feature skeleton inside of teeth that have been segmented. If those branches are not removed, the teeth will be over segmented.

In the initial segmentation, the model and unsegmented teeth are colored white, while segmented teeth are assigned unique colors. To remove the unwanted branches inside of segmented teeth, the color on both sides of every portion of the feature skeleton is checked. If the color is the same on both sides of a branch of the feature line and the color is not white, then the branch is removed from the feature line. If the color is white, the branch may be a portion of the feature skeleton that has a gap and is in need of repair. Unfortunately, it may also be an artifact in the palate or another unwanted branch jutting out from the feature skeleton. These artifacts will be addressed at a later point in the algorithm.

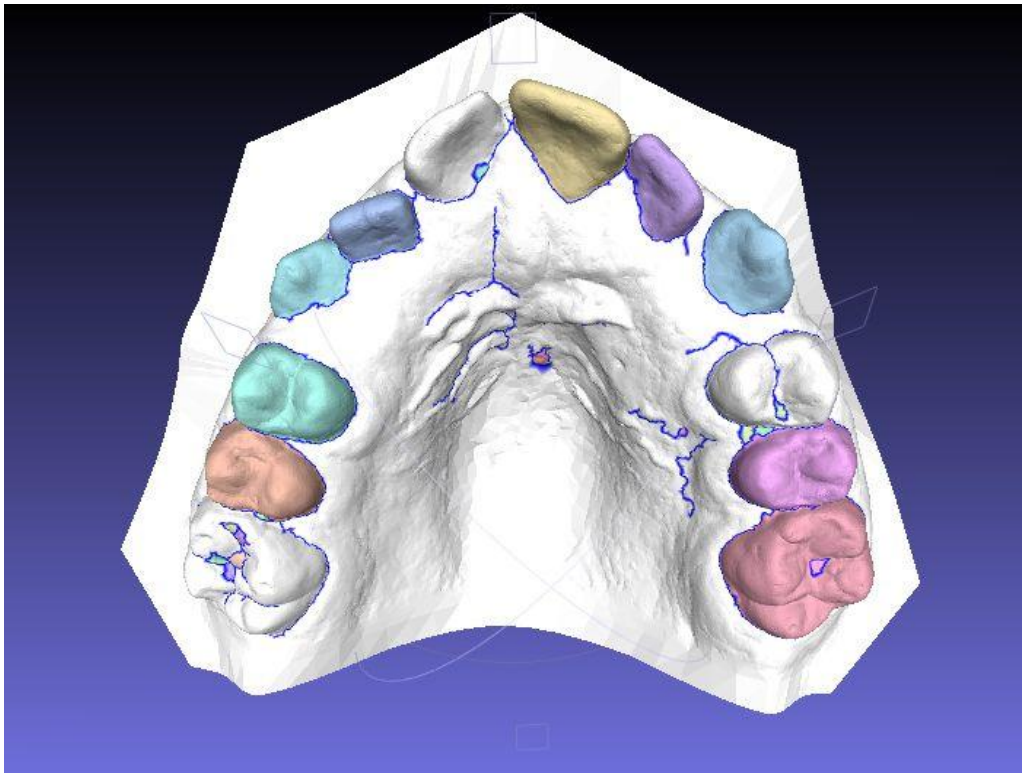

Figure 11. Initial segmentation before removing branches in segments.

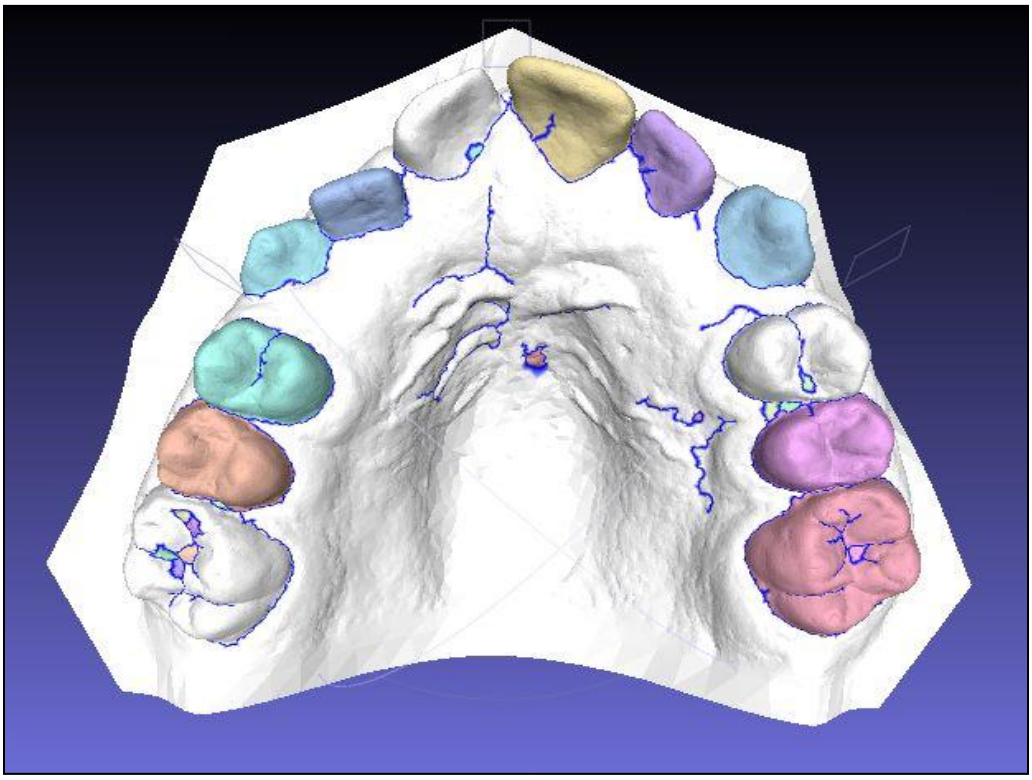

Figure 12. Initial segmentation after removing branches in segments.

Closing Gaps in the Feature Skeleton

The next step in the algorithm is to close gaps in the feature lines. However, we do not want to create a disproportionate amount of undesirable artifacts as vertices are added to the feature lines. Unfortunately, in this completely automated algorithm, it is difficult to restrict the feature line closing to just those areas that we want closed. Inevitably, undesirable connections will be made. In order to restrict the number of undesirable connections, a distance field of 3 mm around the feature line is defined and all closing operations are restricted to this limit.[23] This precludes, for example, connections from one side of the cast to the other and limits closing operations to small gaps in the feature line.

Calculating the Distance Field

The goal of the distance field is to store a distance value at each vertex that represents its distance from the feature skeleton. A path, *P*, between a vertex, *u*, and another vertex, *w*, is defined as $P = (u=v_1, \ldots, v_n=w)$.[23] The distance of a path is defined as $\delta(P) = \sum_{i=1}^{n-1} \|v_i - v_{i+1}\|$ where $\|\cdot\|$ is the Euclidean distance.[23] To calculate a distance field, all vertices that are part of the feature skeleton are assigned a distance of zero. Next, all other vertices are assigned a temporary distance of infinity. Vertices of the feature skeleton comprise the starting array, from which the distance search begins.

Vertices in the immediate neighborhood of the feature skeleton are explored iteratively. The distance value stored at a vertex *v* explored from a vertex *u*, where *v* is a neighbor of *u*, is the smaller of either 1) the current distance stored at the vertex or 2) the distance from *u* to the feature line plus the Euclidean distance between *u* and *v*:

$\delta(v) \leftarrow min(\delta(u) + \|u - v\|, \delta(v))$.[23] In every iteration, the vertex with the current minimum is explored. The distance field calculation is complete when the smallest distance to explore is over 3 mm.

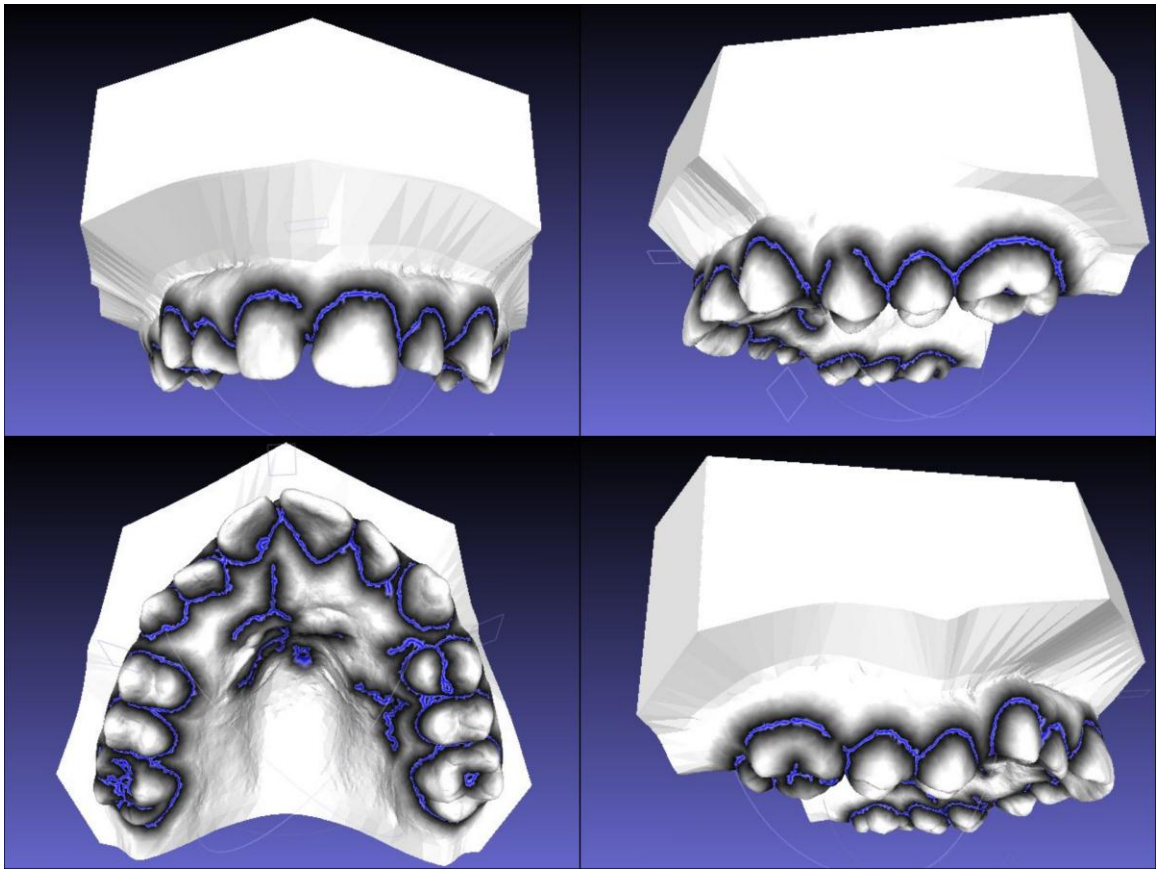

Figure 13. Representation of distance field calculated up to 3mm from feature skeleton.

Find Connecting Points

To bridge gaps in the feature skeleton, it is necessary to first identify connecting points. A connecting point is a vertex that is part of the feature that is closest to the gap. It would be difficult to identify connecting points in the feature region before the skeletonize operator. After the feature region has been transformed into the feature skeleton, though, connecting points are simply defined as non-complex vertices.

Remember the complexity, c, of a vertex v ∈ F is defined by traversing the $N(v)$ in a clockwise direction and counting the transitions between vertices that are elements of F and vertices that are not.[28] A vertex that is in the central part of a feature line will have c = 4, while a vertex that is a node where two or more feature lines intersect will have c > 4.[28] The connecting points, or the last vertex on a branch of the feature skeleton, will have c = 2.

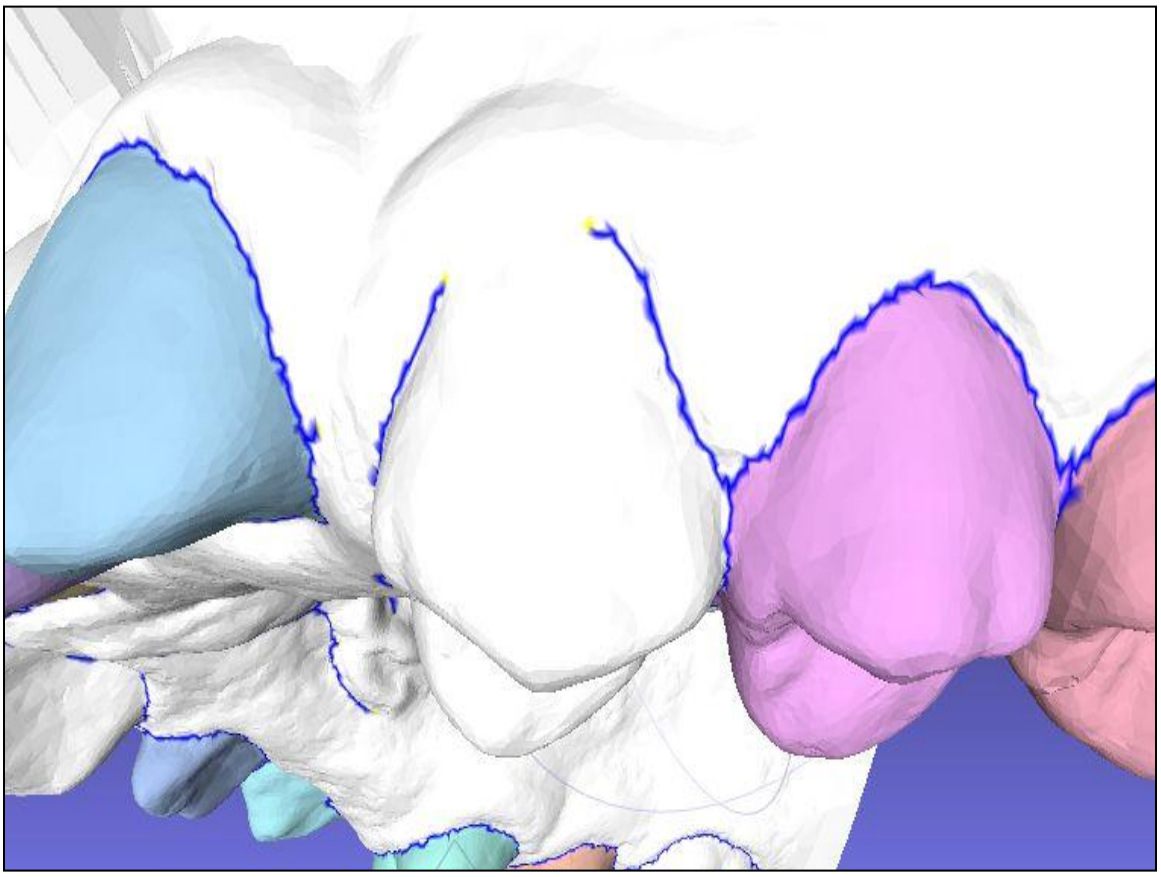

Figure 14. Connecting points represented by yellow vertices on either side of the gap.

In the pruning step, most of the small branches were removed. Then, small branches were deleted inside of teeth that have already been segmented. However, there are still small branches of the feature skeleton that are artifacts associated with teeth that have gaps in the feature skeleton. This means that some connecting points will be

identified that belong to artifacts. Figure 15 shows both types of connecting points: useful connecting points next to a gap in the feature skeleton, and connecting points on small artifact branches.

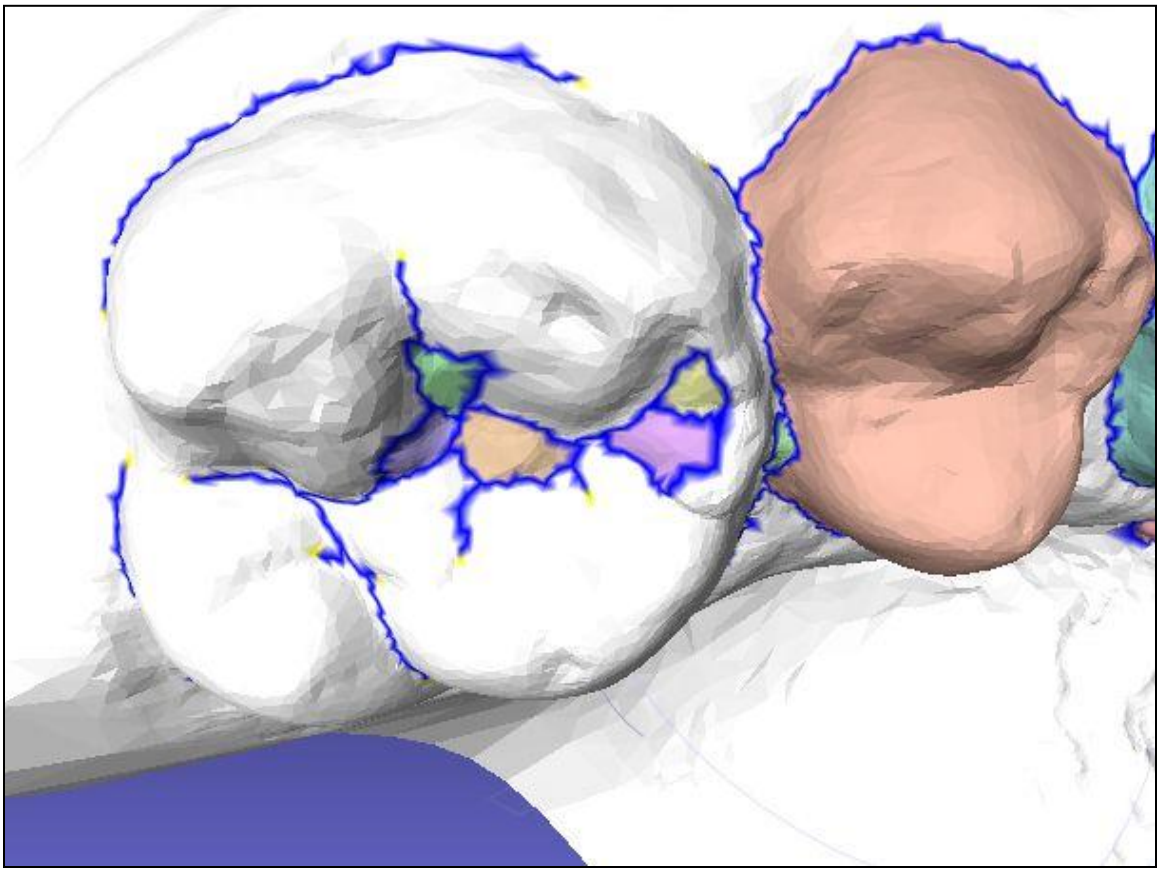

Figure 15. Connecting points represented by yellow vertices.

Connecting Paths Search

After the connecting points are identified, a simultaneous search is initiated at each connecting point to find the shortest path between two connecting points. First, the 1-ring neighborhood of each connecting point is explored. Next, the 2-ring neighborhood of each connecting point is explored. At each iteration of the search, data is stored in each vertex that identifies the shortest path back to the connecting point. When a vertex

is explored that has been visited previously by a search originating at a different connecting point, it is defined to be the midpoint of a new connecting path between the two connecting points. The search is confined to the distance field established previously.

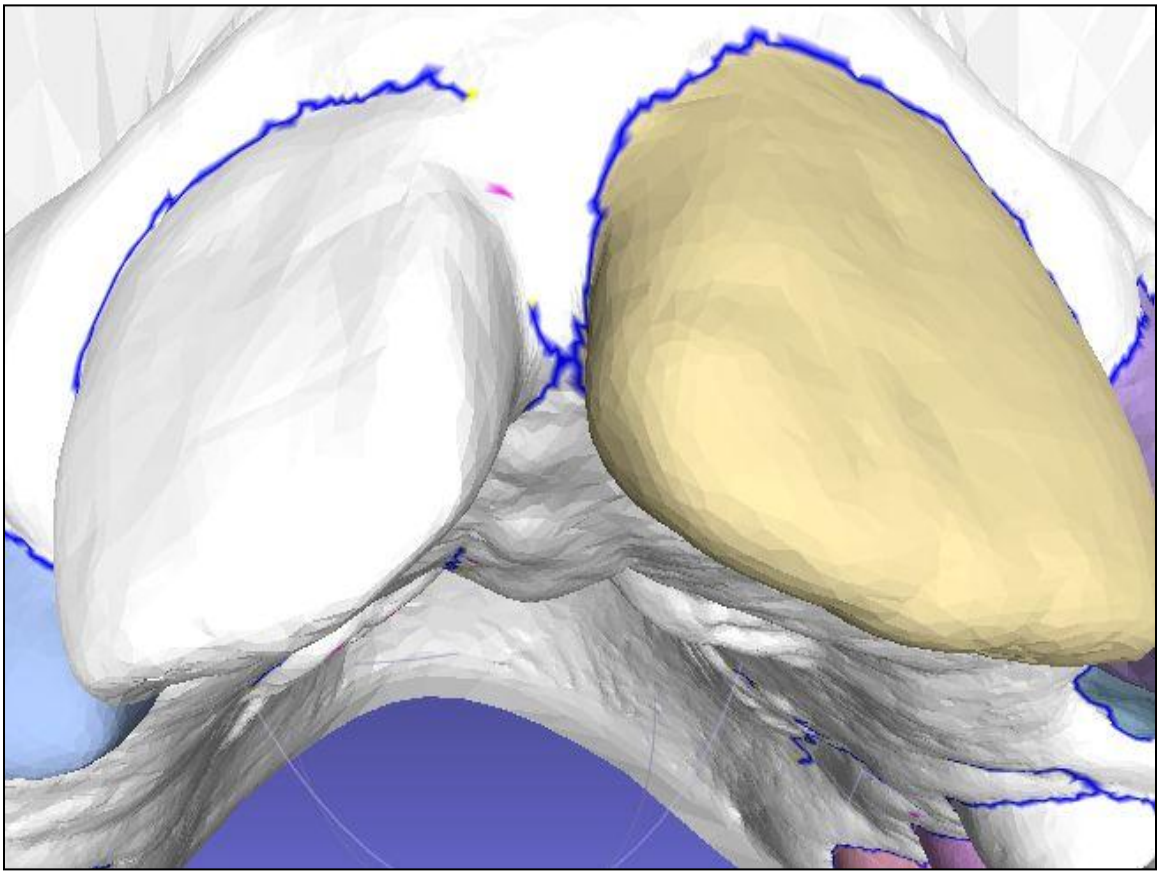

Figure 16. Midpoint represented by a purple vertex.

After the search for midpoints is performed, paths between connecting points are discovered by tracing the data stored in the vertices from the midpoints back to the corresponding connecting points. Figure 17 shows an example of a discovered path from the midpoint back to each of the originating connecting points. Then, the new paths are added to the feature line. Although many of the new paths are desirable as they close

gaps in the feature line, some may be undesirable connections that must be removed at a later step.

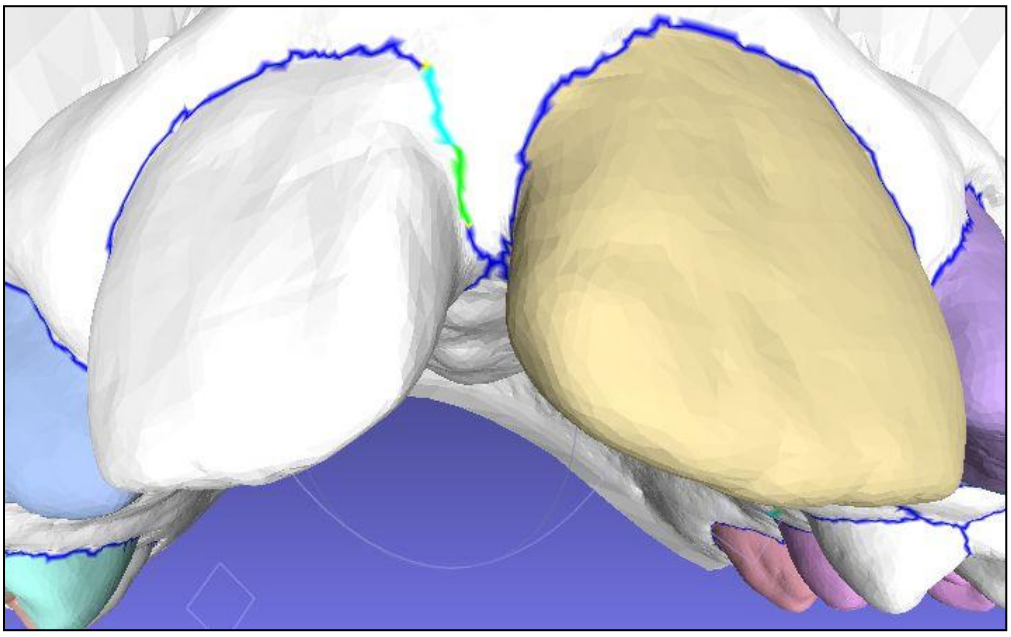

Figure 17. Paths from the midpoint to the connecting points represented by teal and green colored lines.

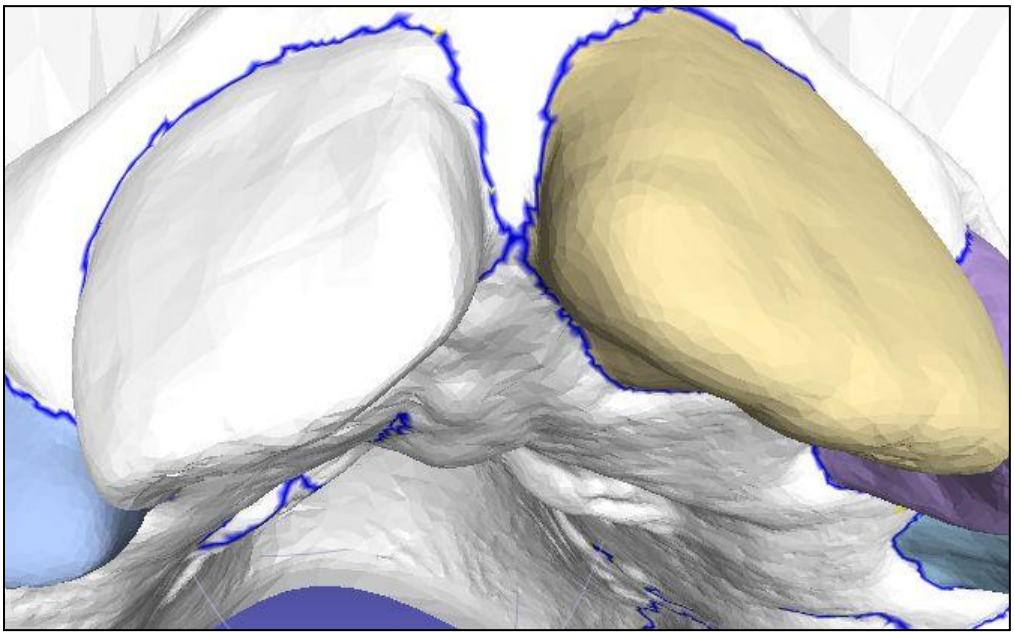

Figure 18. New path is added to the feature line.

Segmentation with Artifacts

Using disjoint subsets and region growing, regions that are completely enclosed by the feature line are colored distinct colors once again. Now, after gaps in the feature line have been closed, one can expect that additional teeth be segmented that were not found in the initial segmentation. Note the remaining undesirable artifacts consisting of lines and small segments, both inside and outside of the segmented teeth.

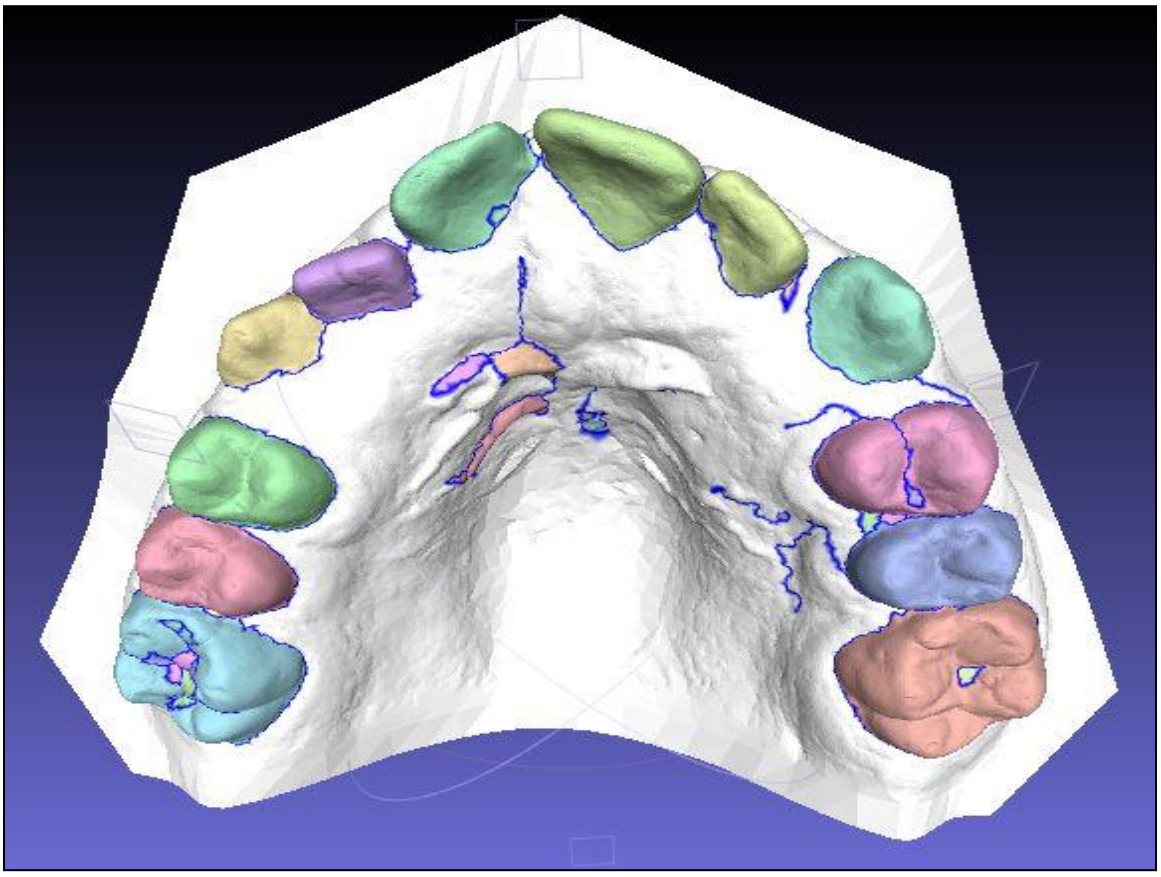

Figure 19. Segmentation after closing gaps in feature line with artifacts still present.

Removal of Artifact Lines

The first step in artifact removal is to delete unnecessary lines, or branches, of the feature line. Like before, the color is checked on both sides of all portions of the feature line. If the color is the same on both sides of a line (including the color white this time), the feature line is deleted because the line must be entirely inside a segmented tooth, or entirely outside of the segmented teeth. Because all gaps in the feature line have been closed, these lines are assumed to be unhelpful artifacts. The resulting segmentation is shown in Figure 19. Notice that there are still small artifacts both inside and outside of the segmented teeth.

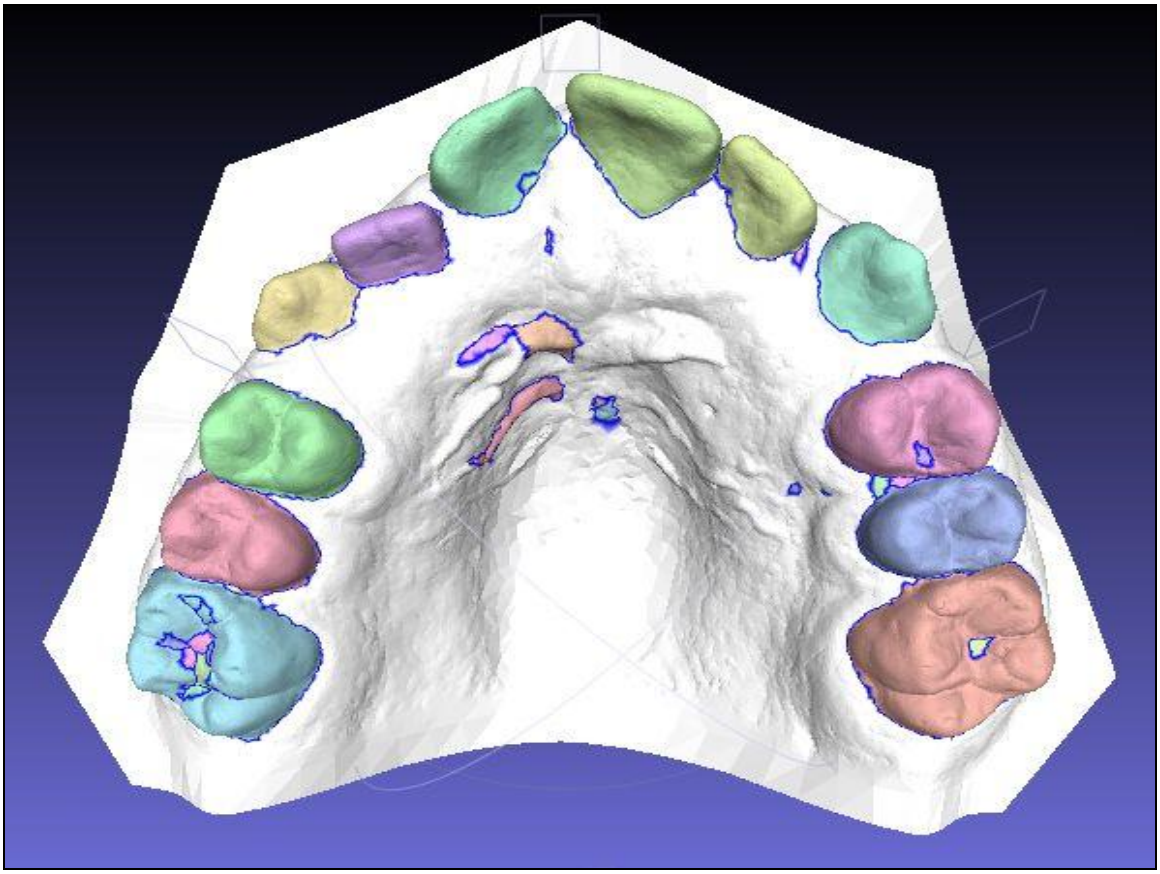

Figure 20. Segmentation after removal of artifact branches of the feature line.

Removal of Small Artifact Segments

The final step in the segmentation process is to remove small segmentation artifacts, both inside and outside the teeth. This is done by assessing the size of all of the segments. We define a segment to be *small* if it is less than ¼ the size of the second biggest segment. The white cast is the biggest segment and the second biggest segment is the biggest tooth. Small segments that are not touching other segments are easily removed. However, care must be exercised when deleting small segments that are on the border of a tooth—one certainly does not want to open gaps in the feature lines again.

To remove the small segments that are bordering other segments, each set is assigned a number. For each small set, each vertex in that set's feature line is traversed, and the set number of the neighboring vertices are recorded. The boundary line between the small set and the most frequently occurring neighbor set is deleted, while all other borders are maintained. This, in essence, opens the small sets up and converts them into small branches (Figure 21). Those small branches of feature lines are easily deleted by once again checking the color on both sides of the lines and deleting all feature lines that have the same color on both sides of the line. The final segmentation is shown in Figure 22.

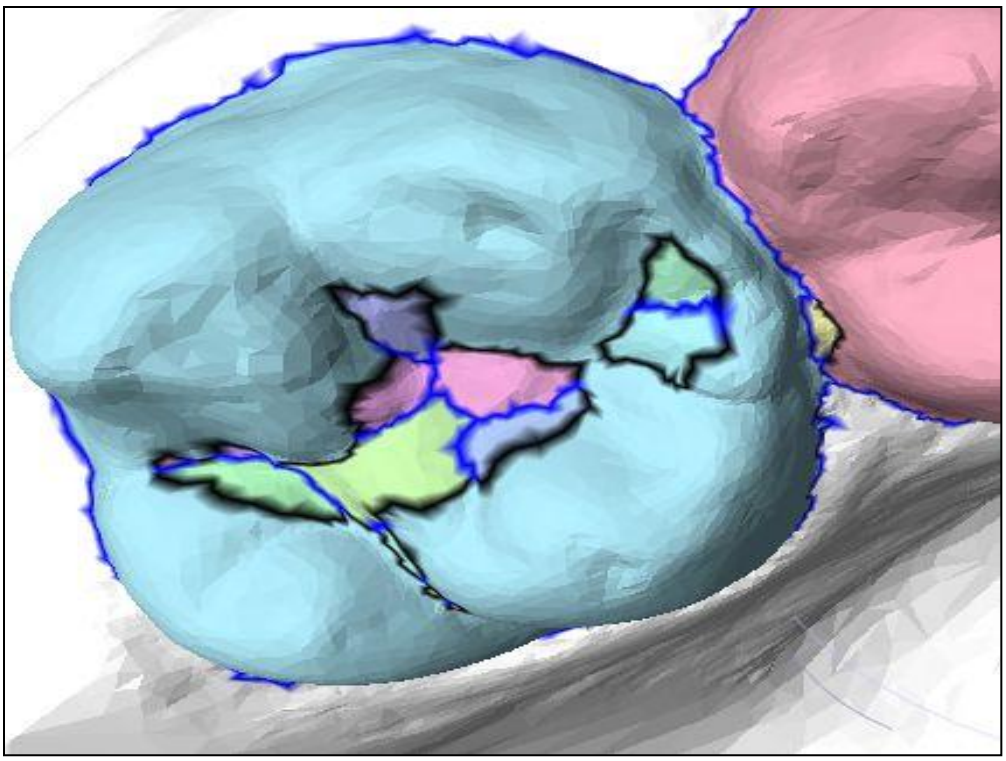

Figure 21. Black lines indicate borders of small segments that will be removed.

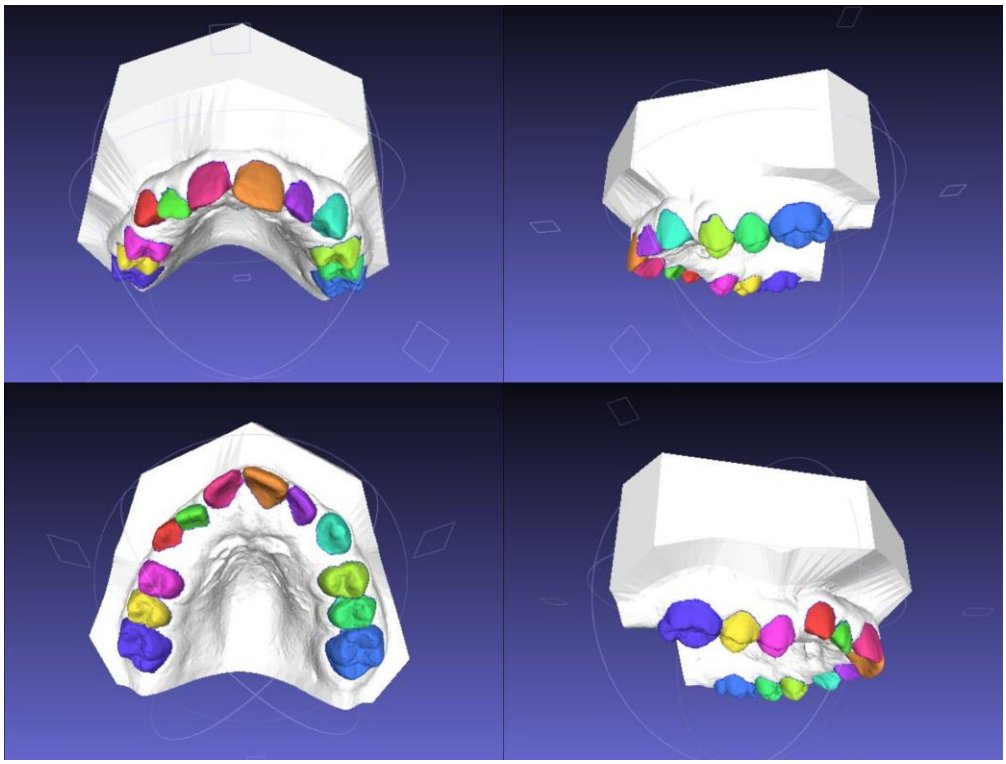

Figure 22. Final segmentation.

Search for the Best ε

The number of teeth found is compared to the number of teeth sought by the user. If ε is too low, not enough teeth will be found. If ε is too high, then extra artifacts will be segmented that are not teeth. A binary search is performed with the initial upper boundary of ε as 0 and lower boundary as -.30. Once the correct number of teeth is identified, ε is lowered by .01 and checked again until the lowest value of ε is discovered that segments the model into the same number of segments as the number of teeth provided by the user.

Testing the Algorithm

Six maxillary and six mandibular OrthoCad models were converted from .3DM to .STL format. The .STL files were analyzed using Materialise Magics 13.0 and inverted normals, overlapping triangles, planar holes, bad edges, bad contours, and intersecting triangles were repaired.[29] The twelve models were used as test cases for the segmentation algorithm.

The accuracy of the segmentation was measured by visual inspection; if more than ¼ of a tooth was not segmented correctly, for example, if a cusp of a posterior tooth was missed, then the segmentation was identified as erroneous. If, after the first segmentation results, one tooth was over-segmented and another tooth was not found, the user-input was increased. The number of correctly segmented teeth and number of errors were recorded for each model. The runtimes of the algorithm for each test model were also recorded.

RESULTS

The 12 models (6 mandibular and 6 maxillary) contained 160 teeth total to segment. The segmentation algorithm correctly segmented 133 of the 160 teeth for an 83.13% success rate overall. The algorithm correctly segmented 70 of 80 maxillary teeth for an 87.5% maxillary success rate. It correctly segmented 63 of 80 mandibular teeth for a 78.8% mandibular success rate. The runtime varied depending on the number of triangles in the model. The average runtime was 32 seconds on a 3.0GHz Intel® CoreTM2 Duo CPU with 4 GB RAM. Figures 23 through 34 show the segmentation of each of the test models. Figures 32 and 33 display the runtime and number of triangles in each model.

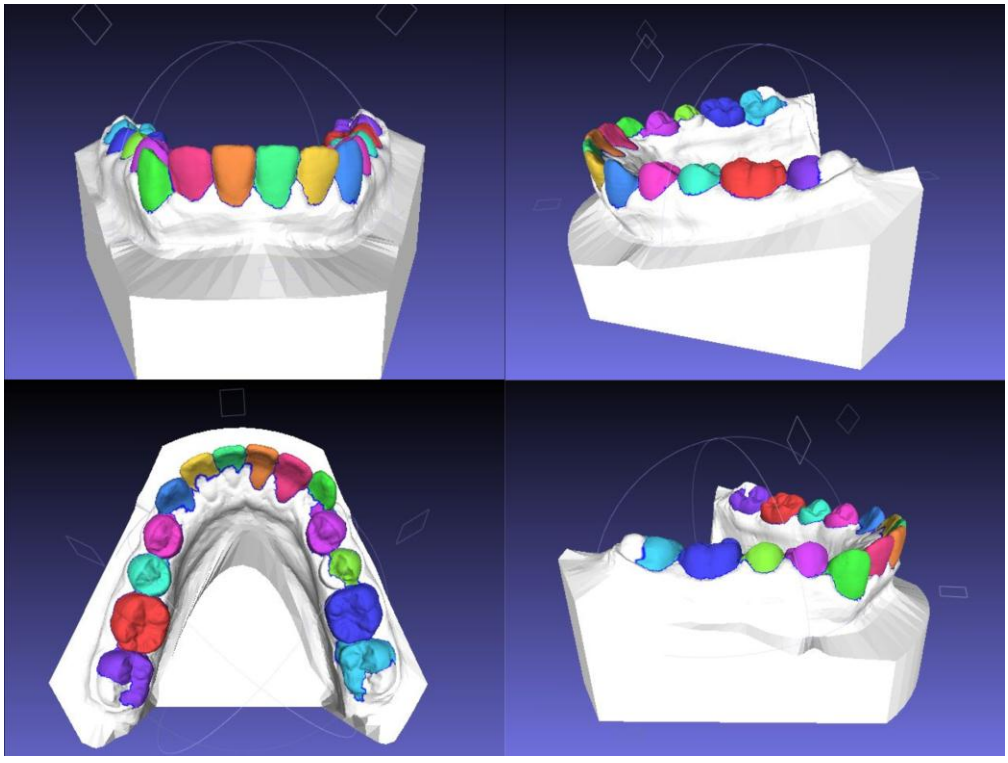

Figure 23. Results of Segmentation for Model 1.

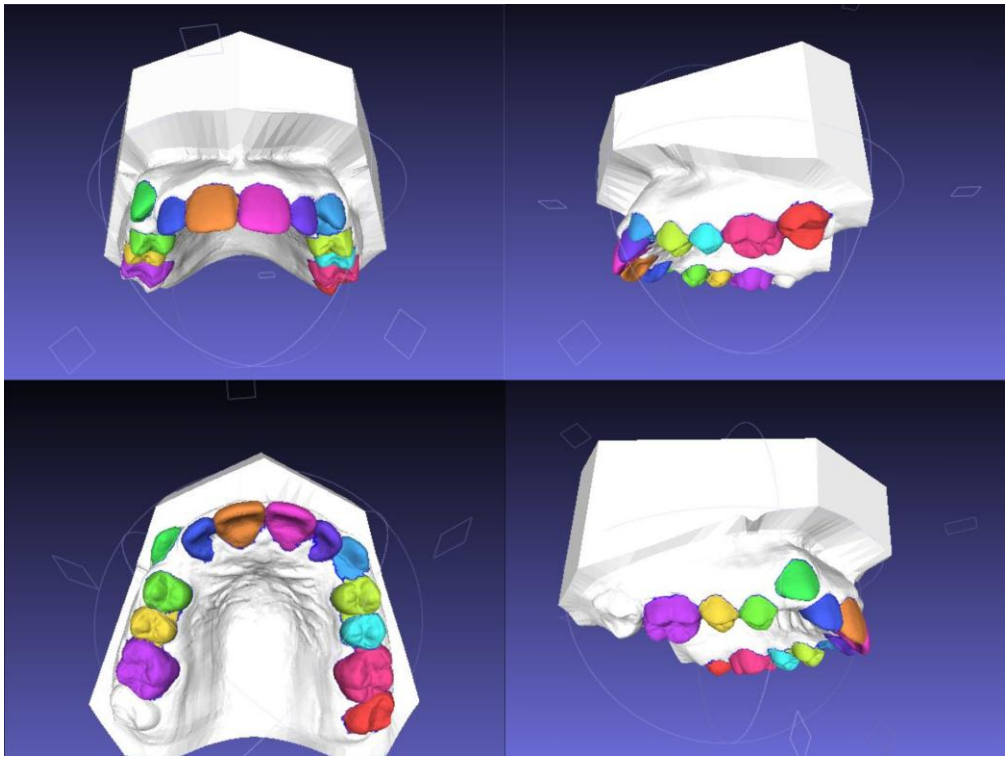

Figure 24. Results of Segmentation for Model 2.

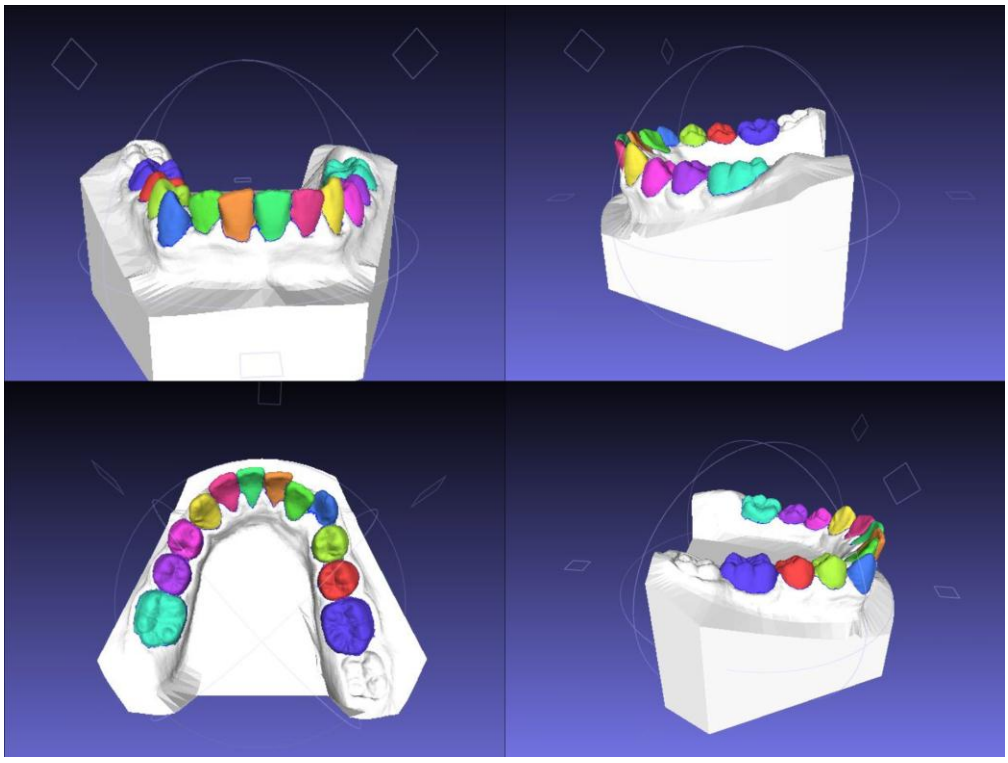

Figure 25. Results of Segmentation for Model 3.

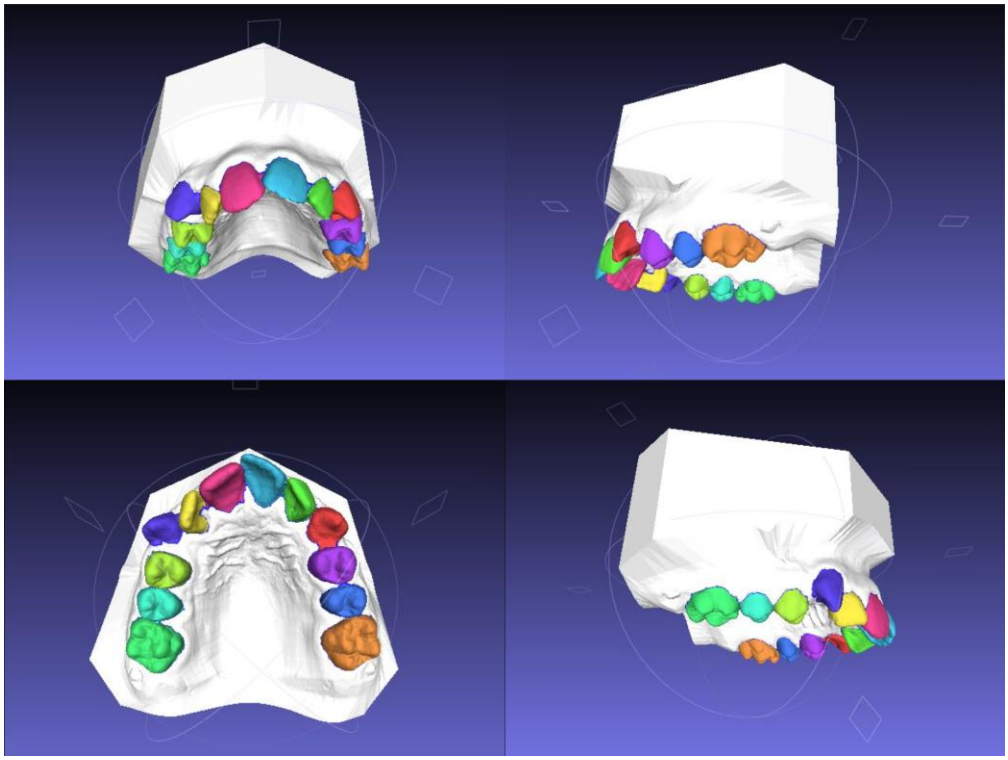

Figure 26. Results of Segmentation for Model 4.

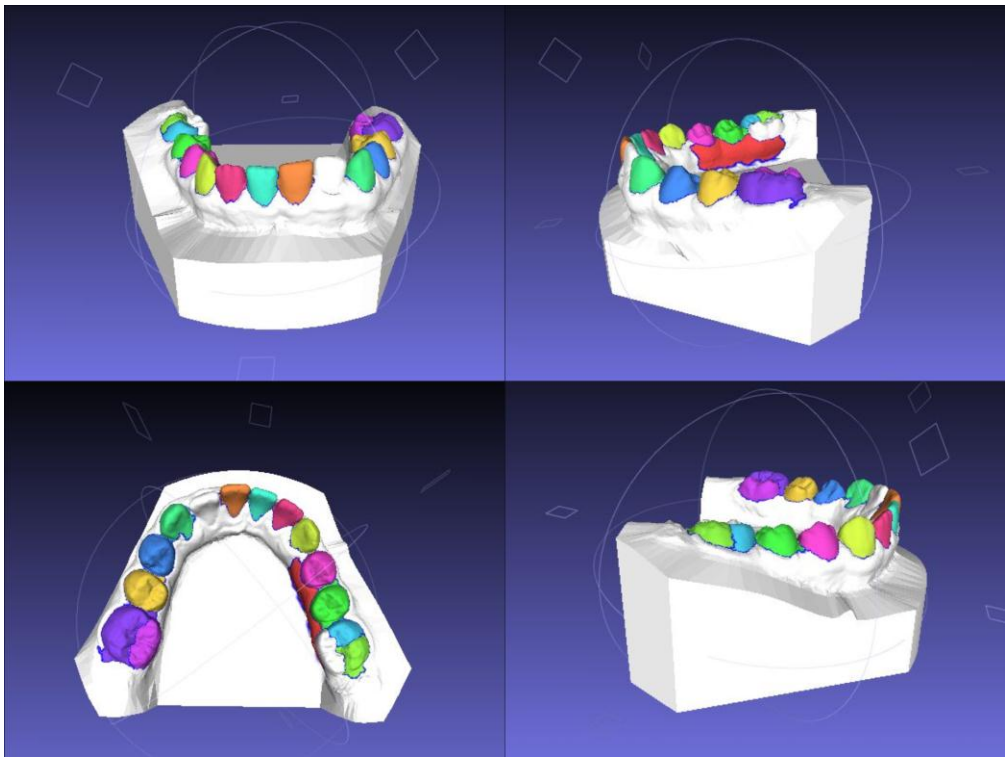

Figure 27. Results of Segmentation for Model 5.

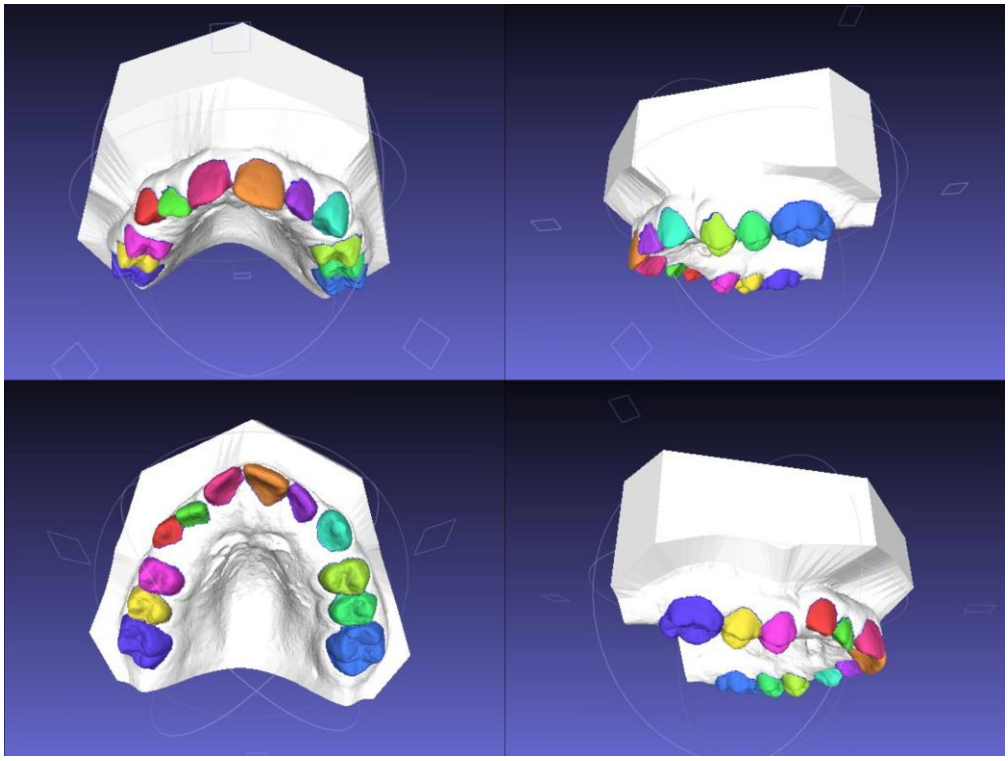

Figure 28. Results of Segmentation for Model 6.

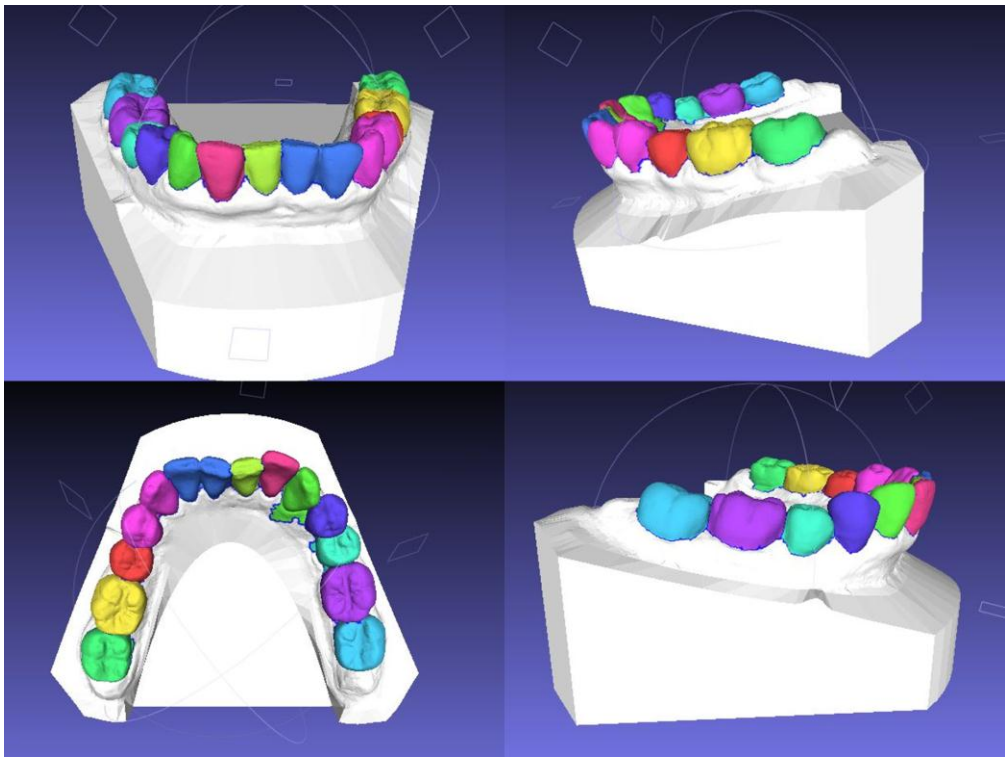

Figure 29. Results of Segmentation for Model 7.

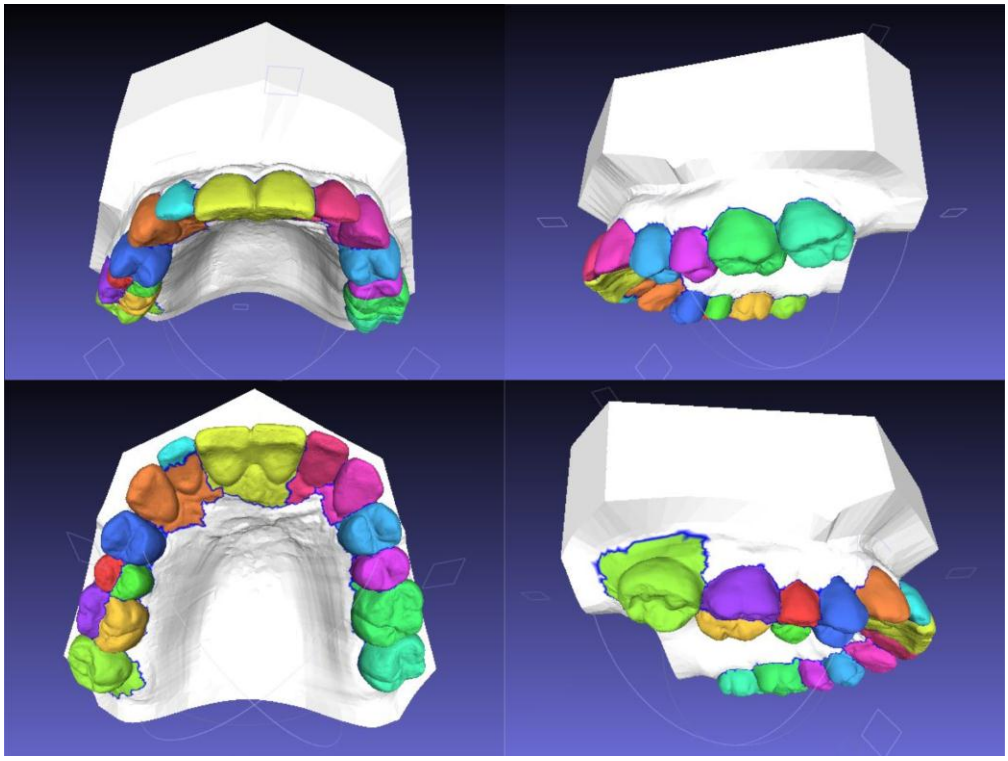

Figure 30. Results of Segmentation for Model 8.

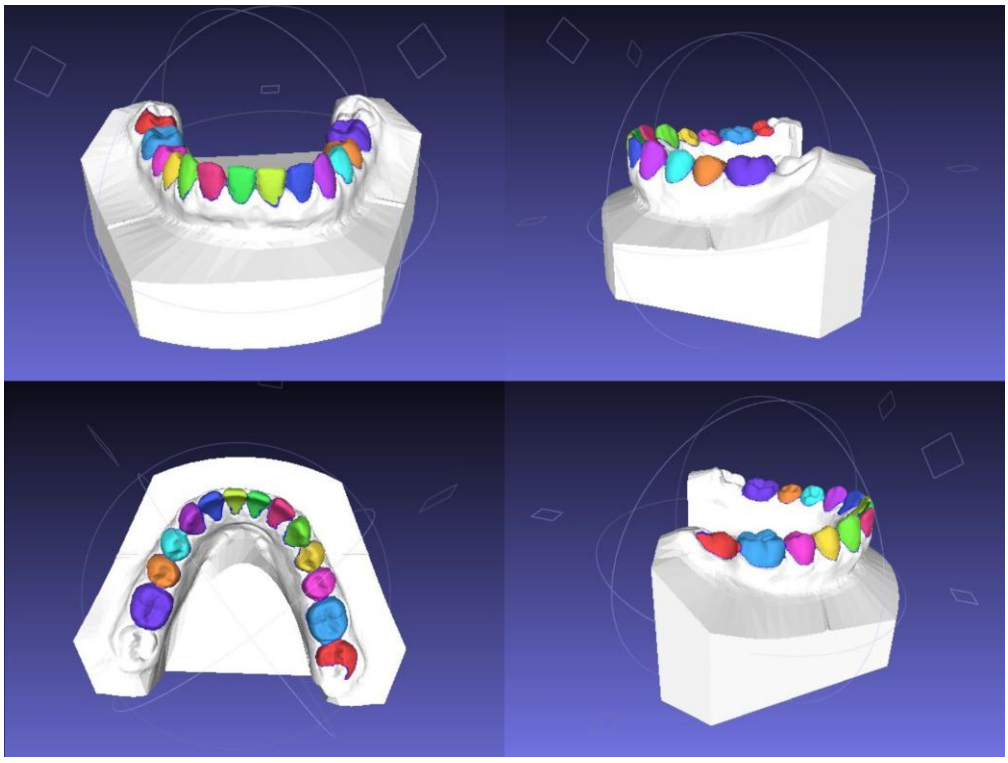

Figure 31. Results of Segmentation for Model 9.

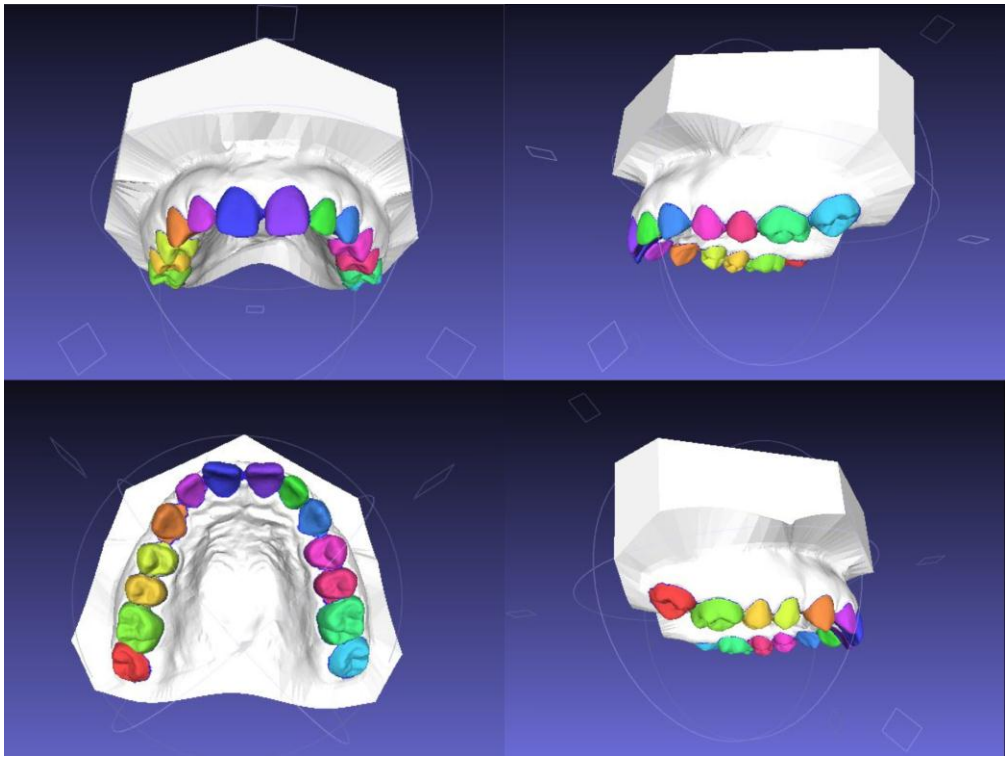

Figure 32. Results of Segmentation for Model 10.

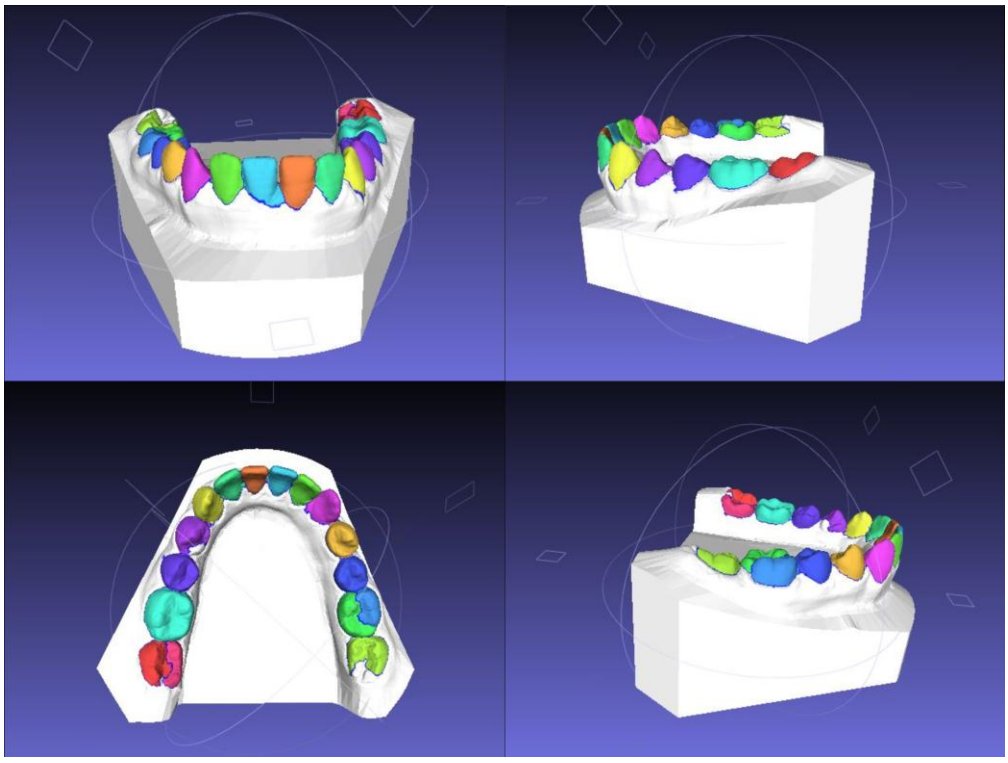

Figure 33. Results of Segmentation for Model 11.

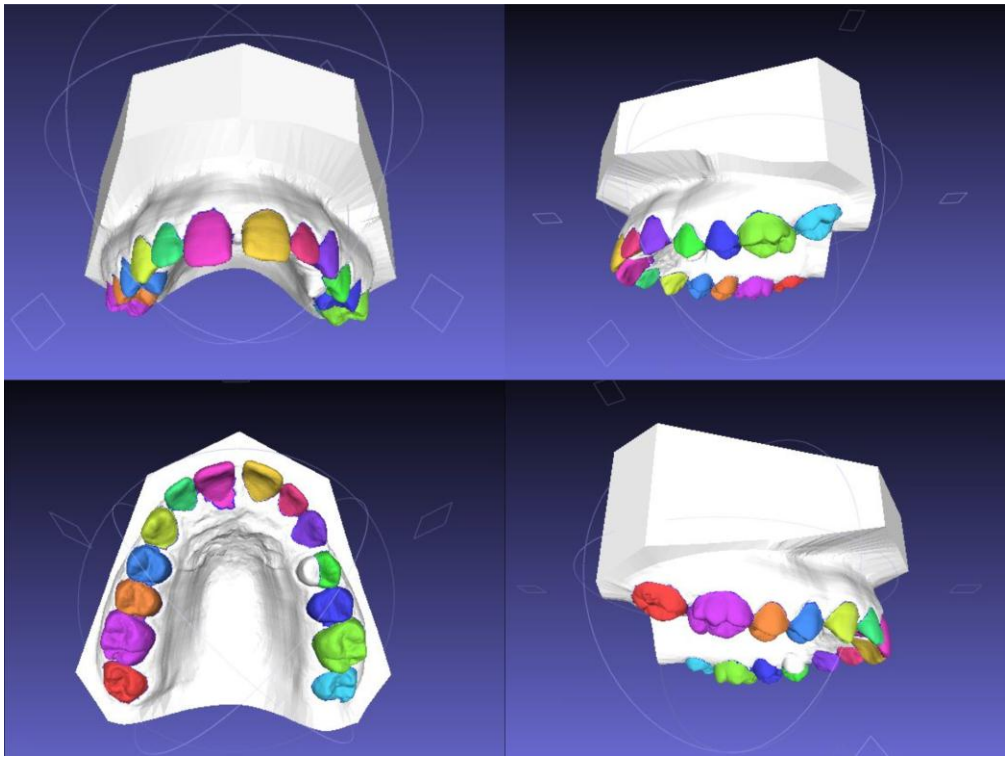

Figure 34. Results of Segmentation for Model 12.

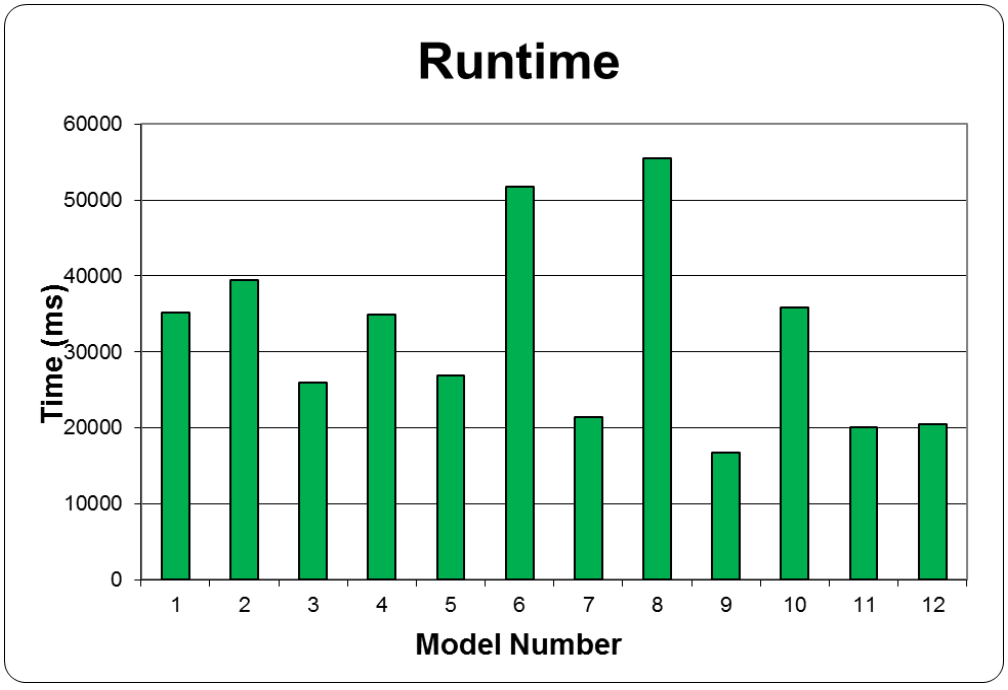

Figure 35. Runtime of algorithm for each model.

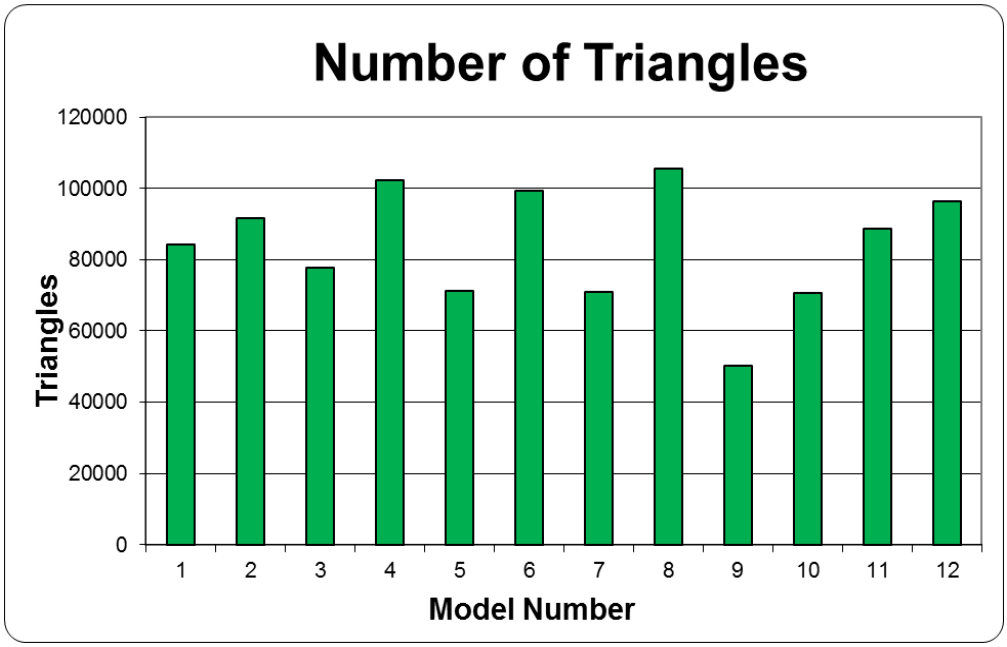

Figure 36. Number of triangles in each model.

DISCUSSION

Although the algorithm results in successful segmentation of teeth in many cases, there are some cases in which the results are less than desired. A disappointing final segmentation may result from: dental impressions of poor quality, digital models of poor quality, anatomic variations, the handling of artifacts, a poor choice of ε, and erupting teeth.

The quality of the digital model is extremely important for the algorithm to be successful. If the dental impression is of poor quality, and the gingival margin is not evident around each tooth, then there is little hope that the algorithm will segment the digital model correctly (Figure 34). Also, if the digital model is of low quality then the sharp distinction between tooth and gums will be softened and smoothed, preventing a distinct curvature change that the algorithm uses for segmentation.

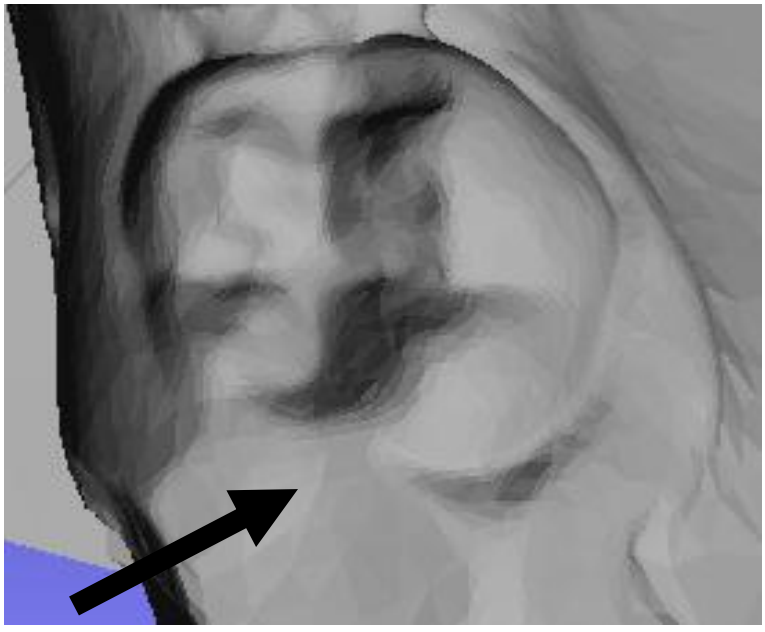

Figure 37. Poor distinction between tooth and gingiva on the distal aspect of a molar.

Over-segmentation, or splitting a single tooth into multiple segments, can occur if the tooth has pronounced occlusal anatomy with mesial and distal grooves. Under-segmentation, or missing a significant portion of a tooth, similarly results from deep grooves on the mesial, distal, buccal or lingual that connect to the occlusal surface. The problem is compounded if a tooth also has an indistinct transitions between tooth and gingiva at the gingival margin. Over-segmentation and under-segmentation occurs more frequently in maxillary first premolars and mandibular molars.

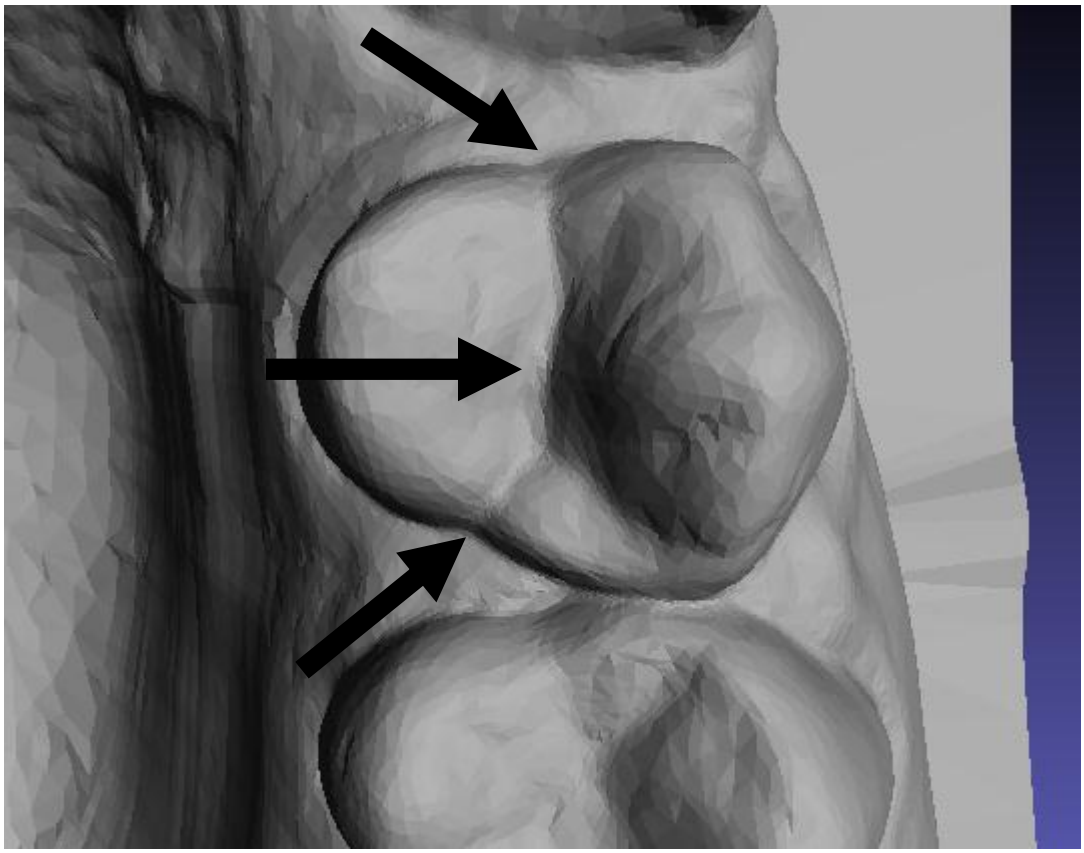

Figure 38. Mesial groove that extends onto the occlusal surface and then to a distal groove on a maxillary first premolar.

When relying only on the curvature to segment teeth, it is very difficult for the algorithm to distinguish between a deep groove that bisects a tooth and the negative curvature that separates the tooth from gingiva (Figure 36).

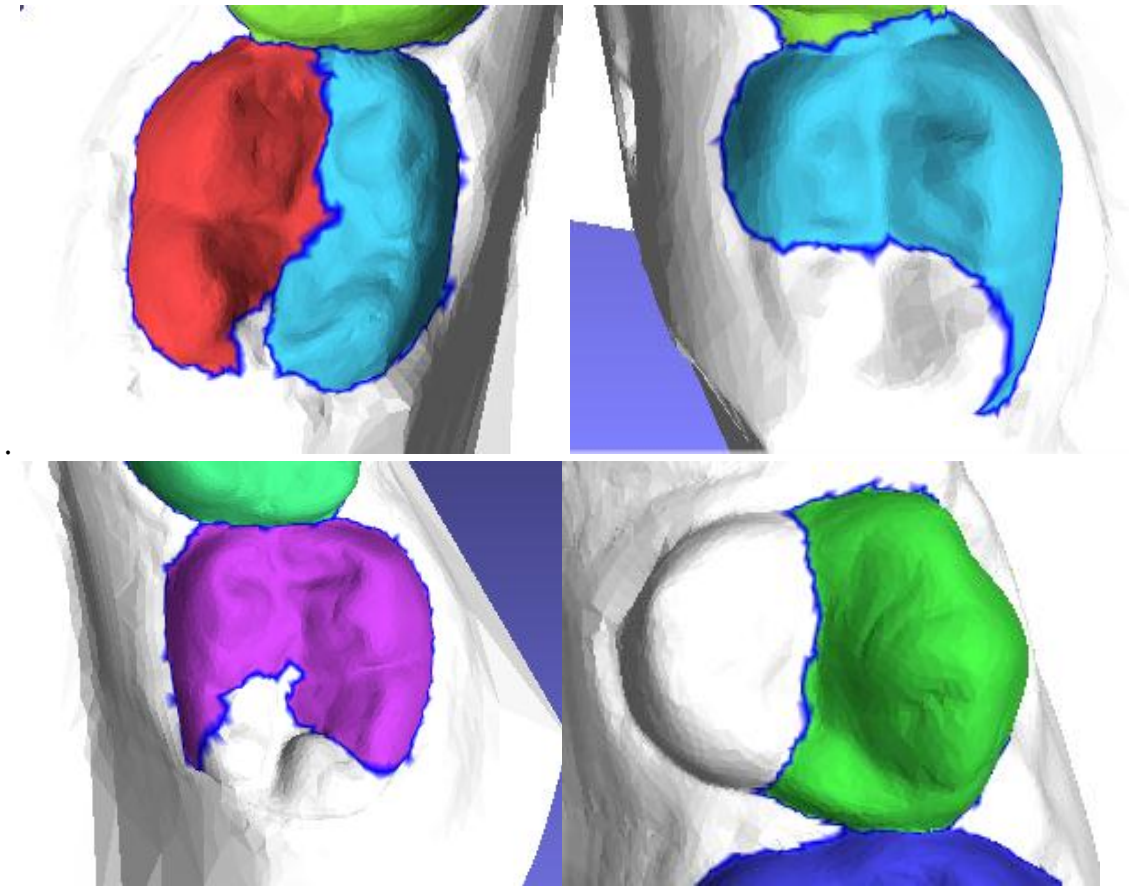

Figure 39. Pronounced occlusal anatomy coupled with a poor transition between tooth and gingiva results in under-segmented and over-segmented teeth.

In some cases the way artifacts are handled results in a final segmentation that is not representative of the actual boundary between tooth and gums. This is a problem when a small artifact occurs on the border of the tooth and gingiva. The algorithm must not open up the border, but must remove the small artifact segment. The algorithm chooses a part of the small segment to delete and opens up the small segment, but some cases will result in unnecessary inclusion of gingiva in a segmented tooth and other cases

will result in missing a small patch of tooth in the final segmentation.

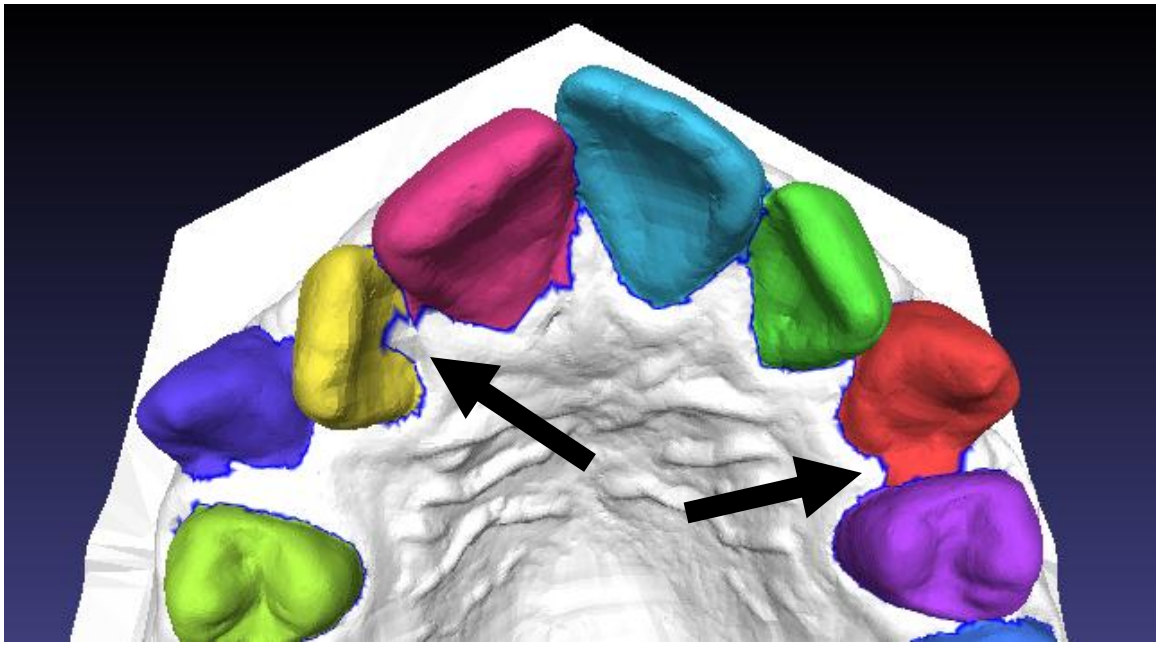

Figure 40. Boundaries between tooth structure and gingiva that are not accurate.

In some cases the automatic, final choice of ε is not ideal. The user specifies how many teeth are to be found, and then the algorithm iterates over different values of ε until the number of segments matches the number requested by the user. However, if some teeth are over- or under-segmented, an inappropriate value of ε may be chosen. This can result in poor segmentation results, like those seen in Figure 38.

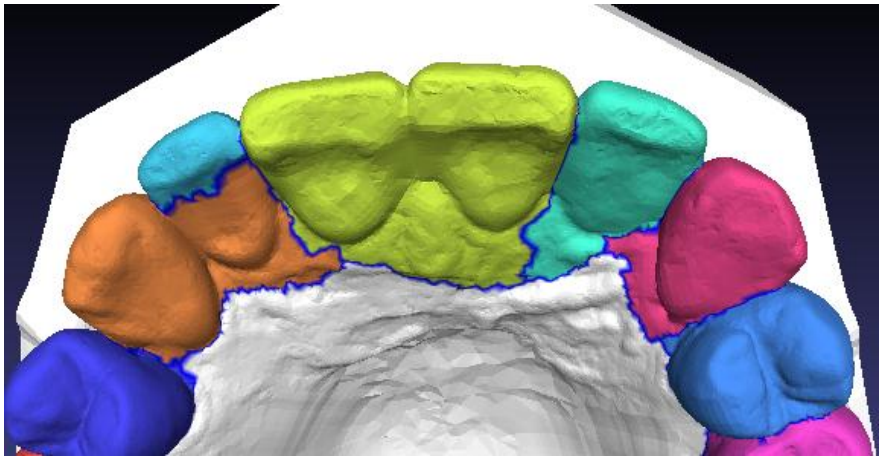

Figure 41. A striking failure of the algorithm to segment maxillary anterior teeth.

When a tooth is just erupting into the mouth, it may be difficult for the algorithm to identify the tooth correctly. This occurs because if the tooth is less that ¼ the size of the largest segmented tooth, then it will be seen as a small segment artifact that needs to be deleted (Figure 42).

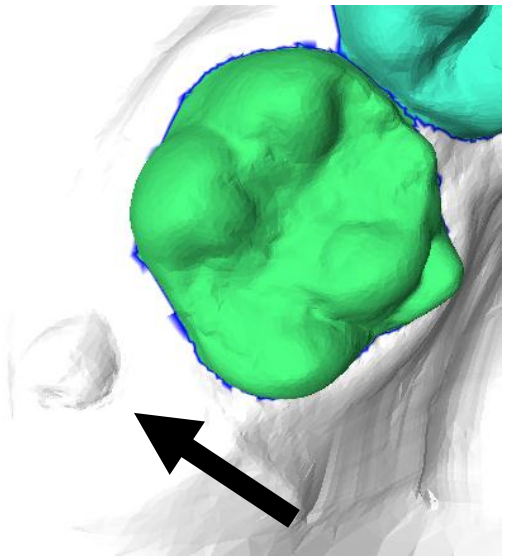

Figure 42. A cusp tip is erupting, but is not segmented as a tooth.

CONCLUSIONS

The novel tooth segmenting algorithm presented in this paper emphasizes usability—ideally tooth segmenting would require no human interaction with the model. With overall segmentation success at 83.13% and average runtime of 32 seconds, future research will focus on improving accuracy and speed, while maintaining usability. It is clear that more research needs to be done before efficient, fully automated segmentation of digital dental casts with no user interaction becomes a reality. By releasing the source code as open-source software it is hoped that future researchers will have a starting place if they choose to modify the code, or a baseline against which to test future algorithms.

REFERENCES

1. Hoffman DD, Singh M. Salience of visual parts. Cognition 1997;63(1):29-78.

2. Keim RG, Gottlieb EL, Nelson AH, III DSV. 2008 JCO Study of Orthodontic Diagnosis and Treatment Procedures, Part 1 Results and Trends. Journal of Clinical Orthodontics 2008;XLII(11):625-40.

3. Digital Record Requirements. 2012. "http://www.americanboardortho.com/professionals/clinicalexam/casereportpresentation/electronicguidelines.aspx". Accessed July 27 2012.

4. Geodigm Corporation: The Future of Dentistry. "http://www.geodigmcorp.com/". Accessed July 27 2012.

5. Lingual Appliances. "http://solutions.3m.com/wps/portal/3M/en_US/orthodontics/Unitek/products/lingual/". Accessed July 27 2012.

6. Invisalign. "www.invisalign.com/". Accessed July 27 2012.

7. Ortho Insight 3D. "http://motionview3d.com/". Accessed July 27 2012.

8. Orthocad. "http://www.cadent.biz/". Accessed July 27 2012.

9. Orthoplex. "https://www.gacorthoplex.com/GACOnline/pub/index.aspx". Accessed July 27 2012.

10. Suresmile: Reinventing the Science of Braces. "http://www.suresmile.com/". Accessed July 27 2012.

11. Resnick BN. A simplified diagnostic setup technique. J Clin Orthod 1979;13(2):128-9.

12. Ince DC, Hatton L, Graham-Cumming J. The case for open computer programs. Nature. 2012;482(7386):485-8. doi: 10.1038/nature10836.

13. Agathos A, Pratikakis I, Perantonis S, Sapidis N, Azariadis P. 3D Mesh Segmentation Methodologies for CAD applications. Computer-Aided Design & Applications 2007;4(6):827-41.

14. Hoffman DD, Richards WA. Parts of recognition. Cognition 1984;18(1-3):65-96.

15. Mokhtari M, Laurendeau D. Feature Detection on 3-D Images of Dental Imprints. Proceedings of IEEE Workshop Biomedical Image Analysis 1994:287-96.

16. Kondo T, Ong SH, Foong KWC. Tooth Segmentation of Dental Study Models Using Range Images. IEEE Transactions on Medical Imaging 2004;23(3):350-62.

17. Zhao M, Ma L, Tan W, Nie D. Interactive Tooth Segmentation of Dental Models. Confernce Proceedings of IEEE Engineering in Medicine and Biology Society 2005;1:654-57.

18. Tian-ran Y, Ning D, Guo-dong H, et al. Bio-information based segmentation of 3D dental models. The 2nd International Conference on Bioinformatics and Biomedical Engineering 2008:624-27.

19. Yau H-T, Yang T-J, Hsu C-Y, Tseng H-S. Development of a Virtual Orthodontic Alignment System using a Haptic Device. Computer-Aided Design & Applications 2010;7(6):889-98.

20. Wongwaen N, Sinthanayothin C. Computerized Algorithm for 3D Teeth Segmentation. International Conference on Electronics and Information Engineering 2010 2010;1:V1-277 - V1-80.

21. Kronfeld T, Brunner D, Brunnett G. Snake-Based Segmentation of Teeth from Virtual Dental Casts. Computer-Aided Design & Applications 2010;7(2):221-33.

22. Y. Kumar RJ, B. Larson and J. Moon. Improved Segmentation of Teeth in Dental Models. Computer-Aided Design & Applications 2011;8(2):211-24.

23. Brunner D, Brunnett G. Closing Feature Regions. Chemnitzer Informatik-Berichte 2011.

24. Meshlab. 2011. "http://meshlab.sourceforge.net/". Accessed May 1 2011.

25. QTE Creator IDE and Tools. 2011. "http://qt.nokia.com/products/developer-tools/". Accessed May 1 2011.

26. Meyer M, Desbrun M, Schroder P, Barr AH. Discrete Differential-Geometry Operators for Triangulated 2-Manifolds. Proceedings of the International Workshop on Visualization and Mathematics 2002.

27. Vergne R, Barla P, Granier X, Schlick C. Apparent relief: a shape descriptor for stylized shading. Proceedings of the 6th international symposium on Non-photorealistic animation and rendering 2008:23-29.

28. Rossl C, Kobbelt L, Seidel H-P. Extraction of feature lines on triangulated surfaces using morphological operators. Proceedings of the AAAI Symposium on Smart Graphics 2000:71-75.

29. Materialise: Software for additive manufacturing. Plymouth, MI: 2012. "http://software.materialise.com/magics-0". Accessed May 1 2012.

APPENDIX A

SEGMENTATION ALGORITHM SOURCE CODE

```
#include "meshcolorizeteeth.h"
#include <math.h>
#include <vcg/complex/algorithms/nringteeth.h>
#include <vcg/math/disjoint_set.h>
#include <vcg/complex/algorithms/geodesic_teeth.h>
#include <wrap/gl/addons.h>
#include <vcg/complex/algorithms/stat.h>
#include <vcg/math/base.h>
#include <vcg/container/simple_temporary_data.h>
#include <deque>
#include <vector>
#include <list>
#include <functional>
#include <time.h>
#define PI 3.14159265
ExtraMeshcolorizeteethPlugin::ExtraMeshcolorizeteethPlugin() {
    typeList <<
                CP_DISCRETE_CURVATURETEETH;
    FilterIDType tt;
    foreach(tt , types())
        actionList << new QAction(filterName(tt), this);
}
QString ExtraMeshcolorizeteethPlugin::filterName(FilterIDType c) const{
    switch(c){
    case CP_DISCRETE_CURVATURETEETH:        return QString("Discrete
Teeth Curvatures");
    default: assert(0);
    }
    return QString("error!");
}
QString ExtraMeshcolorizeteethPlugin::filterInfo(FilterIDType filterId)
const {
    switch(filterId){
    case CP_DISCRETE_CURVATURETEETH :       return
QString("colorize:");
    default: assert(0); return QString("");
    }
}
// What "new" properties the plugin requires
int ExtraMeshcolorizeteethPlugin::getRequirements(QAction *action){
    switch(ID(action)){
    case CP_DISCRETE_CURVATURETEETH:       return
MeshModel::MM_FACEFACETOPO | MeshModel::MM_FACEFLAGBORDER |
MeshModel::MM_VERTCURV | MeshModel::MM_VERTFACETOPO;
    default: assert(0); return 0;
    }
    return 0;
```

```
}
void ExtraMeshcolorizeteethPlugin::initParameterSet(QAction *a,
MeshModel &m, RichParameterSet & par){
    // variables cannot be defined within switch statement
    QStringList metrics;
    QStringList curvNameList;
    pair<float,float> minmax;
    switch(ID(a)){
    case CP_DISCRETE_CURVATURETEETH:
        curvNameList.push_back("Mean Curvature");
        curvNameList.push_back("Gaussian Curvature");
        curvNameList.push_back("RMS Curvature");
        curvNameList.push_back("ABS Curvature");
        par.addParam(new RichEnum("CurvatureType", 0, curvNameList,
tr("Type:"),
                                  QString("Choose the curvatures.")));
        //Curvature cut-off
        par.addParam(new RichInt("numTeeth",
                                 14,
                                 "How many teeth on model?",
                                 "How many teeth are on the model"));
        break;
    }
}
void ExtraMeshcolorizeteethPlugin::printQuality( MeshDocument & md)
{
    MeshModel &m=*(md.mm());
    CMeshO::VertexIterator vi;
    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi)
        if(!(*vi).IsD())
        {
            qDebug() << "Print Quality: " <<  (float)(*vi).Q();
        }
}
/* Auxiliary stcuct for heap of vertices to visit while searching
connecting points
*/
struct VertDist{
    VertDist(){}
    VertDist(CMeshO::VertexPointer _v, float _d):v(_v),d(_d){}
    CMeshO::VertexPointer v;
    CMeshO::ScalarType d;
};
/* Temporary data to for connecting points search: estimated distance
and source
*/
struct TempData{
    TempData(){}
    TempData(const CMeshO::ScalarType & d_){d=d_;source = NULL;}
    CMeshO::ScalarType d;
    CMeshO::VertexPointer source;//closest source
};
typedef SimpleTempData<std::vector<CMeshO::VertexType>, TempData >
TempDataType;
struct pred: public std::binary_function<VertDist,VertDist,bool>{
    pred(){};
    bool operator()(const VertDist& v0, const VertDist& v1) const
```

```
    {return (v0.d > v1.d);}
};
struct pred_addr: public std::binary_function<VertDist,VertDist,bool>{
    pred_addr(){};
    bool operator()(const VertDist& v0, const VertDist& v1) const
    {return (v0.v > v1.v);}
};
bool ExtraMeshcolorizeteethPlugin::applyFilter(QAction *filter,
MeshDocument &md, RichParameterSet & par, vcg::CallBackPos *cb){
    MeshModel &m=*(md.mm());
    // add a per-vertex attribute with type int named "marked"
    CMeshO::PerVertexAttributeHandle<int> marked =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<int>
(m.cm,std::string("marked"));
    // add a per-vertex attribute with type float named "curvature"
    CMeshO::PerVertexAttributeHandle<float> sumPosSetcurvature =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<float>
(m.cm,std::string("sumPosSetcurvature"));
    // add a per-vertex attribute with type float named "curvature"
    CMeshO::PerVertexAttributeHandle<float> curvature =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<float>
(m.cm,std::string("curvature"));
    // add a per-vertex attribute with type float named
"DistFromFeature"
    CMeshO::PerVertexAttributeHandle<float> disth =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<float>
(m.cm,std::string("DistFromFeat"));
    // add a per-vertex attribute with type char named "setFBV"
    CMeshO::PerVertexAttributeHandle<char> setFBV =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<char>
(m.cm,std::string("setFeatureBorderVertex"));
    // add a per-vertex attribute with type float named
"DivergenceFieldValue"
    CMeshO::PerVertexAttributeHandle<float> divfval =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<float>
(m.cm,std::string("DivergenceFieldValue"));
    // add a per-vertex attribute with type int named
"FeatureSetNumber"
    CMeshO::PerVertexAttributeHandle<int> FSetNum =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<int>
(m.cm,std::string("FeatureSetNumber"));
    // add a per-vertex attribute with type int named "CpointOwner"
    CMeshO::PerVertexAttributeHandle<int> CpointOwner =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<int>
(m.cm,std::string("CpointOwner"));
    // add a per-vertex attribute with type VertexPointer named
"source"
    CMeshO::PerVertexAttributeHandle<CMeshO::VertexPointer> source =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<CMeshO::VertexPointe
r>  (m.cm,std::string("source"));
    // add a per-vertex attribute with type VertexPointer named
"source1"
    CMeshO::PerVertexAttributeHandle<CMeshO::VertexPointer> source1 =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<CMeshO::VertexPointe
r>  (m.cm,std::string("source1"));
    // add a per-vertex attribute with type float named "disc"
```

```
    CMeshO::PerVertexAttributeHandle<int> disc =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<int>
(m.cm,std::string("disc"));
    // add a per-vertex attribute with type float named "center"
    CMeshO::PerVertexAttributeHandle<int> center =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<int>
(m.cm,std::string("center"));
    // add a per-vertex attribute with type float named "complex"
    CMeshO::PerVertexAttributeHandle<int> complex =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<int>
(m.cm,std::string("complex"));
    // add a per-vertex attribute with type int named "newF"
    CMeshO::PerVertexAttributeHandle<int> newF =
vcg::tri::Allocator<CMeshO>::AddPerVertexAttribute<int>
(m.cm,std::string("newF"));
    switch(ID(filter)) {
    case CP_DISCRETE_CURVATURETEETH:
    {
        //Curvature cut-off
        int numTeeth = par.getInt("numTeeth");
        bool filterOn = true;
        //Morphological Operations before
        bool morphologicalOperations = true;
        bool skeletonize = true;
        bool prune = true;
        float pruneLength = 3.0f;
        //Show first time
        bool showSegments1st = true;
        bool removeLines1st = true;
        float deletePercentOfF = 0.01f;
        bool onlyShowFeature = true;
        float distanceConstraint = 3.0f;
        //Connect
        bool findConnectingPoints = true;
        float connectingPointAngle = 240.0f;
        bool findMidPoints = true;
        bool findPathstoConnect = true;
        bool shortestEuclidianDistToConnect = true;
        bool addPathstoFeatureRegion = true;
        //Re-skeletonize and Prune
        bool skeletonizeAfter = true;
        bool pruneAfter = true;
        float pruneAfterLength = 10.0f;
        //Finish showing segments
        bool showSegments = true;
        bool removeLines = true;
        float deletePercentOfBiggestSegment = 0.25;
        bool combineNeighboringSmallSets = true;
        bool deletePercentOfLargestSegment = true;
        bool showSegmentsAgain = true;
        if ( tri::Clean<CMeshO>::CountNonManifoldEdgeFF(m.cm) > 0) {
            errorMessage = "Mesh has some not 2-manifold faces,
Curvature computation requires manifoldness"; // text
            return false; // can't continue, mesh can't be processed
        }
        int delvert=tri::Clean<CMeshO>::RemoveUnreferencedVertex(m.cm);
```

```
        if(delvert) Log("Pre-Curvature Cleaning: Removed %d
unreferenced vertices",delvert);
        tri::Allocator<CMeshO>::CompactVertexVector(m.cm);
        //binary search variables
        int first = -30;
        int last = 0;
        bool foundIt = false;
        //for each value of epsilon, test it out
        float epsilon;
        int totalNumBigSets = 0;
        int totalNumLittleSets = 0;
        while (first <= last && foundIt == false) {
            int intEpsilon = (first + last) / 2;  // compute integer
representation of epsilon point.
            epsilon = intEpsilon / 100.0f;
            /* ********************  Calculate curvature: curvature.h
computes the discrete gaussian curvature.
            <vcg/complex/algorithms/update/curvature.h>
            For further details, please, refer to:
            Discrete Differential-Geometry Operators for Triangulated
2-Manifolds Mark Meyer,
            Mathieu Desbrun, Peter Schroder, Alan H. Barr VisMath '02,
Berlin </em>*/
            // results stored in (*vi).Kh() (mean) and (*vi).Kg()
(guassian)
            tri::UpdateCurvature<CMeshO>::MeanAndGaussian(m.cm);
            // ********************  Put the curvature in the quality:
<vcg/complex/algorithms/update/quality.h>
            tri::UpdateQuality<CMeshO>::VertexFromMeanCurvature(m.cm);
            //****************** ComputePerVertexQualityMinMax 15 and
85 percent
            std::pair<float,float> minmax =
std::make_pair(std::numeric_limits<float>::max(),-
std::numeric_limits<float>::max());
            CMeshO::VertexIterator vi;
            std::vector<float> QV;
            QV.reserve(m.cm.vn);
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi)
                if(!(*vi).IsD()) QV.push_back((*vi).Q());
            //bottom 15% maps to -1

std::nth_element(QV.begin(),QV.begin()+15*(m.cm.vn/100),QV.end());
            float newmin=*(QV.begin()+m.cm.vn/100);
            //top 15% maps to 1
            std::nth_element(QV.begin(),QV.begin()+m.cm.vn-
15*(m.cm.vn/100),QV.end());
            float newmax=*(QV.begin()+m.cm.vn-m.cm.vn/100);
            minmax.first = newmin;
            minmax.second = newmax;
            // ************************** MapCurvatureRangetoOneNegOne
            float B = 8.0;
            float min = minmax.first;
            float max = minmax.second;
            float upperBound = 0.0;
            if(min < 0)
                min = min * -1;
            if(min > max)
```

```
                upperBound = min;
            else
                upperBound = max;
            float K = 0.5 * log10((1 + ((pow(2.0,B)-2)/(pow(2.0,B)-1)))
/ ((1 - ((pow(2.0,B)-2)/(pow(2.0,B)-1)))));
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi)
                if(!(*vi).IsD())
                {
                    float remapped = tanh((*vi).Q() * (K /
upperBound));
                    (*vi).Q() = remapped;
                }
            vcg::DisjointSet<CVertexO>* ptrDset = new
vcg::DisjointSet<CVertexO>();
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //all vertices start as set V for Vertex and unmarked
                setFBV[vi] = 'V';
                marked[vi] = 0;
                disc[vi] = 0;
                center[vi] = 0;
                complex[vi] = 0;
                newF[vi] = 0;
                //all vertices start as -1 FSetNum
                FSetNum[vi] = -1;
                //none have been visited to connect C points, so set
CpointOwner to -1
                CpointOwner[vi] = -1;
                //color it White initially
                (*vi).C() = Color4b(Color4b::White);
                curvature[vi] = (*vi).Q();
                //feature region, less than defined min curvature
                if(curvature[vi] < epsilon) {
                    //in set F for feature
                    setFBV[vi] = 'F';
                }
            }
            // apply morphological opperators dilate and then erode
****************************
            if(morphologicalOperations){
                //dilate 1-ring neighbor
                for(int k = 0; k < 1; k++){
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] != 'V') {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature, mark the neighbor to be one
                                //if it is negative curvature
```

```
                                if(setFBV[&(*tempVertexPointer)] ==
'V') {
                                    //mark it to be 'N'
                                    marked[&(*tempVertexPointer)] = 1;
                                }
                            }
                        }
                    }
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it was marked, change it
                        if(marked[vi] == 1){
                            setFBV[vi] = 'F';
                            marked[vi] = 0;
                        }
                    }
                }//grow once
                //erode 1-ring neighbor
                for(int k = 0; k < 1; k++){
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] != 'V') {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            bool neighborsAllF = true;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature, we will erode this vertex
                                if(setFBV[&(*tempVertexPointer)] ==
'V') {
                                    neighborsAllF = false;
                                }
                            }
                            if(neighborsAllF){
                                //mark to not erode this vertex this
round
                                marked[vi] = 1;
                            }
                        }
                    }
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it was marked to not erode, keep it
                        if(marked[vi] == 1){
                            //setFBV[vi] = 'F';
                            marked[vi] = 0;
                        }
                        else{
                            setFBV[vi] = 'V';
                        }
                    }
```

```
                }//erode once
            }
            //Apply Skeletonize operator
            if(skeletonize){
                int debugCounter = 1;
                bool changes = false;
                //if still changes loop through and skeletonize
                do {
                    changes = false;
                    //go through the vertices and if 'F' check for
centers and discs
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            //check for centers and discs
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            bool neighborsAllF = true;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature, vi not a center
                                if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                    neighborsAllF = false;
                                }
                            }
                            if(neighborsAllF){
                                //mark as a center
                                center[vi] = 1;
                                //mark all neighbors as disc
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    disc[&(*tempVertexPointer)] = 1;
                                }
                            }
                        }//in set F
                    }//go through vertices
                    //go through vertices again, if not a center and is
a disc, check complexity and delete if not complex
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(center[vi] == 0 && disc[vi] == 1){
                            //check complexity
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            int numberOfChanges = 0;
```

```
                            char iVertex;
                            char iPlus1Vertex;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if(i < (OneRing.allV.size() - 1)) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(i+1);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                else {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(0);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                //change noted
                                if(iVertex != iPlus1Vertex){
                                    numberOfChanges++;
                                }
                            }
                            //not complex, delete it from feature
                            if(numberOfChanges < 4){
                                //delete from feature
                                setFBV[vi] = 'V';
                                center[vi] = 0;
                                disc[vi] = 0;
                                changes = true;
                            }
                        }//not a center, yes a disc
                    }//go through vertices
                    //******************************** reset disc and
center data for next iteration
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            center[vi] = 0;
                            disc[vi] = 0;
                        }
                    }
                    /* if(debugCounter == 1){
                  changes = false;
               }*/
                } while (changes == true);//loop if changes
            }//skeletonize operator
            //prune
            if(prune){
                //prune a certain number of times
```

```
                for(int pruneIter = 0; pruneIter < pruneLength;
pruneIter++){
                    int pruned = 0;
                    //go through the vertices and if 'F' check
complexity
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F'){
                            //check complexity
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            int numberOfChanges = 0;
                            char iVertex;
                            char iPlus1Vertex;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if(i < (OneRing.allV.size() - 1)) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(i+1);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                else {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(0);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                //change noted
                                if(iVertex != iPlus1Vertex){
                                    numberOfChanges++;
                                }
                            }
                            //not complex, delete it from feature
                            if(numberOfChanges < 4){
                                //prune from feature
                                setFBV[vi] = 'V';
                                pruned++;
                            }
                        }//checking current vertex
                    }//go through vertices
                }//number of times we prune
            }//prune before
```

```
            //****************************** Initialize the Disjoint
sets of 'F' and delete any that are small
*****************************
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                if(setFBV[vi] == 'F'){
                    //put the vertex in the set D
                    ptrDset->MakeSet(&(*vi));
                }
            }
            /*
        if(showMinAndMaxFeature || onlyShowFeature){
            //go through the vertices and color them according to sets
            for(vi=m.cm.vert.begin();vi!=m.cm.vert.end();++vi) {
                if(setFBV[&(*vi)] == 'F')
                    (*vi).C() = Color4b(Color4b::Blue);
                if(setFBV[&(*vi)] == 'N')
                    (*vi).C() = Color4b(Color4b::Green);
                  //more than defined maximum curvature
               // if(!onlyShowFeature) {
                  //   if(curvature[vi] > eta)
                    //    (*vi).C() = Color4b(Color4b::Red);
               // }
            }
        }
    */
            //print number of sets
            //ptrDset->printNumberOfSets();
            //Go through vertices again--if in set 'F',
*****************************
            //then find 1-Ring of vertices and merge sets if orginal
vertex is 'F'
            //and new 1-ring vertex is 'F', define set 'B'
            int blueCounter = 0;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is a feature, then find 1-ring and merge sets
                if(setFBV[vi] == 'F') {
                    blueCounter++;
                    vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*vi));
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                        //if the neighbor of the current vertex is a
feature then merge sets
                        if(setFBV[&(*tempVertexPointer)] == 'F') {
                            //if they are not already in the same set,
merge them
                            if(ptrDset->FindSet(&(*vi)) != ptrDset-
>FindSet(tempVertexPointer)){
                                ptrDset-
>Union(&(*vi),tempVertexPointer);
                            }
                        }
                        else if(setFBV[&(*tempVertexPointer)] == 'V'){
                            //if the neighbor of the blue vertex is
just a vertex it is on the border
```

```
                            //we do this after we delete the small sets
of 'F'
                            //setFBV[&(*tempVertexPointer)] = 'B';
                        }
                    }
                }
            }
            //Find out how many distinct sets there are
*****************************
            int setCounter = 0;
            vector <CVertexO*> parentVertexofSet;
            typedef  std::vector < std::vector <CVertexO*> > matrix;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is in set 'F', then find what set it belongs to
                if(setFBV[vi] == 'F') {
                    if (ptrDset->FindSet(&(*vi)) == &(*vi)){
                        //parent of a new set
                        parentVertexofSet.push_back(&(*vi));
                        setCounter++;
                    }
                }
            }
            matrix allSetsWithVertices(setCounter,
std::vector<CVertexO*>(0));
            // ******************************  Create a vector of
vectors containing our sets / vertices
            vector<CVertexO*>::iterator itVect;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is in set 'F', then find what set it belongs to
                if(setFBV[vi] == 'F') {
                    //find offset in parent vector--that is the vertex
set number
                    itVect = find(parentVertexofSet.begin(),
parentVertexofSet.end(), ptrDset->FindSet(&(*vi)));
                    if( itVect != parentVertexofSet.end() ) {
                        int offset =
std::distance(parentVertexofSet.begin(), itVect);
                        //store the set number in the vertex
                        FSetNum[vi] = offset;
                        allSetsWithVertices[offset].push_back(&(*vi));
                    }
                }
            }
            //delete the disjoint set
            delete ptrDset;
            //delete sets of D if the number of vertices is less than
.01 D
            for(int pos=0; pos < allSetsWithVertices.size(); pos++)
            {
                if(allSetsWithVertices[pos].size() <=
deletePercentOfF*blueCounter) {
                    for(int i=0; i < allSetsWithVertices[pos].size();
i++) {
                        (*(allSetsWithVertices[pos][i])).C() =
Color4b(Color4b::White);
                        setFBV[(*(allSetsWithVertices[pos][i]))] = 'V';
                    }
```

```
                }
            }
            //************************ Color the different segments
1st Time and then delete F lines that have same color on both sides
            //** This clears up a lot of the extraneous connecting
points before we close gaps.
            if(showSegments1st){
                //***** New DisjointSet
                vcg::DisjointSet<CVertexO>* DisjointSetsToColor = new
vcg::DisjointSet<CVertexO>();
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //Every vertex not in set 'F' starts as disjoint
set
                    if(setFBV[vi] != 'F') {
                        //put the vertex in the set D
                        DisjointSetsToColor->MakeSet(&(*vi));
                    }
                }
                //***** Merge neighboring Sets
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is not a feature, then find 1-ring and
merge sets
                    if(setFBV[vi] != 'F') {
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            //if the neighbor of the current vertex is
not a feature then merge sets
                            if(setFBV[&(*tempVertexPointer)] != 'F') {
                                //if they are not already in the same
set, merge them
                                if(DisjointSetsToColor->FindSet(&(*vi))
!= DisjointSetsToColor->FindSet(tempVertexPointer)){
                                    DisjointSetsToColor-
>Union(&(*vi),tempVertexPointer);
                                }
                            }
                        }
                    }
                }
                //Find out how many distinct segment sets there are
*****************************
                int segmentSetCounter = 0;
                vector <CVertexO*> parentVertexofDisjointSets;
                //***** If a vertex is the parent of a DisjointSet, it
is a new Segment
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is not in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        if (DisjointSetsToColor->FindSet(&(*vi)) ==
&(*vi)){
```

```
                            //parent of a new set
parentVertexofDisjointSets.push_back(&(*vi));
                            segmentSetCounter++;
                        }
                    }
                }
                //******* find biggest set
                typedef  std::vector < std::vector <CVertexO*> >
crmatrix;
                crmatrix
vectorOfDisjointSetsWithVertices(segmentSetCounter,
std::vector<CVertexO*>(0));
                vector<CVertexO*>::iterator itVect;
                // ****************************  Create a vector of
vectors containing our sets / vertices
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        //find offset in parent vector--that is the
vertex set number
                        itVect =
find(parentVertexofDisjointSets.begin(),
parentVertexofDisjointSets.end(), DisjointSetsToColor-
>FindSet(&(*vi)));
                        if( itVect != parentVertexofDisjointSets.end()
) {
                            int offset =
std::distance(parentVertexofDisjointSets.begin(), itVect);
                            //store the set number in the vertex
                            FSetNum[vi] = offset;

vectorOfDisjointSetsWithVertices[offset].push_back(&(*vi));
                        }
                    }
                }
                //*********** For each set, sum the positive curvature
and store in new vector
                vector<float>
vectorOfSetPosCurvatureSums(vectorOfDisjointSetsWithVertices.size(),0);
                for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                {
                    float tempSumOfCurvature = 0.0;
                    for(int numInSet=0; numInSet <
vectorOfDisjointSetsWithVertices[pos].size(); numInSet++){
                        CVertexO* tempVertex =
vectorOfDisjointSetsWithVertices[pos][numInSet];
                        float tempCurvature = curvature[tempVertex];
                        //if it's positive, add it up
                        if(tempCurvature > 0.0) {
                            tempSumOfCurvature = tempSumOfCurvature +
tempCurvature;
                        }
                    }
```

```
                    vectorOfSetPosCurvatureSums[pos] =
tempSumOfCurvature;
                }
                //Find the set with the most vertices
                int maxInSet = 0;
                int maxOffset = 0;
                int secondBiggest = 0;
                int secondBiggestOffset = 0;
                for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                {
                    if(vectorOfDisjointSetsWithVertices[pos].size() >
maxInSet) {
                        secondBiggest = maxInSet;
                        secondBiggestOffset = maxOffset;
                        maxInSet =
vectorOfDisjointSetsWithVertices[pos].size();
                        maxOffset = pos;
                    }
                }
                int numBigSets = 0;
                int numLittleSets = 0;
                for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                {
                    if(pos != maxOffset) {
                        if(vectorOfDisjointSetsWithVertices[pos].size()
> (deletePercentOfBiggestSegment * secondBiggest)){
                            numBigSets++;
                        }
                        else{
                            numLittleSets++;
                        }
                    }
                }
                //color the different sets
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F') {
                        (*vi).C() = Color4b(Color4b::Blue);
                    }
                    //if it is not in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        if(FSetNum[vi] == maxOffset) {
                            (*vi).C() = Color4b(Color4b::White);
                        }
                        else {
                            //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                            int ScatterSize =
vectorOfDisjointSetsWithVertices.size();
                            Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                            (*vi).C()=BaseColor;
                        }
```

```
                    }
                }
                delete DisjointSetsToColor;
            }//show segments
            if(removeLines1st) {
                //Check if color same on both sides of F line
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F') {
                        (*vi).SetV();
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        (*vi).ClearV();
                        Color4b color1;
                        Color4b color2;
                        Color4b color3;
                        Color4b whiteColor = Color4b(Color4b::White);
                        Color4b blueColor = Color4b(Color4b::Blue);
                        Color4b tempColor;
                        int colorCounter = 0;
                        CVertexO* tempVertexPointer;
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            if (i == 0) {
                                tempVertexPointer = OneRing.allV.at(i);
                                color1 = (*tempVertexPointer).C();
                                colorCounter++;
                            }
                            if(i > 0 && i < OneRing.allV.size()) {
                                tempVertexPointer = OneRing.allV.at(i);
                                tempColor = (*tempVertexPointer).C();
                                if(colorCounter == 1) {
                                    if(tempColor != color1) {
                                        color2 = tempColor;
                                        colorCounter++;
                                    }
                                }
                                if(colorCounter == 2) {
                                    if(tempColor != color1 && tempColor
!= color2) {
                                        color3 = tempColor;
                                        colorCounter++;
                                    }
                                }
                            }
                        }
                        //delete it from feature
                        if(colorCounter == 2 || colorCounter == 1){
                            if(colorCounter == 2){
                                if(color1 != whiteColor && color2 !=
whiteColor) {
                                    //delete from feature if surrounded
by a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                    if(color2 != blueColor) {
```

```
                                        (*vi).C() = color2;
                                    }
                                }
                            }
                            if(colorCounter == 1) {
                                if(color1 != whiteColor) {
                                    //delete from feature if surrounded
by a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                }
                            }
                        }
                    }
                }//check if color on both sides of line same
            }//remove lines
            //******************************** Mark the borders after
deleting features
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is a feature, then find 1-ring and mark set 'B'
                if(setFBV[vi] == 'F') {
                    vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*vi));
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                        if(setFBV[&(*tempVertexPointer)] == 'V'){
                            //if the neighbor of the 'F' vertex is just
a vertex it is on the border
                            setFBV[&(*tempVertexPointer)] = 'B';
                        }
                    }
                }
            }
            // ******************************  Estimate distance from
feature region
            tri::Geo<CMeshO> g;
            //create a vector of starting vertices in set 'B' (border
of feature)
            std::vector<CVertexO*> fro;
            bool ret;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi)
                if( setFBV[vi] == 'B')
                    fro.push_back(&(*vi));
            if(!fro.empty()) {
                ret = g.DistanceFromFeature(m.cm, fro,
distanceConstraint);
            }
            float maxdist = distanceConstraint;
            float mindist = 0;
            //the distance is now stored in the quality, transfer the
quality to the distance
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                if((*vi).Q() < distanceConstraint) {
                    if( setFBV[vi] == 'F') {
```

```
                        disth[vi] = -((*vi).Q());
                    }
                    if( setFBV[vi] == 'B') {
                        disth[vi] = 0;
                    }
                    if( setFBV[vi] == 'V') {
                        disth[vi] = (*vi).Q();
                    }
                }
                else {
                    disth[vi] = std::numeric_limits<float>::max();
                }
            }
            //filter the distances
            if(filterOn){
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //new distance to vertex is average of neighborhood
distances
                    if( disth[vi] < maxdist) {
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        float numberOfNeigbors = 0.0;
                        float sumOfDistances = 0.0;
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            float tempDist =
disth[(*tempVertexPointer)];
                            if(tempDist !=
std::numeric_limits<float>::max()) {
                                sumOfDistances = sumOfDistances +
tempDist;
                                numberOfNeigbors++;
                            }
                        }
                        float inverseNumNeighbors = 1.0 /
numberOfNeigbors;
                        disth[vi] = inverseNumNeighbors *
sumOfDistances;
                    }
                }
            }
            /****************************** Connecting points by angle
*************************/
            vector <CVertexO*> vectCPoints;
            int CPointCounter = 0;
            int vertexCounter = 0;
            if(findConnectingPoints) {
                //Find Connecting Points
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if( setFBV[vi] == 'F' ) {
                        (*vi).SetV();
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        (*vi).ClearV();
```

```
                        char currentSet = 'a';
                        char initialSet = 'a';
                        int numberOfChanges = 0;
                        float numberOfNeighbors = 0.0;
                        float sumOfDivergence = 0.0;
                        float angle = 0.0;
                        bool addAngle = false;
                        CVertexO* firstVertex;
                        CVertexO* previousVertex;
                        char previousSet = 'a';
                        char firstSet = 'a';
                        int borderNeighborCounter = 0;
                        int featureNeighborCounter = 0;
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            char tempSetOfThisNeighbor =
(setFBV[&(*tempVertexPointer)]);
                            if(tempSetOfThisNeighbor == 'F') {
                                featureNeighborCounter++;
                                //don't add angle, we are still in 'F'
                                addAngle = false;
                            }
                            if(tempSetOfThisNeighbor == 'B'){
                                borderNeighborCounter++;
                                //add angle, we are still in 'F'
                                addAngle = true;
                            }
                            if(i == 0){
                                //initialize the starting set
                                currentSet =
setFBV[&(*tempVertexPointer)];
                                initialSet =
setFBV[&(*tempVertexPointer)];
                                firstVertex = tempVertexPointer;
                                firstSet =
setFBV[&(*tempVertexPointer)];
                                previousVertex = tempVertexPointer;
                                previousSet =
setFBV[&(*tempVertexPointer)];
                            }
                            if(tempSetOfThisNeighbor != currentSet){
                                //changed sets, update the
numberOfChanges
                                currentSet =
setFBV[&(*tempVertexPointer)];
                                numberOfChanges++;
                            }
                            //check if last point is in same set as
first point
                            if(i == (OneRing.allV.size() - 1)) {
                                if(tempSetOfThisNeighbor !=
initialSet){
                                    numberOfChanges++;
                                }
                                if(firstSet == 'F' && currentSet ==
'F'){
```

```
                                    //don't add the angle
                                }
                                else{
                                    //calculate and add the angle
between tempVertexPointer and firstVertex
                                    vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                    //vector from gradient field of w
                                    vcg::Point3f v2 =
(*firstVertex).P() - (*vi).P();
                                    double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                    double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                    v1 = v1 * (1.0/v1_magnitude);
                                    v2 = v2 * (1.0/v2_magnitude);
                                    double cosTheta = 0.0;
                                    if(v1_magnitude*v2_magnitude == 0)
{
                                        cosTheta = 0.0;
                                    }
                                    else{
                                        cosTheta = v1.X()*v2.X() +
v1.Y()*v2.Y() + v1.Z()*v2.Z();
                                    }
                                    float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                    angle = angle + tempAngleinDegrees;
                                }
                            }
                            if(i > 0){
                                if(previousSet == 'F' && currentSet ==
'F'){
                                    //don't add the angle
                                }
                                else{
                                    //calculate and add the angle
between previousVertex and tempVertexPointer
                                    vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                    //vector from gradient field of w
                                    vcg::Point3f v2 =
(*previousVertex).P() - (*vi).P();
                                    double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                    double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                    v1 = v1 * (1.0/v1_magnitude);
                                    v2 = v2 * (1.0/v2_magnitude);
                                    double cosTheta = 0.0;
                                    if(v1_magnitude*v2_magnitude == 0)
{
                                        cosTheta = 0.0;
                                    }
                                    else{
                                        cosTheta = v1.X()*v2.X() +
v1.Y()*v2.Y() + v1.Z()*v2.Z();
```

```
                                    }
                                    float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                    angle = angle + tempAngleinDegrees;
                                }
                                //update the pointers
                                previousVertex = tempVertexPointer;
                                previousSet =
setFBV[&(*tempVertexPointer)];
                            }
                            float tempDivergence =
divfval[&(*tempVertexPointer)];
                            sumOfDivergence = sumOfDivergence +
tempDivergence;
                            numberOfNeighbors++;
                        }
                        //Candidate connecting point
                        if(numberOfChanges <= 2){
                            //not element of Fp, may be connecting
point
                            if(angle > connectingPointAngle) {
                                (*vi).C() = Color4b(Color4b::Yellow);
                                CpointOwner[vi] = CPointCounter;
                                source[vi] = &(*vi);
                                vectCPoints.push_back(&(*vi));
                                CPointCounter++;
                            }
                        }
                    }//in set 'F'
                    vertexCounter++;
                }//go through vertices
            }//find connecting points and color them yellow
            //Find Mid Points
            std::vector<CMeshO::VertexPointer>  midPointVector;
            if(findMidPoints){
                //************************************  Find
connecting paths from connecting points
                //setup the seed vector of connecting points
                std::vector<VertDist> seedVec;
                std::vector<CVertexO*>::iterator fi;
                for( fi  = vectCPoints.begin(); fi != vectCPoints.end()
; ++fi)
                {
                    seedVec.push_back(VertDist(*fi,0.0));
                }
                std::vector<VertDist> frontier;
                CMeshO::VertexPointer curr;
                CMeshO::ScalarType unreached  =
std::numeric_limits<CMeshO::ScalarType>::max();
                CMeshO::VertexPointer pw;
                TempDataType TD(m.cm.vert, unreached);
                std::vector <VertDist >::iterator ifr;
                for(ifr = seedVec.begin(); ifr != seedVec.end();
++ifr){
                    TD[(*ifr).v].d = 0.0;
                    (*ifr).d = 0.0;
                    TD[(*ifr).v].source  = (*ifr).v;
```

```
                    frontier.push_back(VertDist((*ifr).v,0.0));
                }
                // initialize Heap
                make_heap(frontier.begin(),frontier.end(),pred());
                std::vector<int> doneConnectingPoints;
                CMeshO::ScalarType curr_d,d_curr = 0.0,d_heap;
                CMeshO::VertexPointer curr_s = NULL;
                CMeshO::PerVertexAttributeHandle
<CMeshO::VertexPointer> * vertSource = NULL;
                CMeshO::ScalarType max_distance=0.0;
                std::vector<VertDist >:: iterator iv;
                int connectionCounter = 0;
                while(!frontier.empty() && max_distance < maxdist)
                {
                    pop_heap(frontier.begin(),frontier.end(),pred());
                    curr = (frontier.back()).v;
                    int tempCpointOwner = CpointOwner[curr];
                    int tempFSetNum = FSetNum[curr];
                    curr_s = TD[curr].source;
                    if(vertSource!=NULL)
                        (*vertSource)[curr] = curr_s;
                    d_heap = (frontier.back()).d;
                    frontier.pop_back();
                    assert(TD[curr].d <= d_heap);
                    assert(curr_s != NULL);
                    if(TD[curr].d < d_heap )// a vertex whose distance
has been improved after it was inserted in the queue
                        continue;
                    assert(TD[curr].d == d_heap);
                    d_curr =  TD[curr].d;
                    if(d_curr > max_distance) {
                        max_distance = d_curr;
                    }
                    //check the vertices around the current point
                    (*curr).SetV();
                    vcg::tri::Nring<CMeshO> OneRing(&(*curr), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*curr));
                    (*curr).ClearV();
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        pw = OneRing.allV.at(i);
                        //just find the shortest distance between
points
                        curr_d = d_curr + vcg::Distance(pw->cP(),curr-
>cP());
                        //check if we are still searching from this
connecting point
                        std::vector<int>::iterator it;
                        it = std::find(doneConnectingPoints.begin(),
doneConnectingPoints.end(), CpointOwner[curr]);
                        //we are still searching from this connecting
point
                        if (it == doneConnectingPoints.end()){
                            //This point has been explored by a
different Cpoint before
                            //deleted the bit about not connecting to
your own set: && FSetNum[pw] != FSetNum[curr]
```

```
                            if(CpointOwner[pw] != CpointOwner[curr] &&
CpointOwner[pw] != -1 && setFBV[&(*pw)] != 'F'){
                                (*pw).C() = Color4b(Color4b::Magenta);
                                connectionCounter++;
                                source1[pw] = curr;
                                //store the midpoint so we can make the
connecting path later
                                midPointVector.push_back(pw);
                                //add CpointOwner to vector of
doneConnectingPoints

doneConnectingPoints.push_back(CpointOwner[curr]);

doneConnectingPoints.push_back(CpointOwner[pw]);
                            }
                            //deleted  && curvature[pw] <
maxCurvatureInPath
                            else if(TD[(pw)].d > curr_d && curr_d <
maxdist && setFBV[&(*pw)] != 'F'){
                                //This point has not been explored
before, keep looking
                                //update source, Fsetnum, CpointOwner
                                CpointOwner[pw] = tempCpointOwner;
                                source[pw] = curr;
                                FSetNum[pw] = tempFSetNum;
                                TD[(pw)].d = curr_d;
                                TD[pw].source = curr;

frontier.push_back(VertDist(pw,curr_d));

push_heap(frontier.begin(),frontier.end(),pred());
                            }
                        }
                        else {
                            //not searching from this connecting point
anymore
                        }
                    }
                }// end while
            }//find mid points and color magenta
            if(findMidPoints && findPathstoConnect) {
                //********************** Make the connecting path
                vector<CMeshO::VertexPointer>::iterator iter;
                for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                    //track back source to beginning
                    CMeshO::VertexPointer tempVertexPointer = *iter;
                    CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                    while(tempVertexPointer != tempVertexSource){
                        (*tempVertexPointer).C() =
Color4b(Color4b::Green);
                        // setFBV[tempVertexPointer] = 'F';
                        tempVertexPointer = tempVertexSource;
                        tempVertexSource = source[tempVertexPointer];
                    }
                    //track back source1 to beginning
```

```
                    tempVertexPointer = *iter;
                    tempVertexSource = source1[tempVertexPointer];
                    while(tempVertexPointer != tempVertexSource){
                        (*tempVertexPointer).C() =
Color4b(Color4b::Cyan);
                        //setFBV[tempVertexPointer] = 'F';
                        tempVertexPointer = tempVertexSource;
                        tempVertexSource = source[tempVertexPointer];
                    }
                }//make connecting path
            }//find paths to connect and color green and cyan
            if(addPathstoFeatureRegion){
                //********************** Make the connecting path
                vector<CMeshO::VertexPointer>::iterator iter;
                for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                    //track back source to beginning
                    CMeshO::VertexPointer tempVertexPointer = *iter;
                    CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                    while(tempVertexPointer != tempVertexSource){
                        (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                        setFBV[tempVertexPointer] = 'F';
                        newF[tempVertexPointer] = 1;
                        tempVertexPointer = tempVertexSource;
                        tempVertexSource = source[tempVertexPointer];
                    }
                    //track back source1 to beginning
                    tempVertexPointer = *iter;
                    tempVertexSource = source1[tempVertexPointer];
                    while(tempVertexPointer != tempVertexSource){
                        (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                        setFBV[tempVertexPointer] = 'F';
                        newF[tempVertexPointer] = 1;
                        tempVertexPointer = tempVertexSource;
                        tempVertexSource = source[tempVertexPointer];
                    }
                }
            }//add paths to feature region
            //******************************** get ready to
skeletonize and prune by changing sets to either F or V
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is not a feature
                if(setFBV[vi] != 'F') {
                    setFBV[vi] = 'V';
                }
            }
            //***************************** Skeletonize and prune
            //Apply Skeletonize operator
            if(skeletonizeAfter){
                bool changes = false;
                //if still changes loop through and skeletonize
                do {
                    changes = false;
```

```
                    //go through the vertices and if 'F' check for
centers and discs
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            //check for centers and discs
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            bool neighborsAllF = true;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature, vi not a center
                                if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                    neighborsAllF = false;
                                }
                            }
                            if(neighborsAllF){
                                //mark as a center
                                center[vi] = 1;
                                //mark all neighbors as disc
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    disc[&(*tempVertexPointer)] = 1;
                                }
                            }
                        }//in set F
                    }//go through vertices
                    //go through vertices again, if not a center and is
a disc, check complexity and delete if not complex
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F' && center[vi] == 0 &&
disc[vi] == 1){
                            //check complexity
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            int numberOfChanges = 0;
                            char iVertex;
                            char iPlus1Vertex;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if(i < (OneRing.allV.size() - 1)) {
                                    tempVertexPointer =
OneRing.allV.at(i);
```

```
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(i+1);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                else {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(0);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                //change noted
                                if(iVertex != iPlus1Vertex){
                                    numberOfChanges++;
                                }
                            }
                            //not complex, delete it from feature
                            if(numberOfChanges < 4){
                                //delete from feature
                                setFBV[vi] = 'V';
                                center[vi] = 0;
                                disc[vi] = 0;
                                changes = true;
                            }
                        }//not a center, yes a disc
                    }//go through vertices
                    //******************************** reset disc and
center data for next iteration
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is a feature, reset center and disc
data
                        if(setFBV[vi] == 'F') {
                            center[vi] = 0;
                            disc[vi] = 0;
                        }
                    }
                } while (changes == true);//loop if changes
            }//skeletonize operator
            //prune
            if(pruneAfter){
                //pune a certain number of times
                for(int pruneIter = 0; pruneIter < pruneAfterLength;
pruneIter++){
                    int pruned = 0;
                    vertexCounter = 0;
                    //go through the vertices and if 'F' check
complexity
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F'){
```

```
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i < OneRing.allV.size();
i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //prune from feature
                                    setFBV[vi] = 'V';
                                    pruned++;
                                }
                            }//set
                            vertexCounter++;
                        }//go through vertices
                    }//number of times we prune
                }//prune after
                /*
                if(showFinalFeature) {
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                          (*vi).C() = Color4b(Color4b::Blue);
                        }
                        if(setFBV[vi] != 'F') {
                          (*vi).C() = Color4b(Color4b::White);
```

```
                    }
                }
            }
            */
            if(showSegments){
                //************************* Color the different
segments 2nd Time
                //***** New DisjointSet
                vcg::DisjointSet<CVertexO>* ptrDsetSegment = new
vcg::DisjointSet<CVertexO>();
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //Every vertex not in set 'F' starts as disjoint
set
                    if(setFBV[vi] != 'F') {
                        //put the vertex in the set D
                        ptrDsetSegment->MakeSet(&(*vi));
                    }
                }
                //***** Merge neighboring Sets
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is not a feature, then find 1-ring and
merge sets
                    if(setFBV[vi] != 'F') {
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            //if the neighbor of the current vertex is
not a feature then merge sets
                            if(setFBV[&(*tempVertexPointer)] != 'F') {
                                //if they are not already in the same
set, merge them
                                if(ptrDsetSegment->FindSet(&(*vi)) !=
ptrDsetSegment->FindSet(tempVertexPointer)){
                                    ptrDsetSegment-
>Union(&(*vi),tempVertexPointer);
                                }
                            }
                        }
                    }
                }
                //Find out how many distinct segment sets there are
*****************************
                int segmentSetCounter = 0;
                vector <CVertexO*> parentVertexofSegmentSet;
                //***** If a vertex is the parent of a DisjointSet, it
is a new Segment
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is not in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        if (ptrDsetSegment->FindSet(&(*vi)) == &(*vi)){
                            //parent of a new set
```

```
                            parentVertexofSegmentSet.push_back(&(*vi));
                            segmentSetCounter++;
                        }
                    }
                }
                //******* find biggest set
                typedef  std::vector < std::vector <CVertexO*> >
crmatrix;
                crmatrix vectorOfSetsWithVertices(segmentSetCounter,
std::vector<CVertexO*>(0));
                // ******************************  Create a vector of
vectors containing our sets / vertices
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        //find offset in parent vector--that is the
vertex set number
                        itVect = find(parentVertexofSegmentSet.begin(),
parentVertexofSegmentSet.end(), ptrDsetSegment->FindSet(&(*vi)));
                        if( itVect != parentVertexofSegmentSet.end() )
{
                            int offset =
std::distance(parentVertexofSegmentSet.begin(), itVect);
                            //store the set number in the vertex
                            FSetNum[vi] = offset;

vectorOfSetsWithVertices[offset].push_back(&(*vi));
                        }
                    }
                }
                //Find the set with the most vertices
                int maxInSet = 0;
                int maxOffset = 0;
                int secondBiggest = 0;
                int secondBiggestOffset = 0;
                for(int pos=0; pos < vectorOfSetsWithVertices.size();
pos++)
                {
                    if(vectorOfSetsWithVertices[pos].size() > maxInSet)
{
                        secondBiggest = maxInSet;
                        secondBiggestOffset = maxOffset;
                        maxInSet =
vectorOfSetsWithVertices[pos].size();
                        maxOffset = pos;
                    }
                }
                int numBigSets = 0;
                int numLittleSets = 0;
                for(int pos=0; pos < vectorOfSetsWithVertices.size();
pos++)
                {
                    if(pos != maxOffset) {
                        if(vectorOfSetsWithVertices[pos].size() >
(deletePercentOfBiggestSegment * secondBiggest)){
```

```
                            numBigSets++;
                        }
                        else{
                            numLittleSets++;
                        }
                    }
                }
                totalNumBigSets = numBigSets;
                totalNumLittleSets = numLittleSets;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F') {
                        (*vi).C() = Color4b(Color4b::Blue);
                    }
                    //if it is not in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        if(FSetNum[vi] == maxOffset) {
                            (*vi).C() = Color4b(Color4b::White);
                        }
                        else {
                            //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                            int ScatterSize =
vectorOfSetsWithVertices.size();
                            Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                            (*vi).C()=BaseColor;
                        }
                    }
                }
                delete ptrDsetSegment;
            }//show segments
            if(removeLines) {
                //Check if color same on both sides of F line
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F') {
                        (*vi).SetV();
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        (*vi).ClearV();
                        Color4b color1;
                        Color4b color2;
                        Color4b color3;
                        Color4b blueColor = Color4b(Color4b::Blue);
                        Color4b tempColor;
                        int colorCounter = 0;
                        CVertexO* tempVertexPointer;
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            if (i == 0) {
                                tempVertexPointer = OneRing.allV.at(i);
                                color1 = (*tempVertexPointer).C();
                                colorCounter++;
                            }
                            if(i > 0 && i < OneRing.allV.size()) {
```

```
                                    tempVertexPointer = OneRing.allV.at(i);
                                    tempColor = (*tempVertexPointer).C();
                                    if(colorCounter == 1) {
                                        if(tempColor != color1) {
                                            color2 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                    if(colorCounter == 2) {
                                        if(tempColor != color1 && tempColor
!= color2) {
                                            color3 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                }
                            }
                            //delete it from feature
                            if(colorCounter == 2 || colorCounter == 1){
                                if(colorCounter == 2){
                                    //delete from feature if surrounded by
a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                    if(color2 != blueColor) {
                                        (*vi).C() = color2;
                                    }
                                }
                                if(colorCounter == 1) {
                                    //delete from feature if surrounded by
a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                }
                            }
                        }
                    }//check if color on both sides of line same
                }//remove lines
                //binary search stuff
                if (numTeeth > totalNumBigSets)
                    first = intEpsilon + 1;  // repeat search in top half.
                else if (numTeeth < totalNumBigSets)
                    last = intEpsilon - 1; // repeat search in bottom half.
                else
                    foundIt = true;
            }//while first <= last
            if(foundIt == false)
                if(foundIt == true) {
                }
            if(foundIt == true) {
                //now, if we found it, start moving epsilon smaller till we
know we have the smallest value
                do {
```

```
                //try the next smallest epsilon
                epsilon = epsilon - .01;
                /* ********************  Calculate curvature:
curvature.h computes the discrete gaussian curvature.
     <vcg/complex/algorithms/update/curvature.h>
     For further details, please, refer to:
     Discrete Differential-Geometry Operators for Triangulated 2-
Manifolds Mark Meyer,
     Mathieu Desbrun, Peter Schroder, Alan H. Barr VisMath '02, Berlin
</em>*/
                // results stored in (*vi).Kh() (mean) and (*vi).Kg()
(guassian)
                tri::UpdateCurvature<CMeshO>::MeanAndGaussian(m.cm);
                // ********************  Put the curvature in the
quality: <vcg/complex/algorithms/update/quality.h>

tri::UpdateQuality<CMeshO>::VertexFromMeanCurvature(m.cm);
                //****************** ComputePerVertexQualityMinMax 15
and 85 percent
                std::pair<float,float> minmax =
std::make_pair(std::numeric_limits<float>::max(),-
std::numeric_limits<float>::max());
                CMeshO::VertexIterator vi;
                std::vector<float> QV;
                QV.reserve(m.cm.vn);
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                    if(!(*vi).IsD()) QV.push_back((*vi).Q());
                //bottom 15% maps to -1

std::nth_element(QV.begin(),QV.begin()+15*(m.cm.vn/100),QV.end());
                float newmin=*(QV.begin()+m.cm.vn/100);
                //top 15% maps to 1
                std::nth_element(QV.begin(),QV.begin()+m.cm.vn-
15*(m.cm.vn/100),QV.end());
                float newmax=*(QV.begin()+m.cm.vn-m.cm.vn/100);
                minmax.first = newmin;
                minmax.second = newmax;
                // ************************
MapCurvatureRangetoOneNegOne
                float B = 8.0;
                float min = minmax.first;
                float max = minmax.second;
                float upperBound = 0.0;
                if(min < 0)
                    min = min * -1;
                if(min > max)
                    upperBound = min;
                else
                    upperBound = max;
                float K = 0.5 * log10((1 + ((pow(2.0,B)-2)/(pow(2.0,B)-
1))) / ((1 - ((pow(2.0,B)-2)/(pow(2.0,B)-1)))));
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                    if(!(*vi).IsD())
                    {
```

```
                        float remapped = tanh((*vi).Q() * (K /
upperBound));
                        (*vi).Q() = remapped;
                        //Log( "Value to be remapped: %f   remapped:
%f\n", (*vi).Q(), remapped );
                        // Log( "inside mapping");
                    }
                vcg::DisjointSet<CVertexO>* ptrDset = new
vcg::DisjointSet<CVertexO>();
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //all vertices start as set V for Vertex and
unmarked
                    setFBV[vi] = 'V';
                    marked[vi] = 0;
                    disc[vi] = 0;
                    center[vi] = 0;
                    complex[vi] = 0;
                    newF[vi] = 0;
                    //all vertices start as -1 FSetNum
                    FSetNum[vi] = -1;
                    //none have been visited to connect C points, so
set CpointOwner to -1
                    CpointOwner[vi] = -1;
                    //color it White initially
                    (*vi).C() = Color4b(Color4b::White);
                    curvature[vi] = (*vi).Q();
                    //feature region, less than defined min curvature
                    if(curvature[vi] < epsilon) {
                        //in set F for feature
                        setFBV[vi] = 'F';
                    }
                }
                // apply morphological opperators dilate and then erode
*****************************
                if(morphologicalOperations){
                    //dilate 1-ring neighbor
                    for(int k = 0; k < 1; k++){
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] != 'V') {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, mark the neighbor to be one
                                    //if it is negative curvature
                                    if(setFBV[&(*tempVertexPointer)] ==
'V') {
                                        //mark it to be 'N'
```

```
                                        marked[&(*tempVertexPointer)] =
1;
                                    }
                                }
                            }
                        }
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it was marked, change it
                            if(marked[vi] == 1){
                                setFBV[vi] = 'F';
                                marked[vi] = 0;
                            }
                        }
                    }//grow once
                    //erode 1-ring neighbor
                    for(int k = 0; k < 1; k++){
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] != 'V') {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                bool neighborsAllF = true;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, we will erode this vertex
                                    if(setFBV[&(*tempVertexPointer)] ==
'V') {
                                        neighborsAllF = false;
                                    }
                                }
                                if(neighborsAllF){
                                    //mark to not erode this vertex
this round
                                    marked[vi] = 1;
                                }
                            }
                        }
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it was marked to not erode, keep it
                            if(marked[vi] == 1){
                                //setFBV[vi] = 'F';
                                marked[vi] = 0;
                            }
                            else{
                                setFBV[vi] = 'V';
                            }
                        }
                    }//erode once
                }
```

```
                //Apply Skeletonize operator
                if(skeletonize){
                    int debugCounter = 1;
                    bool changes = false;
                    //if still changes loop through and skeletonize
                    do {
                        changes = false;
                        //go through the vertices and if 'F' check for
centers and discs
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                //check for centers and discs
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                bool neighborsAllF = true;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, vi not a center
                                    if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                        neighborsAllF = false;
                                    }
                                }
                                if(neighborsAllF){
                                    //mark as a center
                                    center[vi] = 1;
                                    //mark all neighbors as disc
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                        disc[&(*tempVertexPointer)] =
1;
                                    }
                                }
                            }//in set F
                        }//go through vertices
                        //go through vertices again, if not a center
and is a disc, check complexity and delete if not complex
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(center[vi] == 0 && disc[vi] == 1){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
```

```
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //delete from feature
                                    setFBV[vi] = 'V';
                                    center[vi] = 0;
                                    disc[vi] = 0;
                                    changes = true;
                                }
                            }//not a center, yes a disc
                        }//go through vertices
                        //******************************** reset disc
and center data for next iteration
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                center[vi] = 0;
                                disc[vi] = 0;
                            }
                        }
                        /* if(debugCounter == 1){
                   changes = false;
                }*/
                    } while (changes == true);//loop if changes
                 }//skeletonize operator
                 //prune
                 if(prune){
                     //prune a certain number of times
```

```
                    for(int pruneIter = 0; pruneIter < pruneLength;
pruneIter++){
                        int pruned = 0;
                        //go through the vertices and if 'F' check
complexity
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F'){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //prune from feature
                                    setFBV[vi] = 'V';
                                    pruned++;
                                }
                            }//checking current vertex
                        }//go through vertices
                    }//number of times we prune
                }//prune before
```

```
                //****************************** Initialize the
Disjoint sets of 'F' and delete any that are small
*****************************
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F'){
                        //put the vertex in the set D
                        ptrDset->MakeSet(&(*vi));
                    }
                }
                /*
        if(showMinAndMaxFeature || onlyShowFeature){
            //go through the vertices and color them according to sets
            for(vi=m.cm.vert.begin();vi!=m.cm.vert.end();++vi) {
                if(setFBV[&(*vi)] == 'F')
                    (*vi).C() = Color4b(Color4b::Blue);
                if(setFBV[&(*vi)] == 'N')
                    (*vi).C() = Color4b(Color4b::Green);
                  //more than defined maximum curvature
               // if(!onlyShowFeature) {
                 //   if(curvature[vi] > eta)
                   //    (*vi).C() = Color4b(Color4b::Red);
               // }
            }
        }
    */
                //print number of sets
                //ptrDset->printNumberOfSets();
                //Go through vertices again--if in set 'F',
*****************************
                //then find 1-Ring of vertices and merge sets if
orginal vertex is 'F'
                //and new 1-ring vertex is 'F', define set 'B'
                int blueCounter = 0;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is a feature, then find 1-ring and merge
sets
                    if(setFBV[vi] == 'F') {
                        blueCounter++;
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            //if the neighbor of the current vertex is
a feature then merge sets
                            if(setFBV[&(*tempVertexPointer)] == 'F') {
                                //if they are not already in the same
set, merge them
                                if(ptrDset->FindSet(&(*vi)) != ptrDset-
>FindSet(tempVertexPointer)){
                                    ptrDset-
>Union(&(*vi),tempVertexPointer);
                                }
                            }
```

```
                            else if(setFBV[&(*tempVertexPointer)] ==
'V'){
                                //if the neighbor of the blue vertex is
just a vertex it is on the border
                                //we do this after we delete the small
sets of 'F'
                                //setFBV[&(*tempVertexPointer)] = 'B';
                            }
                        }
                    }
                }
                //Find out how many distinct sets there are
*****************************
                int setCounter = 0;
                vector <CVertexO*> parentVertexofSet;
                typedef  std::vector < std::vector <CVertexO*> >
matrix;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is in set 'F', then find what set it
belongs to
                    if(setFBV[vi] == 'F') {
                        if (ptrDset->FindSet(&(*vi)) == &(*vi)){
                            //parent of a new set
                            parentVertexofSet.push_back(&(*vi));
                            setCounter++;
                        }
                    }
                }
                matrix allSetsWithVertices(setCounter,
std::vector<CVertexO*>(0));
                // ****************************  Create a vector of
vectors containing our sets / vertices
                vector<CVertexO*>::iterator itVect;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is in set 'F', then find what set it
belongs to
                    if(setFBV[vi] == 'F') {
                        //find offset in parent vector--that is the
vertex set number
                        itVect = find(parentVertexofSet.begin(),
parentVertexofSet.end(), ptrDset->FindSet(&(*vi)));
                        if( itVect != parentVertexofSet.end() ) {
                            int offset =
std::distance(parentVertexofSet.begin(), itVect);
                            //store the set number in the vertex
                            FSetNum[vi] = offset;

allSetsWithVertices[offset].push_back(&(*vi));
                        }
                    }
                }
                //delete the disjoint set
                delete ptrDset;
                //print out the stored set data
                /*
```

```
      for(int pos=0; pos < allSetsWithVertices.size(); pos++)
      {
      }*/
                //delete sets of D if the number of vertices is less
than .01 D
                for(int pos=0; pos < allSetsWithVertices.size(); pos++)
                {
                    if(allSetsWithVertices[pos].size() <=
deletePercentOfF*blueCounter) {
                        for(int i=0; i <
allSetsWithVertices[pos].size(); i++) {
                            (*(allSetsWithVertices[pos][i])).C() =
Color4b(Color4b::White);
                            setFBV[(*(allSetsWithVertices[pos][i]))] =
'V';
                        }
                    }
                }
                //************************ Color the different
segments 1st Time and then delete F lines that have same color on both
sides
                //** This clears up a lot of the extraneous connecting
points before we close gaps.
                if(showSegments1st){
                    //***** New DisjointSet
                    vcg::DisjointSet<CVertexO>* DisjointSetsToColor =
new vcg::DisjointSet<CVertexO>();
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //Every vertex not in set 'F' starts as
disjoint set
                        if(setFBV[vi] != 'F') {
                            //put the vertex in the set D
                            DisjointSetsToColor->MakeSet(&(*vi));
                        }
                    }
                    //***** Merge neighboring Sets
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not a feature, then find 1-ring and
merge sets
                        if(setFBV[vi] != 'F') {
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature then merge sets
                                if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                    //if they are not already in the
same set, merge them
                                    if(DisjointSetsToColor-
>FindSet(&(*vi)) != DisjointSetsToColor->FindSet(tempVertexPointer)){
```

```
                                        DisjointSetsToColor-
>Union(&(*vi),tempVertexPointer);
                                    }
                                }
                            }
                        }
                    }
                    //Find out how many distinct segment sets there are
*******************************
                    int segmentSetCounter = 0;
                    vector <CVertexO*> parentVertexofDisjointSets;
                    //***** If a vertex is the parent of a DisjointSet,
it is a new Segment
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if (DisjointSetsToColor->FindSet(&(*vi)) ==
&(*vi)){
                                //parent of a new set

parentVertexofDisjointSets.push_back(&(*vi));
                                segmentSetCounter++;
                            }
                        }
                    }
                    //******* find biggest set
                    typedef  std::vector < std::vector <CVertexO*> >
crmatrix;
                    crmatrix
vectorOfDisjointSetsWithVertices(segmentSetCounter,
std::vector<CVertexO*>(0));
                    vector<CVertexO*>::iterator itVect;
                    // ******************************  Create a vector
of vectors containing our sets / vertices
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is in set 'F', then find what set it
belongs to
                        if(setFBV[vi] != 'F') {
                            //find offset in parent vector--that is the
vertex set number
                            itVect =
find(parentVertexofDisjointSets.begin(),
parentVertexofDisjointSets.end(), DisjointSetsToColor-
>FindSet(&(*vi)));
                            if( itVect !=
parentVertexofDisjointSets.end() ) {
                                int offset =
std::distance(parentVertexofDisjointSets.begin(), itVect);
                                //store the set number in the vertex
                                FSetNum[vi] = offset;

vectorOfDisjointSetsWithVertices[offset].push_back(&(*vi));
                            }
                        }
```

```
                    }
                    //*********** For each set, sum the positive
curvature and store in new vector
                    vector<float>
vectorOfSetPosCurvatureSums(vectorOfDisjointSetsWithVertices.size(),0);
                    for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                    {
                        float tempSumOfCurvature = 0.0;
                        for(int numInSet=0; numInSet <
vectorOfDisjointSetsWithVertices[pos].size(); numInSet++){
                            CVertexO* tempVertex =
vectorOfDisjointSetsWithVertices[pos][numInSet];
                            float tempCurvature =
curvature[tempVertex];
                            //if it's positive, add it up
                            if(tempCurvature > 0.0) {
                                tempSumOfCurvature = tempSumOfCurvature
+ tempCurvature;
                            }
                        }
                        vectorOfSetPosCurvatureSums[pos] =
tempSumOfCurvature;
                    }
                    //Find the set with the most vertices
                    int maxInSet = 0;
                    int maxOffset = 0;
                    int secondBiggest = 0;
                    int secondBiggestOffset = 0;
                    for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                    {
                        if(vectorOfDisjointSetsWithVertices[pos].size()
> maxInSet) {
                            secondBiggest = maxInSet;
                            secondBiggestOffset = maxOffset;
                            maxInSet =
vectorOfDisjointSetsWithVertices[pos].size();
                            maxOffset = pos;
                        }
                    }
                    int numBigSets = 0;
                    int numLittleSets = 0;
                    for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                    {
                        if(pos != maxOffset) {

if(vectorOfDisjointSetsWithVertices[pos].size() >
(deletePercentOfBiggestSegment * secondBiggest)){
                                numBigSets++;
                            }
                            else{
                                numLittleSets++;
                            }
                        }
                    }
```

```
                    //color the different sets
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).C() = Color4b(Color4b::Blue);
                        }
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if(FSetNum[vi] == maxOffset) {
                                (*vi).C() = Color4b(Color4b::White);
                            }
                            else {
                                //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                                int ScatterSize =
vectorOfDisjointSetsWithVertices.size();
                                Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                                (*vi).C()=BaseColor;
                            }
                        }
                    }
                    delete DisjointSetsToColor;
                }//show segments
                if(removeLines1st) {
                    //Check if color same on both sides of F line
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            Color4b color1;
                            Color4b color2;
                            Color4b color3;
                            Color4b whiteColor =
Color4b(Color4b::White);
                            Color4b blueColor = Color4b(Color4b::Blue);
                            Color4b tempColor;
                            int colorCounter = 0;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if (i == 0) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    color1 = (*tempVertexPointer).C();
                                    colorCounter++;
                                }
                                if(i > 0 && i < OneRing.allV.size()) {
                                    tempVertexPointer =
OneRing.allV.at(i);
```

```
                                    tempColor =
(*tempVertexPointer).C();
                                    if(colorCounter == 1) {
                                        if(tempColor != color1) {
                                            color2 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                    if(colorCounter == 2) {
                                        if(tempColor != color1 &&
tempColor != color2) {
                                            color3 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                }
                            }
                            //delete it from feature
                            if(colorCounter == 2 || colorCounter == 1){
                                if(colorCounter == 2){
                                    if(color1 != whiteColor && color2
!= whiteColor) {
                                        //delete from feature if
surrounded by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                        if(color2 != blueColor) {
                                            (*vi).C() = color2;
                                        }
                                    }
                                }
                                if(colorCounter == 1) {
                                    if(color1 != whiteColor) {
                                        //delete from feature if
surrounded by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                    }
                                }
                            }
                        }
                    }//check if color on both sides of line same
                }//remove lines
                //******************************** Mark the borders
after deleting features
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is a feature, then find 1-ring and mark set
'B'
                    if(setFBV[vi] == 'F') {
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        for (int i = 0; i < OneRing.allV.size(); i++){
```

```
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            if(setFBV[&(*tempVertexPointer)] == 'V'){
                                //if the neighbor of the 'F' vertex is
just a vertex it is on the border
                                setFBV[&(*tempVertexPointer)] = 'B';
                            }
                        }
                    }
                }
                // ******************************  Estimate distance
from feature region
                tri::Geo<CMeshO> g;
                //create a vector of starting vertices in set 'B'
(border of feature)
                std::vector<CVertexO*> fro;
                bool ret;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                    if( setFBV[vi] == 'B')
                        fro.push_back(&(*vi));
                if(!fro.empty()) {
                    ret = g.DistanceFromFeature(m.cm, fro,
distanceConstraint);
                }
                float maxdist = distanceConstraint;
                float mindist = 0;
                //the distance is now stored in the quality, transfer
the quality to the distance
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if((*vi).Q() < distanceConstraint) {
                        if( setFBV[vi] == 'F') {
                            disth[vi] = -((*vi).Q());
                        }
                        if( setFBV[vi] == 'B') {
                            disth[vi] = 0;
                        }
                        if( setFBV[vi] == 'V') {
                            disth[vi] = (*vi).Q();
                        }
                    }
                    else {
                        disth[vi] = std::numeric_limits<float>::max();
                    }
                }
                //filter the distances
                if(filterOn){
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //new distance to vertex is average of
neighborhood distances
                        if( disth[vi] < maxdist) {
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            float numberOfNeigbors = 0.0;
```

```
                            float sumOfDistances = 0.0;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                float tempDist =
disth[(*tempVertexPointer)];
                                if(tempDist !=
std::numeric_limits<float>::max()) {
                                    sumOfDistances = sumOfDistances +
tempDist;
                                    numberOfNeigbors++;
                                }
                            }
                            float inverseNumNeighbors = 1.0 /
numberOfNeigbors;
                            disth[vi] = inverseNumNeighbors *
sumOfDistances;
                        }
                    }
                }
                /****************************** Connecting points by
angle *************************/
                vector <CVertexO*> vectCPoints;
                int CPointCounter = 0;
                int vertexCounter = 0;
                if(findConnectingPoints) {
                    //Find Connecting Points
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if( setFBV[vi] == 'F' ) {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            char currentSet = 'a';
                            char initialSet = 'a';
                            int numberOfChanges = 0;
                            float numberOfNeighbors = 0.0;
                            float sumOfDivergence = 0.0;
                            float angle = 0.0;
                            bool addAngle = false;
                            CVertexO* firstVertex;
                            CVertexO* previousVertex;
                            char previousSet = 'a';
                            char firstSet = 'a';
                            int borderNeighborCounter = 0;
                            int featureNeighborCounter = 0;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                char tempSetOfThisNeighbor =
(setFBV[&(*tempVertexPointer)]);
                                if(tempSetOfThisNeighbor == 'F') {
                                    featureNeighborCounter++;
```

```
                                    //don't add angle, we are still in
'F'
                                    addAngle = false;
                                }
                                if(tempSetOfThisNeighbor == 'B'){
                                    borderNeighborCounter++;
                                    //add angle, we are still in 'F'
                                    addAngle = true;
                                }
                                if(i == 0){
                                    //initialize the starting set
                                    currentSet =
setFBV[&(*tempVertexPointer)];
                                    initialSet =
setFBV[&(*tempVertexPointer)];
                                    firstVertex = tempVertexPointer;
                                    firstSet =
setFBV[&(*tempVertexPointer)];
                                    previousVertex = tempVertexPointer;
                                    previousSet =
setFBV[&(*tempVertexPointer)];
                                }
                                if(tempSetOfThisNeighbor !=
currentSet){
                                    //changed sets, update the
numberOfChanges
                                    currentSet =
setFBV[&(*tempVertexPointer)];
                                    numberOfChanges++;
                                }
                                //check if last point is in same set as
first point
                                if(i == (OneRing.allV.size() - 1)) {
                                    if(tempSetOfThisNeighbor !=
initialSet){
                                        numberOfChanges++;
                                    }
                                    if(firstSet == 'F' && currentSet ==
'F'){
                                        //don't add the angle
                                    }
                                    else{
                                        //calculate and add the angle
between tempVertexPointer and firstVertex
                                        vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                        //vector from gradient field of
w
                                        vcg::Point3f v2 =
(*firstVertex).P() - (*vi).P();
                                        double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                        double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                        v1 = v1 * (1.0/v1_magnitude);
                                        v2 = v2 * (1.0/v2_magnitude);
                                        double cosTheta = 0.0;
```

```
                                        if(v1_magnitude*v2_magnitude ==
0) {
                                            cosTheta = 0.0;
                                        }
                                        else{
                                            cosTheta = v1.X()*v2.X() +
v1.Y()*v2.Y() + v1.Z()*v2.Z();
                                        }
                                        float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                        angle = angle +
tempAngleinDegrees;
                                    }
                                }
                                if(i > 0){
                                    if(previousSet == 'F' && currentSet
== 'F'){
                                        //don't add the angle
                                    }
                                    else{
                                        //calculate and add the angle
between previousVertex and tempVertexPointer
                                        vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                        //vector from gradient field of
w
                                        vcg::Point3f v2 =
(*previousVertex).P() - (*vi).P();
                                        double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                        double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                        v1 = v1 * (1.0/v1_magnitude);
                                        v2 = v2 * (1.0/v2_magnitude);
                                        double cosTheta = 0.0;
                                        if(v1_magnitude*v2_magnitude ==
0) {
                                            cosTheta = 0.0;
                                        }
                                        else{
                                            cosTheta = v1.X()*v2.X() +
v1.Y()*v2.Y() + v1.Z()*v2.Z();
                                        }
                                        float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                        angle = angle +
tempAngleinDegrees;
                                    }
                                    //update the pointers
                                    previousVertex = tempVertexPointer;
                                    previousSet =
setFBV[&(*tempVertexPointer)];
                                }
                                float tempDivergence =
divfval[&(*tempVertexPointer)];
                                sumOfDivergence = sumOfDivergence +
tempDivergence;
```

```
                                numberOfNeighbors++;
                            }
                            //Candidate connecting point
                            if(numberOfChanges <= 2){
                                //not element of Fp, may be connecting
point
                                if(angle > connectingPointAngle) {
                                    (*vi).C() =
Color4b(Color4b::Yellow);
                                    CpointOwner[vi] = CPointCounter;
                                    source[vi] = &(*vi);
                                    vectCPoints.push_back(&(*vi));
                                    CPointCounter++;
                                }
                            }
                        }//in set 'F'
                        vertexCounter++;
                    }//go through vertices
                }//find connecting points and color them yellow
                //Find Mid Points
                std::vector<CMeshO::VertexPointer>  midPointVector;
                if(findMidPoints){
                    //************************************  Find
connecting paths from connecting points
                    //setup the seed vector of connecting points
                    std::vector<VertDist> seedVec;
                    std::vector<CVertexO*>::iterator fi;
                    for( fi  = vectCPoints.begin(); fi !=
vectCPoints.end() ; ++fi)
                    {
                        seedVec.push_back(VertDist(*fi,0.0));
                    }
                    std::vector<VertDist> frontier;
                    CMeshO::VertexPointer curr;
                    CMeshO::ScalarType unreached  =
std::numeric_limits<CMeshO::ScalarType>::max();
                    CMeshO::VertexPointer pw;
                    TempDataType TD(m.cm.vert, unreached);
                    std::vector <VertDist >::iterator ifr;
                    for(ifr = seedVec.begin(); ifr != seedVec.end();
++ifr){
                        TD[(*ifr).v].d = 0.0;
                        (*ifr).d = 0.0;
                        TD[(*ifr).v].source  = (*ifr).v;
                        frontier.push_back(VertDist((*ifr).v,0.0));
                    }
                    // initialize Heap
                    make_heap(frontier.begin(),frontier.end(),pred());
                    std::vector<int> doneConnectingPoints;
                    CMeshO::ScalarType curr_d,d_curr = 0.0,d_heap;
                    CMeshO::VertexPointer curr_s = NULL;
                    CMeshO::PerVertexAttributeHandle
<CMeshO::VertexPointer> * vertSource = NULL;
                    CMeshO::ScalarType max_distance=0.0;
                    std::vector<VertDist >:: iterator iv;
                    int connectionCounter = 0;
                    while(!frontier.empty() && max_distance < maxdist)
```

```
                    {

pop_heap(frontier.begin(),frontier.end(),pred());
                        curr = (frontier.back()).v;
                        int tempCpointOwner = CpointOwner[curr];
                        int tempFSetNum = FSetNum[curr];
                        curr_s = TD[curr].source;
                        if(vertSource!=NULL)
                            (*vertSource)[curr] = curr_s;
                        d_heap = (frontier.back()).d;
                        frontier.pop_back();
                        assert(TD[curr].d <= d_heap);
                        assert(curr_s != NULL);
                        if(TD[curr].d < d_heap )// a vertex whose
distance has been improved after it was inserted in the queue
                            continue;
                        assert(TD[curr].d == d_heap);
                        d_curr =  TD[curr].d;
                        if(d_curr > max_distance) {
                            max_distance = d_curr;
                        }
                        //check the vertices around the current point
                        (*curr).SetV();
                        vcg::tri::Nring<CMeshO> OneRing(&(*curr),
&m.cm);
                        OneRing.insertAndFlag1Ring(&(*curr));
                        (*curr).ClearV();
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            pw = OneRing.allV.at(i);
                            //just find the shortest distance between
points
                            curr_d = d_curr + vcg::Distance(pw-
>cP(),curr->cP());
                            //check if we are still searching from this
connecting point
                            std::vector<int>::iterator it;
                            it =
std::find(doneConnectingPoints.begin(), doneConnectingPoints.end(),
CpointOwner[curr]);
                            //we are still searching from this
connecting point
                            if (it == doneConnectingPoints.end()){
                                //This point has been explored by a
different Cpoint before
                                //deleted the bit about not connecting
to your own set: && FSetNum[pw] != FSetNum[curr]
                                if(CpointOwner[pw] != CpointOwner[curr]
&& CpointOwner[pw] != -1 && setFBV[&(*pw)] != 'F'){
                                    //This point has been explored
before, we may have a connection
                                    //(check if the CpointOwner is
active or already connected up with someone else
                                    //std::vector<int>::iterator chk;
                                    //chk =
std::find(doneConnectingPoints.begin(), doneConnectingPoints.end(),
CpointOwner[pw]);
```

```
                                    //we are still searching from the
CpointSource of the point we found
                                    //if(chk ==
doneConnectingPoints.end()) {
                                    (*pw).C() =
Color4b(Color4b::Magenta);
                                    connectionCounter++;
                                    source1[pw] = curr;
                                    //store the midpoint so we can make
the connecting path later
                                    midPointVector.push_back(pw);
                                    //add CpointOwner to vector of
doneConnectingPoints

doneConnectingPoints.push_back(CpointOwner[curr]);

doneConnectingPoints.push_back(CpointOwner[pw]);
                                    // }
                                    /* else{
                                }*/
                                    }
                                    //deleted  && curvature[pw] <
maxCurvatureInPath
                                    else if(TD[(pw)].d > curr_d && curr_d <
maxdist && setFBV[&(*pw)] != 'F'){
                                    //This point has not been explored
before, keep looking
                                    //update source, Fsetnum,
CpointOwner
                                    CpointOwner[pw] = tempCpointOwner;
                                    source[pw] = curr;
                                    FSetNum[pw] = tempFSetNum;
                                    TD[(pw)].d = curr_d;
                                    TD[pw].source = curr;

frontier.push_back(VertDist(pw,curr_d));

push_heap(frontier.begin(),frontier.end(),pred());
                                    }
                                }
                                else {
                                    //not searching from this connecting
point anymore
                                }
                            }
                        }// end while
                    }//find mid points and color magenta
                    if(findMidPoints && findPathstoConnect) {
                        //********************** Make the connecting path
                        vector<CMeshO::VertexPointer>::iterator iter;
                        for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                            //track back source to beginning
                            CMeshO::VertexPointer tempVertexPointer =
*iter;
                            CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
```

```
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Green);
                            // setFBV[tempVertexPointer] = 'F';
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                        //track back source1 to beginning
                        tempVertexPointer = *iter;
                        tempVertexSource = source1[tempVertexPointer];
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Cyan);
                            //setFBV[tempVertexPointer] = 'F';
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                    }//make connecting path
                }//find paths to connect and color green and cyan
                if(addPathstoFeatureRegion){
                    //********************** Make the connecting path
                    vector<CMeshO::VertexPointer>::iterator iter;
                    for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                        //track back source to beginning
                        CMeshO::VertexPointer tempVertexPointer =
*iter;
                        CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                            setFBV[tempVertexPointer] = 'F';
                            newF[tempVertexPointer] = 1;
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                        //track back source1 to beginning
                        tempVertexPointer = *iter;
                        tempVertexSource = source1[tempVertexPointer];
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                            setFBV[tempVertexPointer] = 'F';
                            newF[tempVertexPointer] = 1;
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                    }
                }//add paths to feature region
                //******************************** get ready to
skeletonize and prune by changing sets to either F or V
```

```
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is not a feature
                    if(setFBV[vi] != 'F') {
                        setFBV[vi] = 'V';
                    }
                }
                //**************************** Skeletonize and prune
                //Apply Skeletonize operator
                if(skeletonizeAfter){
                    bool changes = false;
                    //if still changes loop through and skeletonize
                    do {
                        changes = false;
                        //go through the vertices and if 'F' check for
centers and discs
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                //check for centers and discs
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                bool neighborsAllF = true;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, vi not a center
                                    if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                        neighborsAllF = false;
                                    }
                                }
                                if(neighborsAllF){
                                    //mark as a center
                                    center[vi] = 1;
                                    //mark all neighbors as disc
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                        disc[&(*tempVertexPointer)] =
1;
                                    }
                                }
                            }//in set F
                        }//go through vertices
                        //go through vertices again, if not a center
and is a disc, check complexity and delete if not complex
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F' && center[vi] == 0 &&
disc[vi] == 1){
```

```
                                    //check complexity
                                    (*vi).SetV();
                                    vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                    OneRing.insertAndFlag1Ring(&(*vi));
                                    (*vi).ClearV();
                                    int numberOfChanges = 0;
                                    char iVertex;
                                    char iPlus1Vertex;
                                    CVertexO* tempVertexPointer;
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        if(i < (OneRing.allV.size() - 1)) {
                                            tempVertexPointer =
OneRing.allV.at(i);
                                            iVertex =
(setFBV[&(*tempVertexPointer)]);
                                            tempVertexPointer =
OneRing.allV.at(i+1);
                                            iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                        }
                                        else {
                                            tempVertexPointer =
OneRing.allV.at(i);
                                            iVertex =
(setFBV[&(*tempVertexPointer)]);
                                            tempVertexPointer =
OneRing.allV.at(0);
                                            iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                        }
                                        //change noted
                                        if(iVertex != iPlus1Vertex){
                                            numberOfChanges++;
                                        }
                                    }
                                    //not complex, delete it from feature
                                    if(numberOfChanges < 4){
                                        //delete from feature
                                        setFBV[vi] = 'V';
                                        center[vi] = 0;
                                        disc[vi] = 0;
                                        changes = true;
                                    }
                                }//not a center, yes a disc
                            }//go through vertices
                            //******************************** reset disc
and center data for next iteration
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                //if it is a feature, reset center and disc
data
                                if(setFBV[vi] == 'F') {
                                    center[vi] = 0;
                                    disc[vi] = 0;
                                }
```

```
                        }
                    } while (changes == true);//loop if changes
                }//skeletonize operator
                //prune
                if(pruneAfter){
                    //pune a certain number of times
                    for(int pruneIter = 0; pruneIter <
pruneAfterLength; pruneIter++){
                        int pruned = 0;
                        vertexCounter = 0;
                        //go through the vertices and if 'F' check
complexity
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F'){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //prune from feature
                                    setFBV[vi] = 'V';
                                    pruned++;
```

```
                                }
                            }//set
                            vertexCounter++;
                        }//go through vertices
                    }//number of times we prune
                }//prune after
                /*
            if(showFinalFeature) {
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F') {
                      (*vi).C() = Color4b(Color4b::Blue);
                    }
                    if(setFBV[vi] != 'F') {
                      (*vi).C() = Color4b(Color4b::White);
                    }
                }
            }
            */
                if(showSegments){
                    //************************* Color the different
segments 2nd Time
                    //***** New DisjointSet
                    vcg::DisjointSet<CVertexO>* ptrDsetSegment = new
vcg::DisjointSet<CVertexO>();
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //Every vertex not in set 'F' starts as
disjoint set
                        if(setFBV[vi] != 'F') {
                            //put the vertex in the set D
                            ptrDsetSegment->MakeSet(&(*vi));
                        }
                    }
                    //***** Merge neighboring Sets
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not a feature, then find 1-ring and
merge sets
                        if(setFBV[vi] != 'F') {
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature then merge sets
                                if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                    //if they are not already in the
same set, merge them
                                    if(ptrDsetSegment->FindSet(&(*vi))
!= ptrDsetSegment->FindSet(tempVertexPointer)){
                                        ptrDsetSegment-
>Union(&(*vi),tempVertexPointer);
```

```
                                    }
                                }
                            }
                        }
                    }
                    //Find out how many distinct segment sets there are
******************************
                    int segmentSetCounter = 0;
                    vector <CVertexO*> parentVertexofSegmentSet;
                    //***** If a vertex is the parent of a DisjointSet,
it is a new Segment
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if (ptrDsetSegment->FindSet(&(*vi)) ==
&(*vi)){
                                //parent of a new set

parentVertexofSegmentSet.push_back(&(*vi));
                                segmentSetCounter++;
                            }
                        }
                    }
                    //******* find biggest set
                    typedef  std::vector < std::vector <CVertexO*> >
crmatrix;
                    crmatrix
vectorOfSetsWithVertices(segmentSetCounter, std::vector<CVertexO*>(0));
                    // ******************************  Create a vector
of vectors containing our sets / vertices
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is in set 'F', then find what set it
belongs to
                        if(setFBV[vi] != 'F') {
                            //find offset in parent vector--that is the
vertex set number
                            itVect =
find(parentVertexofSegmentSet.begin(), parentVertexofSegmentSet.end(),
ptrDsetSegment->FindSet(&(*vi)));
                            if( itVect !=
parentVertexofSegmentSet.end() ) {
                                int offset =
std::distance(parentVertexofSegmentSet.begin(), itVect);
                                //store the set number in the vertex
                                FSetNum[vi] = offset;

vectorOfSetsWithVertices[offset].push_back(&(*vi));
                            }
                        }
                    }
                    //Find the set with the most vertices
                    int maxInSet = 0;
                    int maxOffset = 0;
                    int secondBiggest = 0;
```

```
                    int secondBiggestOffset = 0;
                    for(int pos=0; pos <
vectorOfSetsWithVertices.size(); pos++)
                    {
                        if(vectorOfSetsWithVertices[pos].size() >
maxInSet) {
                            secondBiggest = maxInSet;
                            secondBiggestOffset = maxOffset;
                            maxInSet =
vectorOfSetsWithVertices[pos].size();
                            maxOffset = pos;
                        }
                    }
                    int numBigSets = 0;
                    int numLittleSets = 0;
                    for(int pos=0; pos <
vectorOfSetsWithVertices.size(); pos++)
                    {
                        if(pos != maxOffset) {
                            if(vectorOfSetsWithVertices[pos].size() >
(deletePercentOfBiggestSegment * secondBiggest)){
                                numBigSets++;
                            }
                            else{
                                numLittleSets++;
                            }
                        }
                    }
                    totalNumBigSets = numBigSets;
                    totalNumLittleSets = numLittleSets;
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).C() = Color4b(Color4b::Blue);
                        }
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if(FSetNum[vi] == maxOffset) {
                                (*vi).C() = Color4b(Color4b::White);
                            }
                            else {
                                //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                                int ScatterSize =
vectorOfSetsWithVertices.size();
                                Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                                (*vi).C()=BaseColor;
                            }
                        }
                    }
                    delete ptrDsetSegment;
                }//show segments
                if(removeLines) {
                    //Check if color same on both sides of F line
```

```
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            Color4b color1;
                            Color4b color2;
                            Color4b color3;
                            Color4b blueColor = Color4b(Color4b::Blue);
                            Color4b tempColor;
                            int colorCounter = 0;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if (i == 0) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    color1 = (*tempVertexPointer).C();
                                    colorCounter++;
                                }
                                if(i > 0 && i < OneRing.allV.size()) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    tempColor =
(*tempVertexPointer).C();
                                    if(colorCounter == 1) {
                                        if(tempColor != color1) {
                                            color2 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                    if(colorCounter == 2) {
                                        if(tempColor != color1 &&
tempColor != color2) {
                                            color3 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                }
                            }
                            //delete it from feature
                            if(colorCounter == 2 || colorCounter == 1){
                                if(colorCounter == 2){
                                    //delete from feature if surrounded
by a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                    if(color2 != blueColor) {
                                        (*vi).C() = color2;
                                    }
                                }
                                if(colorCounter == 1) {
```

```
                                        //delete from feature if surrounded
by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                    }
                                }
                            }
                        }//check if color on both sides of line same
                    }//remove lines
                } while (totalNumBigSets == numTeeth);//while
                epsilon = epsilon + .01;
                /* ********************  Calculate curvature: curvature.h
computes the discrete gaussian curvature.
     <vcg/complex/algorithms/update/curvature.h>
     For further details, please, refer to:
     Discrete Differential-Geometry Operators for Triangulated 2-
Manifolds Mark Meyer,
     Mathieu Desbrun, Peter Schroder, Alan H. Barr VisMath '02, Berlin
</em>*/
                // results stored in (*vi).Kh() (mean) and (*vi).Kg()
(guassian)
                tri::UpdateCurvature<CMeshO>::MeanAndGaussian(m.cm);
                // ********************  Put the curvature in the quality:
<vcg/complex/algorithms/update/quality.h>
                tri::UpdateQuality<CMeshO>::VertexFromMeanCurvature(m.cm);
                //****************** ComputePerVertexQualityMinMax 15 and
85 percent
                std::pair<float,float> minmax =
std::make_pair(std::numeric_limits<float>::max(),-
std::numeric_limits<float>::max());
                CMeshO::VertexIterator vi;
                std::vector<float> QV;
                QV.reserve(m.cm.vn);
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi)
                    if(!(*vi).IsD()) QV.push_back((*vi).Q());
                //bottom 15% maps to -1

std::nth_element(QV.begin(),QV.begin()+15*(m.cm.vn/100),QV.end());
                float newmin=*(QV.begin()+m.cm.vn/100);
                //top 15% maps to 1
                std::nth_element(QV.begin(),QV.begin()+m.cm.vn-
15*(m.cm.vn/100),QV.end());
                float newmax=*(QV.begin()+m.cm.vn-m.cm.vn/100);
                minmax.first = newmin;
                minmax.second = newmax;
                // ************************** MapCurvatureRangetoOneNegOne
                float B = 8.0;
                float min = minmax.first;
                float max = minmax.second;
                float upperBound = 0.0;
                if(min < 0)
                    min = min * -1;
                if(min > max)
                    upperBound = min;
                else
```

```
                upperBound = max;
            float K = 0.5 * log10((1 + ((pow(2.0,B)-2)/(pow(2.0,B)-1)))
/ ((1 - ((pow(2.0,B)-2)/(pow(2.0,B)-1)))));
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi)
                if(!(*vi).IsD())
                {
                    float remapped = tanh((*vi).Q() * (K /
upperBound));
                    (*vi).Q() = remapped;
                    //Log( "Value to be remapped: %f   remapped: %f\n",
(*vi).Q(), remapped );
                    // Log( "inside mapping");
                }
            vcg::DisjointSet<CVertexO>* ptrDset = new
vcg::DisjointSet<CVertexO>();
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //all vertices start as set V for Vertex and unmarked
                setFBV[vi] = 'V';
                marked[vi] = 0;
                disc[vi] = 0;
                center[vi] = 0;
                complex[vi] = 0;
                newF[vi] = 0;
                //all vertices start as -1 FSetNum
                FSetNum[vi] = -1;
                //none have been visited to connect C points, so set
CpointOwner to -1
                CpointOwner[vi] = -1;
                //color it White initially
                (*vi).C() = Color4b(Color4b::White);
                curvature[vi] = (*vi).Q();
                //feature region, less than defined min curvature
                if(curvature[vi] < epsilon) {
                    //in set F for feature
                    setFBV[vi] = 'F';
                }
            }
            // apply morphological opperators dilate and then erode
****************************
            if(morphologicalOperations){
                //dilate 1-ring neighbor
                for(int k = 0; k < 1; k++){
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] != 'V') {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature, mark the neighbor to be one
                                //if it is negative curvature
```

```
                                if(setFBV[&(*tempVertexPointer)] ==
'V') {
                                    //mark it to be 'N'
                                    marked[&(*tempVertexPointer)] = 1;
                                }
                            }
                        }
                    }
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it was marked, change it
                        if(marked[vi] == 1){
                            setFBV[vi] = 'F';
                            marked[vi] = 0;
                        }
                    }
                }//grow once
                //erode 1-ring neighbor
                for(int k = 0; k < 1; k++){
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] != 'V') {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            bool neighborsAllF = true;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature, we will erode this vertex
                                if(setFBV[&(*tempVertexPointer)] ==
'V') {
                                    neighborsAllF = false;
                                }
                            }
                            if(neighborsAllF){
                                //mark to not erode this vertex this
round
                                marked[vi] = 1;
                            }
                        }
                    }
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it was marked to not erode, keep it
                        if(marked[vi] == 1){
                            //setFBV[vi] = 'F';
                            marked[vi] = 0;
                        }
                        else{
                            setFBV[vi] = 'V';
                        }
                    }
```

```
                }//erode once
            }
            //Apply Skeletonize operator
            if(skeletonize){
                int debugCounter = 1;
                bool changes = false;
                //if still changes loop through and skeletonize
                do {
                    changes = false;
                    //go through the vertices and if 'F' check for
centers and discs
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            //check for centers and discs
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            bool neighborsAllF = true;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature, vi not a center
                                if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                    neighborsAllF = false;
                                }
                            }
                            if(neighborsAllF){
                                //mark as a center
                                center[vi] = 1;
                                //mark all neighbors as disc
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    disc[&(*tempVertexPointer)] = 1;
                                }
                            }
                        }//in set F
                    }//go through vertices
                    //go through vertices again, if not a center and is
a disc, check complexity and delete if not complex
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(center[vi] == 0 && disc[vi] == 1){
                            //check complexity
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            int numberOfChanges = 0;
```

```
                            char iVertex;
                            char iPlus1Vertex;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if(i < (OneRing.allV.size() - 1)) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(i+1);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                else {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(0);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                //change noted
                                if(iVertex != iPlus1Vertex){
                                    numberOfChanges++;
                                }
                            }
                            //not complex, delete it from feature
                            if(numberOfChanges < 4){
                                //delete from feature
                                setFBV[vi] = 'V';
                                center[vi] = 0;
                                disc[vi] = 0;
                                changes = true;
                            }
                        }//not a center, yes a disc
                    }//go through vertices
                    //******************************** reset disc and
center data for next iteration
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            center[vi] = 0;
                            disc[vi] = 0;
                        }
                    }
                    /* if(debugCounter == 1){
                  changes = false;
               }*/
                } while (changes == true);//loop if changes
            }//skeletonize operator
            //prune
            if(prune){
                //prune a certain number of times
```

```
                for(int pruneIter = 0; pruneIter < pruneLength;
pruneIter++){
                    int pruned = 0;
                    //go through the vertices and if 'F' check
complexity
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F'){
                            //check complexity
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            int numberOfChanges = 0;
                            char iVertex;
                            char iPlus1Vertex;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if(i < (OneRing.allV.size() - 1)) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(i+1);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                else {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(0);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                //change noted
                                if(iVertex != iPlus1Vertex){
                                    numberOfChanges++;
                                }
                            }
                            //not complex, delete it from feature
                            if(numberOfChanges < 4){
                                //prune from feature
                                setFBV[vi] = 'V';
                                pruned++;
                            }
                        }//checking current vertex
                    }//go through vertices
                }//number of times we prune
            }//prune before
```

```
            //***************************** Initialize the Disjoint
sets of 'F' and delete any that are small
*****************************
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                if(setFBV[vi] == 'F'){
                    //put the vertex in the set D
                    ptrDset->MakeSet(&(*vi));
                }
            }
            /*
        if(showMinAndMaxFeature || onlyShowFeature){
            //go through the vertices and color them according to sets
            for(vi=m.cm.vert.begin();vi!=m.cm.vert.end();++vi) {
                if(setFBV[&(*vi)] == 'F')
                    (*vi).C() = Color4b(Color4b::Blue);
                if(setFBV[&(*vi)] == 'N')
                    (*vi).C() = Color4b(Color4b::Green);
                  //more than defined maximum curvature
               // if(!onlyShowFeature) {
                  //   if(curvature[vi] > eta)
                    //    (*vi).C() = Color4b(Color4b::Red);
               // }
            }
        }
    */
            //print number of sets
            //ptrDset->printNumberOfSets();
            //Go through vertices again--if in set 'F',
*****************************
            //then find 1-Ring of vertices and merge sets if orginal
vertex is 'F'
            //and new 1-ring vertex is 'F', define set 'B'
            int blueCounter = 0;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is a feature, then find 1-ring and merge sets
                if(setFBV[vi] == 'F') {
                    blueCounter++;
                    vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*vi));
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                        //if the neighbor of the current vertex is a
feature then merge sets
                        if(setFBV[&(*tempVertexPointer)] == 'F') {
                            //if they are not already in the same set,
merge them
                            if(ptrDset->FindSet(&(*vi)) != ptrDset-
>FindSet(tempVertexPointer)){
                                ptrDset-
>Union(&(*vi),tempVertexPointer);
                            }
                        }
                        else if(setFBV[&(*tempVertexPointer)] == 'V'){
                            //if the neighbor of the blue vertex is
just a vertex it is on the border
```

```
                            //we do this after we delete the small sets
of 'F'
                            //setFBV[&(*tempVertexPointer)] = 'B';
                        }
                    }
                }
            }
            //Find out how many distinct sets there are
*****************************
            int setCounter = 0;
            vector <CVertexO*> parentVertexofSet;
            typedef  std::vector < std::vector <CVertexO*> > matrix;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is in set 'F', then find what set it belongs to
                if(setFBV[vi] == 'F') {
                    if (ptrDset->FindSet(&(*vi)) == &(*vi)){
                        //parent of a new set
                        parentVertexofSet.push_back(&(*vi));
                        setCounter++;
                    }
                }
            }
            matrix allSetsWithVertices(setCounter,
std::vector<CVertexO*>(0));
            // ******************************  Create a vector of
vectors containing our sets / vertices
            vector<CVertexO*>::iterator itVect;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is in set 'F', then find what set it belongs to
                if(setFBV[vi] == 'F') {
                    //find offset in parent vector--that is the vertex
set number
                    itVect = find(parentVertexofSet.begin(),
parentVertexofSet.end(), ptrDset->FindSet(&(*vi)));
                    if( itVect != parentVertexofSet.end() ) {
                        int offset =
std::distance(parentVertexofSet.begin(), itVect);
                        //store the set number in the vertex
                        FSetNum[vi] = offset;
                        allSetsWithVertices[offset].push_back(&(*vi));
                    }
                }
            }
            //delete the disjoint set
            delete ptrDset;
            //delete sets of D if the number of vertices is less than
.01 D
            for(int pos=0; pos < allSetsWithVertices.size(); pos++)
            {
                if(allSetsWithVertices[pos].size() <=
deletePercentOfF*blueCounter) {
                    for(int i=0; i < allSetsWithVertices[pos].size();
i++) {
                        (*(allSetsWithVertices[pos][i])).C() =
Color4b(Color4b::White);
                        setFBV[(*(allSetsWithVertices[pos][i]))] = 'V';
                    }
```

```
                }
            }
            //************************ Color the different segments
1st Time and then delete F lines that have same color on both sides
            //** This clears up a lot of the extraneous connecting
points before we close gaps.
            if(showSegments1st){
                //***** New DisjointSet
                vcg::DisjointSet<CVertexO>* DisjointSetsToColor = new
vcg::DisjointSet<CVertexO>();
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //Every vertex not in set 'F' starts as disjoint
set
                    if(setFBV[vi] != 'F') {
                        //put the vertex in the set D
                        DisjointSetsToColor->MakeSet(&(*vi));
                    }
                }
                //***** Merge neighboring Sets
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is not a feature, then find 1-ring and
merge sets
                    if(setFBV[vi] != 'F') {
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            //if the neighbor of the current vertex is
not a feature then merge sets
                            if(setFBV[&(*tempVertexPointer)] != 'F') {
                                //if they are not already in the same
set, merge them
                                if(DisjointSetsToColor->FindSet(&(*vi))
!= DisjointSetsToColor->FindSet(tempVertexPointer)){
                                    DisjointSetsToColor-
>Union(&(*vi),tempVertexPointer);
                                }
                            }
                        }
                    }
                }
                //Find out how many distinct segment sets there are
*****************************
                int segmentSetCounter = 0;
                vector <CVertexO*> parentVertexofDisjointSets;
                //***** If a vertex is the parent of a DisjointSet, it
is a new Segment
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is not in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        if (DisjointSetsToColor->FindSet(&(*vi)) ==
&(*vi)){
```

```
                            //parent of a new set
parentVertexofDisjointSets.push_back(&(*vi));
                            segmentSetCounter++;
                        }
                    }
                }
                //******* find biggest set
                typedef  std::vector < std::vector <CVertexO*> >
crmatrix;
                crmatrix
vectorOfDisjointSetsWithVertices(segmentSetCounter,
std::vector<CVertexO*>(0));
                vector<CVertexO*>::iterator itVect;
                // ****************************  Create a vector of
vectors containing our sets / vertices
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        //find offset in parent vector--that is the
vertex set number
                        itVect =
find(parentVertexofDisjointSets.begin(),
parentVertexofDisjointSets.end(), DisjointSetsToColor-
>FindSet(&(*vi)));
                        if( itVect != parentVertexofDisjointSets.end()
) {
                            int offset =
std::distance(parentVertexofDisjointSets.begin(), itVect);
                            //store the set number in the vertex
                            FSetNum[vi] = offset;

vectorOfDisjointSetsWithVertices[offset].push_back(&(*vi));
                        }
                    }
                }
                //*********** For each set, sum the positive curvature
and store in new vector
                vector<float>
vectorOfSetPosCurvatureSums(vectorOfDisjointSetsWithVertices.size(),0);
                for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                {
                    float tempSumOfCurvature = 0.0;
                    for(int numInSet=0; numInSet <
vectorOfDisjointSetsWithVertices[pos].size(); numInSet++){
                        CVertexO* tempVertex =
vectorOfDisjointSetsWithVertices[pos][numInSet];
                        float tempCurvature = curvature[tempVertex];
                        //if it's positive, add it up
                        if(tempCurvature > 0.0) {
                            tempSumOfCurvature = tempSumOfCurvature +
tempCurvature;
                        }
                    }
```

```
                    vectorOfSetPosCurvatureSums[pos] =
tempSumOfCurvature;
                }
                //Find the set with the most vertices
                int maxInSet = 0;
                int maxOffset = 0;
                int secondBiggest = 0;
                int secondBiggestOffset = 0;
                for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                {
                    if(vectorOfDisjointSetsWithVertices[pos].size() >
maxInSet) {
                        secondBiggest = maxInSet;
                        secondBiggestOffset = maxOffset;
                        maxInSet =
vectorOfDisjointSetsWithVertices[pos].size();
                        maxOffset = pos;
                    }
                }
                int numBigSets = 0;
                int numLittleSets = 0;
                for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                {
                    if(pos != maxOffset) {
                        if(vectorOfDisjointSetsWithVertices[pos].size()
> (deletePercentOfBiggestSegment * secondBiggest)){
                            numBigSets++;
                        }
                        else{
                            numLittleSets++;
                        }
                    }
                }
                //color the different sets
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F') {
                        (*vi).C() = Color4b(Color4b::Blue);
                    }
                    //if it is not in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        if(FSetNum[vi] == maxOffset) {
                            (*vi).C() = Color4b(Color4b::White);
                        }
                        else {
                            //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                            int ScatterSize =
vectorOfDisjointSetsWithVertices.size();
                            Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                            (*vi).C()=BaseColor;
                        }
```

```
                    }
                }
                delete DisjointSetsToColor;
            }//show segments
            if(removeLines1st) {
                //Check if color same on both sides of F line
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F') {
                        (*vi).SetV();
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        (*vi).ClearV();
                        Color4b color1;
                        Color4b color2;
                        Color4b color3;
                        Color4b whiteColor = Color4b(Color4b::White);
                        Color4b blueColor = Color4b(Color4b::Blue);
                        Color4b tempColor;
                        int colorCounter = 0;
                        CVertexO* tempVertexPointer;
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            if (i == 0) {
                                tempVertexPointer = OneRing.allV.at(i);
                                color1 = (*tempVertexPointer).C();
                                colorCounter++;
                            }
                            if(i > 0 && i < OneRing.allV.size()) {
                                tempVertexPointer = OneRing.allV.at(i);
                                tempColor = (*tempVertexPointer).C();
                                if(colorCounter == 1) {
                                    if(tempColor != color1) {
                                        color2 = tempColor;
                                        colorCounter++;
                                    }
                                }
                                if(colorCounter == 2) {
                                    if(tempColor != color1 && tempColor
!= color2) {
                                        color3 = tempColor;
                                        colorCounter++;
                                    }
                                }
                            }
                        }
                        //delete it from feature
                        if(colorCounter == 2 || colorCounter == 1){
                            if(colorCounter == 2){
                                if(color1 != whiteColor && color2 !=
whiteColor) {
                                    //delete from feature if surrounded
by a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                    if(color2 != blueColor) {
```

```
                                        (*vi).C() = color2;
                                    }
                                }
                            }
                            if(colorCounter == 1) {
                                if(color1 != whiteColor) {
                                    //delete from feature if surrounded
by a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                }
                            }
                        }
                    }
                }//check if color on both sides of line same
            }//remove lines
            //******************************** Mark the borders after
deleting features
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is a feature, then find 1-ring and mark set 'B'
                if(setFBV[vi] == 'F') {
                    vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*vi));
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                        if(setFBV[&(*tempVertexPointer)] == 'V'){
                            //if the neighbor of the 'F' vertex is just
a vertex it is on the border
                            setFBV[&(*tempVertexPointer)] = 'B';
                        }
                    }
                }
            }
            // ******************************  Estimate distance from
feature region
            tri::Geo<CMeshO> g;
            //create a vector of starting vertices in set 'B' (border
of feature)
            std::vector<CVertexO*> fro;
            bool ret;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi)
                if( setFBV[vi] == 'B')
                    fro.push_back(&(*vi));
            if(!fro.empty()) {
                ret = g.DistanceFromFeature(m.cm, fro,
distanceConstraint);
            }
            float maxdist = distanceConstraint;
            float mindist = 0;
            //the distance is now stored in the quality, transfer the
quality to the distance
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                if((*vi).Q() < distanceConstraint) {
                    if( setFBV[vi] == 'F') {
```

```
                        disth[vi] = -((*vi).Q());
                    }
                    if( setFBV[vi] == 'B') {
                        disth[vi] = 0;
                    }
                    if( setFBV[vi] == 'V') {
                        disth[vi] = (*vi).Q();
                    }
                }
                else {
                    disth[vi] = std::numeric_limits<float>::max();
                }
            }
            //filter the distances
            if(filterOn){
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //new distance to vertex is average of neighborhood
distances
                    if( disth[vi] < maxdist) {
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        float numberOfNeigbors = 0.0;
                        float sumOfDistances = 0.0;
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            float tempDist =
disth[(*tempVertexPointer)];
                            if(tempDist !=
std::numeric_limits<float>::max()) {
                                sumOfDistances = sumOfDistances +
tempDist;
                                numberOfNeigbors++;
                            }
                        }
                        float inverseNumNeighbors = 1.0 /
numberOfNeigbors;
                        disth[vi] = inverseNumNeighbors *
sumOfDistances;
                    }
                }
            }
            /****************************** Connecting points by angle
*************************/
            vector <CVertexO*> vectCPoints;
            int CPointCounter = 0;
            int vertexCounter = 0;
            if(findConnectingPoints) {
                //Find Connecting Points
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if( setFBV[vi] == 'F' ) {
                        (*vi).SetV();
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        (*vi).ClearV();
```

```
                        char currentSet = 'a';
                        char initialSet = 'a';
                        int numberOfChanges = 0;
                        float numberOfNeighbors = 0.0;
                        float sumOfDivergence = 0.0;
                        float angle = 0.0;
                        bool addAngle = false;
                        CVertexO* firstVertex;
                        CVertexO* previousVertex;
                        char previousSet = 'a';
                        char firstSet = 'a';
                        int borderNeighborCounter = 0;
                        int featureNeighborCounter = 0;
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            char tempSetOfThisNeighbor =
(setFBV[&(*tempVertexPointer)]);
                            if(tempSetOfThisNeighbor == 'F') {
                                featureNeighborCounter++;
                                //don't add angle, we are still in 'F'
                                addAngle = false;
                            }
                            if(tempSetOfThisNeighbor == 'B'){
                                borderNeighborCounter++;
                                //add angle, we are still in 'F'
                                addAngle = true;
                            }
                            if(i == 0){
                                //initialize the starting set
                                currentSet =
setFBV[&(*tempVertexPointer)];
                                initialSet =
setFBV[&(*tempVertexPointer)];
                                firstVertex = tempVertexPointer;
                                firstSet =
setFBV[&(*tempVertexPointer)];
                                previousVertex = tempVertexPointer;
                                previousSet =
setFBV[&(*tempVertexPointer)];
                            }
                            if(tempSetOfThisNeighbor != currentSet){
                                //changed sets, update the
numberOfChanges
                                currentSet =
setFBV[&(*tempVertexPointer)];
                                numberOfChanges++;
                            }
                            //check if last point is in same set as
first point
                            if(i == (OneRing.allV.size() - 1)) {
                                if(tempSetOfThisNeighbor !=
initialSet){
                                    numberOfChanges++;
                                }
                                if(firstSet == 'F' && currentSet ==
'F'){
```

```
                                    //don't add the angle
                                }
                                else{
                                    //calculate and add the angle
between tempVertexPointer and firstVertex
                                    vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                    //vector from gradient field of w
                                    vcg::Point3f v2 =
(*firstVertex).P() - (*vi).P();
                                    double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                    double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                    v1 = v1 * (1.0/v1_magnitude);
                                    v2 = v2 * (1.0/v2_magnitude);
                                    double cosTheta = 0.0;
                                    if(v1_magnitude*v2_magnitude == 0)
{
                                        cosTheta = 0.0;
                                    }
                                    else{
                                        cosTheta = v1.X()*v2.X() +
v1.Y()*v2.Y() + v1.Z()*v2.Z();
                                    }
                                    float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                    angle = angle + tempAngleinDegrees;
                                }
                            }
                            if(i > 0){
                                if(previousSet == 'F' && currentSet ==
'F'){
                                    //don't add the angle
                                }
                                else{
                                    //calculate and add the angle
between previousVertex and tempVertexPointer
                                    vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                    //vector from gradient field of w
                                    vcg::Point3f v2 =
(*previousVertex).P() - (*vi).P();
                                    double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                    double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                    v1 = v1 * (1.0/v1_magnitude);
                                    v2 = v2 * (1.0/v2_magnitude);
                                    double cosTheta = 0.0;
                                    if(v1_magnitude*v2_magnitude == 0)
{
                                        cosTheta = 0.0;
                                    }
                                    else{
                                        cosTheta = v1.X()*v2.X() +
v1.Y()*v2.Y() + v1.Z()*v2.Z();
```

```
                                    }
                                    float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                    angle = angle + tempAngleinDegrees;
                                }
                                //update the pointers
                                previousVertex = tempVertexPointer;
                                previousSet =
setFBV[&(*tempVertexPointer)];
                            }
                            float tempDivergence =
divfval[&(*tempVertexPointer)];
                            sumOfDivergence = sumOfDivergence +
tempDivergence;
                            numberOfNeighbors++;
                        }
                        //Candidate connecting point
                        if(numberOfChanges <= 2){
                            //not element of Fp, may be connecting
point
                            if(angle > connectingPointAngle) {
                                (*vi).C() = Color4b(Color4b::Yellow);
                                CpointOwner[vi] = CPointCounter;
                                source[vi] = &(*vi);
                                vectCPoints.push_back(&(*vi));
                                CPointCounter++;
                            }
                        }
                    }//in set 'F'
                    vertexCounter++;
                }//go through vertices
            }//find connecting points and color them yellow
            //Find Mid Points
            std::vector<CMeshO::VertexPointer>  midPointVector;
            if(findMidPoints){
                //************************************  Find
connecting paths from connecting points
                //setup the seed vector of connecting points
                std::vector<VertDist> seedVec;
                std::vector<CVertexO*>::iterator fi;
                for( fi  = vectCPoints.begin(); fi != vectCPoints.end()
; ++fi)
                {
                    seedVec.push_back(VertDist(*fi,0.0));
                }
                std::vector<VertDist> frontier;
                CMeshO::VertexPointer curr;
                CMeshO::ScalarType unreached  =
std::numeric_limits<CMeshO::ScalarType>::max();
                CMeshO::VertexPointer pw;
                TempDataType TD(m.cm.vert, unreached);
                std::vector <VertDist >::iterator ifr;
                for(ifr = seedVec.begin(); ifr != seedVec.end();
++ifr){
                    TD[(*ifr).v].d = 0.0;
                    (*ifr).d = 0.0;
                    TD[(*ifr).v].source  = (*ifr).v;
```

```
                    frontier.push_back(VertDist((*ifr).v,0.0));
                }
                // initialize Heap
                make_heap(frontier.begin(),frontier.end(),pred());
                std::vector<int> doneConnectingPoints;
                CMeshO::ScalarType curr_d,d_curr = 0.0,d_heap;
                CMeshO::VertexPointer curr_s = NULL;
                CMeshO::PerVertexAttributeHandle
<CMeshO::VertexPointer> * vertSource = NULL;
                CMeshO::ScalarType max_distance=0.0;
                std::vector<VertDist >:: iterator iv;
                int connectionCounter = 0;
                while(!frontier.empty() && max_distance < maxdist)
                {
                    pop_heap(frontier.begin(),frontier.end(),pred());
                    curr = (frontier.back()).v;
                    int tempCpointOwner = CpointOwner[curr];
                    int tempFSetNum = FSetNum[curr];
                    curr_s = TD[curr].source;
                    if(vertSource!=NULL)
                        (*vertSource)[curr] = curr_s;
                    d_heap = (frontier.back()).d;
                    frontier.pop_back();
                    assert(TD[curr].d <= d_heap);
                    assert(curr_s != NULL);
                    if(TD[curr].d < d_heap )// a vertex whose distance
has been improved after it was inserted in the queue
                        continue;
                    assert(TD[curr].d == d_heap);
                    d_curr =  TD[curr].d;
                    if(d_curr > max_distance) {
                        max_distance = d_curr;
                    }
                    //check the vertices around the current point
                    (*curr).SetV();
                    vcg::tri::Nring<CMeshO> OneRing(&(*curr), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*curr));
                    (*curr).ClearV();
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        pw = OneRing.allV.at(i);
                        //just find the shortest distance between
points
                        curr_d = d_curr + vcg::Distance(pw->cP(),curr-
>cP());
                        //check if we are still searching from this
connecting point
                        std::vector<int>::iterator it;
                        it = std::find(doneConnectingPoints.begin(),
doneConnectingPoints.end(), CpointOwner[curr]);
                        //we are still searching from this connecting
point
                        if (it == doneConnectingPoints.end()){
                            //This point has been explored by a
different Cpoint before
                            //deleted the bit about not connecting to
your own set: && FSetNum[pw] != FSetNum[curr]
```

```
                            if(CpointOwner[pw] != CpointOwner[curr] &&
CpointOwner[pw] != -1 && setFBV[&(*pw)] != 'F'){
                                //This point has been explored before,
we may have a connection
                                //(check if the CpointOwner is active
or already connected up with someone else
                                //std::vector<int>::iterator chk;
                                //chk =
std::find(doneConnectingPoints.begin(), doneConnectingPoints.end(),
CpointOwner[pw]);
                                //we are still searching from the
CpointSource of the point we found
                                //if(chk == doneConnectingPoints.end())
{
                                (*pw).C() = Color4b(Color4b::Magenta);
                                connectionCounter++;
                                source1[pw] = curr;
                                //store the midpoint so we can make the
connecting path later
                                midPointVector.push_back(pw);
                                //add CpointOwner to vector of
doneConnectingPoints

doneConnectingPoints.push_back(CpointOwner[curr]);

doneConnectingPoints.push_back(CpointOwner[pw]);
                                // }
                                /* else{
                            }*/
                            }
                            //deleted  && curvature[pw] <
maxCurvatureInPath
                            else if(TD[(pw)].d > curr_d && curr_d <
maxdist && setFBV[&(*pw)] != 'F'){
                                //This point has not been explored
before, keep looking
                                //update source, Fsetnum, CpointOwner
                                CpointOwner[pw] = tempCpointOwner;
                                source[pw] = curr;
                                FSetNum[pw] = tempFSetNum;
                                TD[(pw)].d = curr_d;
                                TD[pw].source = curr;

frontier.push_back(VertDist(pw,curr_d));

push_heap(frontier.begin(),frontier.end(),pred());
                                }
                        }
                        else {
                            //not searching from this connecting point
anymore
                        }
                    }
                }// end while
            }//find mid points and color magenta
            if(findMidPoints && findPathstoConnect) {
                //********************** Make the connecting path
```

```
                vector<CMeshO::VertexPointer>::iterator iter;
                for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                    //track back source to beginning
                    CMeshO::VertexPointer tempVertexPointer = *iter;
                    CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                    while(tempVertexPointer != tempVertexSource){
                        (*tempVertexPointer).C() =
Color4b(Color4b::Green);
                        // setFBV[tempVertexPointer] = 'F';
                        tempVertexPointer = tempVertexSource;
                        tempVertexSource = source[tempVertexPointer];
                    }
                    //track back source1 to beginning
                    tempVertexPointer = *iter;
                    tempVertexSource = source1[tempVertexPointer];
                    while(tempVertexPointer != tempVertexSource){
                        (*tempVertexPointer).C() =
Color4b(Color4b::Cyan);
                        //setFBV[tempVertexPointer] = 'F';
                        tempVertexPointer = tempVertexSource;
                        tempVertexSource = source[tempVertexPointer];
                    }
                }//make connecting path
            }//find paths to connect and color green and cyan
            if(addPathstoFeatureRegion){
                //********************** Make the connecting path
                vector<CMeshO::VertexPointer>::iterator iter;
                for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                    //track back source to beginning
                    CMeshO::VertexPointer tempVertexPointer = *iter;
                    CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                    while(tempVertexPointer != tempVertexSource){
                        (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                        setFBV[tempVertexPointer] = 'F';
                        newF[tempVertexPointer] = 1;
                        tempVertexPointer = tempVertexSource;
                        tempVertexSource = source[tempVertexPointer];
                    }
                    //track back source1 to beginning
                    tempVertexPointer = *iter;
                    tempVertexSource = source1[tempVertexPointer];
                    while(tempVertexPointer != tempVertexSource){
                        (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                        setFBV[tempVertexPointer] = 'F';
                        newF[tempVertexPointer] = 1;
                        tempVertexPointer = tempVertexSource;
                        tempVertexSource = source[tempVertexPointer];
                    }
                }
            }//add paths to feature region
```

```
            //******************************** get ready to
skeletonize and prune by changing sets to either F or V
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is not a feature
                if(setFBV[vi] != 'F') {
                    setFBV[vi] = 'V';
                }
            }
            //*************************** Skeletonize and prune
            //Apply Skeletonize operator
            if(skeletonizeAfter){
                bool changes = false;
                //if still changes loop through and skeletonize
                do {
                    changes = false;
                    //go through the vertices and if 'F' check for
centers and discs
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            //check for centers and discs
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            bool neighborsAllF = true;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature, vi not a center
                                if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                    neighborsAllF = false;
                                }
                            }
                            if(neighborsAllF){
                                //mark as a center
                                center[vi] = 1;
                                //mark all neighbors as disc
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    disc[&(*tempVertexPointer)] = 1;
                                }
                            }
                        }//in set F
                    }//go through vertices
                    //go through vertices again, if not a center and is
a disc, check complexity and delete if not complex
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F' && center[vi] == 0 &&
disc[vi] == 1){
```

```
                            //check complexity
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            int numberOfChanges = 0;
                            char iVertex;
                            char iPlus1Vertex;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if(i < (OneRing.allV.size() - 1)) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(i+1);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                else {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(0);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                //change noted
                                if(iVertex != iPlus1Vertex){
                                    numberOfChanges++;
                                }
                            }
                            //not complex, delete it from feature
                            if(numberOfChanges < 4){
                                //delete from feature
                                setFBV[vi] = 'V';
                                center[vi] = 0;
                                disc[vi] = 0;
                                changes = true;
                            }
                        }//not a center, yes a disc
                    }//go through vertices
                    //******************************** reset disc and
center data for next iteration
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is a feature, reset center and disc
data
                        if(setFBV[vi] == 'F') {
                            center[vi] = 0;
                            disc[vi] = 0;
                        }
```

```
                    }
                } while (changes == true);//loop if changes
            }//skeletonize operator
            //prune
            if(pruneAfter){
                //pune a certain number of times
                for(int pruneIter = 0; pruneIter < pruneAfterLength;
pruneIter++){
                    int pruned = 0;
                    vertexCounter = 0;
                    //go through the vertices and if 'F' check
complexity
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F'){
                            //check complexity
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            int numberOfChanges = 0;
                            char iVertex;
                            char iPlus1Vertex;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if(i < (OneRing.allV.size() - 1)) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(i+1);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                else {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    iVertex =
(setFBV[&(*tempVertexPointer)]);
                                    tempVertexPointer =
OneRing.allV.at(0);
                                    iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                }
                                //change noted
                                if(iVertex != iPlus1Vertex){
                                    numberOfChanges++;
                                }
                            }
                            //not complex, delete it from feature
                            if(numberOfChanges < 4){
                                //prune from feature
                                setFBV[vi] = 'V';
                                pruned++;
```

```
                        }
                    }//set
                    vertexCounter++;
                }//go through vertices
            }//number of times we prune
        }//prune after
        /*
        if(showFinalFeature) {
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                if(setFBV[vi] == 'F') {
                  (*vi).C() = Color4b(Color4b::Blue);
                }
                if(setFBV[vi] != 'F') {
                  (*vi).C() = Color4b(Color4b::White);
                }
            }
        }
        */
        if(showSegments){
            //************************* Color the different
segments 2nd Time
            //***** New DisjointSet
            vcg::DisjointSet<CVertexO>* ptrDsetSegment = new
vcg::DisjointSet<CVertexO>();
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                //Every vertex not in set 'F' starts as disjoint
set
                if(setFBV[vi] != 'F') {
                    //put the vertex in the set D
                    ptrDsetSegment->MakeSet(&(*vi));
                }
            }
            //***** Merge neighboring Sets
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                //if it is not a feature, then find 1-ring and
merge sets
                if(setFBV[vi] != 'F') {
                    vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*vi));
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                        //if the neighbor of the current vertex is
not a feature then merge sets
                        if(setFBV[&(*tempVertexPointer)] != 'F') {
                            //if they are not already in the same
set, merge them
                            if(ptrDsetSegment->FindSet(&(*vi)) !=
ptrDsetSegment->FindSet(tempVertexPointer)){
                                ptrDsetSegment-
>Union(&(*vi),tempVertexPointer);
                            }
                        }
                    }
```

```
                    }
                }
                //Find out how many distinct segment sets there are
*****************************
                int segmentSetCounter = 0;
                vector <CVertexO*> parentVertexofSegmentSet;
                //***** If a vertex is the parent of a DisjointSet, it
is a new Segment
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is not in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        if (ptrDsetSegment->FindSet(&(*vi)) == &(*vi)){
                            //parent of a new set
                            parentVertexofSegmentSet.push_back(&(*vi));
                            segmentSetCounter++;
                        }
                    }
                }
                //******* find biggest set
                typedef  std::vector < std::vector <CVertexO*> >
crmatrix;
                crmatrix vectorOfSetsWithVertices(segmentSetCounter,
std::vector<CVertexO*>(0));
                // ****************************  Create a vector of
vectors containing our sets / vertices
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        //find offset in parent vector--that is the
vertex set number
                        itVect = find(parentVertexofSegmentSet.begin(),
parentVertexofSegmentSet.end(), ptrDsetSegment->FindSet(&(*vi)));
                        if( itVect != parentVertexofSegmentSet.end() )
{
                            int offset =
std::distance(parentVertexofSegmentSet.begin(), itVect);
                            //store the set number in the vertex
                            FSetNum[vi] = offset;

vectorOfSetsWithVertices[offset].push_back(&(*vi));
                        }
                    }
                }
                //Find the set with the most vertices
                int maxInSet = 0;
                int maxOffset = 0;
                int secondBiggest = 0;
                int secondBiggestOffset = 0;
                for(int pos=0; pos < vectorOfSetsWithVertices.size();
pos++)
                {
                    if(vectorOfSetsWithVertices[pos].size() > maxInSet)
{
```

```
                        secondBiggest = maxInSet;
                        secondBiggestOffset = maxOffset;
                        maxInSet =
vectorOfSetsWithVertices[pos].size();
                        maxOffset = pos;
                    }
                }
                int numBigSets = 0;
                int numLittleSets = 0;
                for(int pos=0; pos < vectorOfSetsWithVertices.size();
pos++)
                {
                    if(pos != maxOffset) {
                        if(vectorOfSetsWithVertices[pos].size() >
(deletePercentOfBiggestSegment * secondBiggest)){
                            numBigSets++;
                        }
                        else{
                            numLittleSets++;
                        }
                    }
                }
                totalNumBigSets = numBigSets;
                totalNumLittleSets = numLittleSets;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F') {
                        (*vi).C() = Color4b(Color4b::Blue);
                    }
                    //if it is not in set 'F', then find what set it
belongs to
                    if(setFBV[vi] != 'F') {
                        if(FSetNum[vi] == maxOffset) {
                            (*vi).C() = Color4b(Color4b::White);
                        }
                        else {
                            //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                            int ScatterSize =
vectorOfSetsWithVertices.size();
                            Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                            (*vi).C()=BaseColor;
                        }
                    }
                }
                delete ptrDsetSegment;
            }//show segments
            if(removeLines) {
                //Check if color same on both sides of F line
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F') {
                        (*vi).SetV();
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
```

```
                        (*vi).ClearV();
                        Color4b color1;
                        Color4b color2;
                        Color4b color3;
                        Color4b blueColor = Color4b(Color4b::Blue);
                        Color4b tempColor;
                        int colorCounter = 0;
                        CVertexO* tempVertexPointer;
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            if (i == 0) {
                                tempVertexPointer = OneRing.allV.at(i);
                                color1 = (*tempVertexPointer).C();
                                colorCounter++;
                            }
                            if(i > 0 && i < OneRing.allV.size()) {
                                tempVertexPointer = OneRing.allV.at(i);
                                tempColor = (*tempVertexPointer).C();
                                if(colorCounter == 1) {
                                    if(tempColor != color1) {
                                        color2 = tempColor;
                                        colorCounter++;
                                    }
                                }
                                if(colorCounter == 2) {
                                    if(tempColor != color1 && tempColor
!= color2) {
                                        color3 = tempColor;
                                        colorCounter++;
                                    }
                                }
                            }
                        }
                        //delete it from feature
                        if(colorCounter == 2 || colorCounter == 1){
                            if(colorCounter == 2){
                                //delete from feature if surrounded by
a color besides white
                                setFBV[vi] = 'V';
                                if(color1 != blueColor) {
                                    (*vi).C() = color1;
                                }
                                if(color2 != blueColor) {
                                    (*vi).C() = color2;
                                }
                            }
                            if(colorCounter == 1) {
                                //delete from feature if surrounded by
a color besides white
                                setFBV[vi] = 'V';
                                if(color1 != blueColor) {
                                    (*vi).C() = color1;
                                }
                            }
                        }
                    }
                }//check if color on both sides of line same
            }//remove lines
```

```
        }//if foundIt == true
        else {//didn't find it with connections
            //reset binary search variables
            first = -30;
            last = 0;
            foundIt = false;
            //no connections this time
            bool findConnectingPoints = false;
            bool findMidPoints = false;
            bool findPathstoConnect = false;
            bool shortestEuclidianDistToConnect = false;
            bool addPathstoFeatureRegion = false;
            while (first <= last && foundIt == false) {
                int intEpsilon = (first + last) / 2;  // compute
integer representation of epsilon point.
                epsilon = intEpsilon / 100.0f;
                /* ********************  Calculate curvature:
curvature.h computes the discrete gaussian curvature.
         <vcg/complex/algorithms/update/curvature.h>
         For further details, please, refer to:
         Discrete Differential-Geometry Operators for Triangulated 2-
Manifolds Mark Meyer,
         Mathieu Desbrun, Peter Schroder, Alan H. Barr VisMath '02,
Berlin </em>*/
                // results stored in (*vi).Kh() (mean) and (*vi).Kg()
(guassian)
                tri::UpdateCurvature<CMeshO>::MeanAndGaussian(m.cm);
                // ********************  Put the curvature in the
quality: <vcg/complex/algorithms/update/quality.h>

tri::UpdateQuality<CMeshO>::VertexFromMeanCurvature(m.cm);
                //******************* ComputePerVertexQualityMinMax 15
and 85 percent
                std::pair<float,float> minmax =
std::make_pair(std::numeric_limits<float>::max(),-
std::numeric_limits<float>::max());
                CMeshO::VertexIterator vi;
                std::vector<float> QV;
                QV.reserve(m.cm.vn);
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                    if(!(*vi).IsD()) QV.push_back((*vi).Q());
                //bottom 15% maps to -1

std::nth_element(QV.begin(),QV.begin()+15*(m.cm.vn/100),QV.end());
                float newmin=*(QV.begin()+m.cm.vn/100);
                //top 15% maps to 1
                std::nth_element(QV.begin(),QV.begin()+m.cm.vn-
15*(m.cm.vn/100),QV.end());
                float newmax=*(QV.begin()+m.cm.vn-m.cm.vn/100);
                minmax.first = newmin;
                minmax.second = newmax;
                // ***************************
MapCurvatureRangetoOneNegOne
                float B = 8.0;
                float min = minmax.first;
                float max = minmax.second;
```

```
                float upperBound = 0.0;
                if(min < 0)
                    min = min * -1;
                if(min > max)
                    upperBound = min;
                else
                    upperBound = max;
                float K = 0.5 * log10((1 + ((pow(2.0,B)-2)/(pow(2.0,B)-
1))) / ((1 - ((pow(2.0,B)-2)/(pow(2.0,B)-1)))));
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                    if(!(*vi).IsD())
                    {
                        float remapped = tanh((*vi).Q() * (K /
upperBound));
                        (*vi).Q() = remapped;
                        //Log( "Value to be remapped: %f   remapped:
%f\n", (*vi).Q(), remapped );
                        // Log( "inside mapping");
                    }
                vcg::DisjointSet<CVertexO>* ptrDset = new
vcg::DisjointSet<CVertexO>();
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //all vertices start as set V for Vertex and
unmarked
                    setFBV[vi] = 'V';
                    marked[vi] = 0;
                    disc[vi] = 0;
                    center[vi] = 0;
                    complex[vi] = 0;
                    newF[vi] = 0;
                    //all vertices start as -1 FSetNum
                    FSetNum[vi] = -1;
                    //none have been visited to connect C points, so
set CpointOwner to -1
                    CpointOwner[vi] = -1;
                    //color it White initially
                    (*vi).C() = Color4b(Color4b::White);
                    curvature[vi] = (*vi).Q();
                    //feature region, less than defined min curvature
                    if(curvature[vi] < epsilon) {
                        //in set F for feature
                        setFBV[vi] = 'F';
                    }
                }
                // apply morphological opperators dilate and then erode
****************************
                if(morphologicalOperations){
                    //dilate 1-ring neighbor
                    for(int k = 0; k < 1; k++){
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] != 'V') {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
```

```
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, mark the neighbor to be one
                                    //if it is negative curvature
                                    if(setFBV[&(*tempVertexPointer)] ==
'V') {
                                        //mark it to be 'N'
                                        marked[&(*tempVertexPointer)] =
1;
                                    }
                                }
                            }
                        }
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it was marked, change it
                            if(marked[vi] == 1){
                                setFBV[vi] = 'F';
                                marked[vi] = 0;
                            }
                        }
                    }//grow once
                    //erode 1-ring neighbor
                    for(int k = 0; k < 1; k++){
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] != 'V') {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                bool neighborsAllF = true;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, we will erode this vertex
                                    if(setFBV[&(*tempVertexPointer)] ==
'V') {
                                        neighborsAllF = false;
                                    }
                                }
                                if(neighborsAllF){
                                    //mark to not erode this vertex
this round
                                    marked[vi] = 1;
                                }
                            }
                        }
```

```
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it was marked to not erode, keep it
                            if(marked[vi] == 1){
                                //setFBV[vi] = 'F';
                                marked[vi] = 0;
                            }
                            else{
                                setFBV[vi] = 'V';
                            }
                        }
                    }//erode once
                }
                //Apply Skeletonize operator
                if(skeletonize){
                    int debugCounter = 1;
                    bool changes = false;
                    //if still changes loop through and skeletonize
                    do {
                        changes = false;
                        //go through the vertices and if 'F' check for
centers and discs
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                //check for centers and discs
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                bool neighborsAllF = true;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, vi not a center
                                    if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                        neighborsAllF = false;
                                    }
                                }
                                if(neighborsAllF){
                                    //mark as a center
                                    center[vi] = 1;
                                    //mark all neighbors as disc
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                        disc[&(*tempVertexPointer)] =
1;
                                    }
                                }
                            }//in set F
                        }//go through vertices
```

```
                        //go through vertices again, if not a center
and is a disc, check complexity and delete if not complex
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(center[vi] == 0 && disc[vi] == 1){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //delete from feature
                                    setFBV[vi] = 'V';
                                    center[vi] = 0;
                                    disc[vi] = 0;
                                    changes = true;
                                }
                            }//not a center, yes a disc
                        }//go through vertices
                        //******************************** reset disc
and center data for next iteration
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
```

```
                                center[vi] = 0;
                                disc[vi] = 0;
                            }
                        }
                        /* if(debugCounter == 1){
                      changes = false;
                   }*/
                    } while (changes == true);//loop if changes
                }//skeletonize operator
                //prune
                if(prune){
                    //prune a certain number of times
                    for(int pruneIter = 0; pruneIter < pruneLength;
pruneIter++){
                        int pruned = 0;
                        //go through the vertices and if 'F' check
complexity
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F'){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
```

```
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //prune from feature
                                    setFBV[vi] = 'V';
                                    pruned++;
                                }
                            }//checking current vertex
                        }//go through vertices
                    }//number of times we prune
                }//prune before
                //****************************** Initialize the
Disjoint sets of 'F' and delete any that are small
*****************************
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F'){
                        //put the vertex in the set D
                        ptrDset->MakeSet(&(*vi));
                    }
                }
                /*
            if(showMinAndMaxFeature || onlyShowFeature){
                //go through the vertices and color them according to
sets
                for(vi=m.cm.vert.begin();vi!=m.cm.vert.end();++vi) {
                    if(setFBV[&(*vi)] == 'F')
                        (*vi).C() = Color4b(Color4b::Blue);
                    if(setFBV[&(*vi)] == 'N')
                        (*vi).C() = Color4b(Color4b::Green);
                      //more than defined maximum curvature
                   // if(!onlyShowFeature) {
                     //   if(curvature[vi] > eta)
                       //    (*vi).C() = Color4b(Color4b::Red);
                   // }
                }
            }
        */
                //print number of sets
                //ptrDset->printNumberOfSets();
                //Go through vertices again--if in set 'F',
*****************************
                //then find 1-Ring of vertices and merge sets if
orginal vertex is 'F'
                //and new 1-ring vertex is 'F', define set 'B'
                int blueCounter = 0;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is a feature, then find 1-ring and merge
sets
                    if(setFBV[vi] == 'F') {
                        blueCounter++;
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
```

```
                            //if the neighbor of the current vertex is
a feature then merge sets
                            if(setFBV[&(*tempVertexPointer)] == 'F') {
                                //if they are not already in the same
set, merge them
                                if(ptrDset->FindSet(&(*vi)) != ptrDset-
>FindSet(tempVertexPointer)){
                                    ptrDset-
>Union(&(*vi),tempVertexPointer);
                                }
                            }
                            else if(setFBV[&(*tempVertexPointer)] ==
'V'){
                                //if the neighbor of the blue vertex is
just a vertex it is on the border
                                //we do this after we delete the small
sets of 'F'
                                //setFBV[&(*tempVertexPointer)] = 'B';
                            }
                        }
                    }
                }
                //Find out how many distinct sets there are
*******************************
                int setCounter = 0;
                vector <CVertexO*> parentVertexofSet;
                typedef  std::vector < std::vector <CVertexO*> >
matrix;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is in set 'F', then find what set it
belongs to
                    if(setFBV[vi] == 'F') {
                        if (ptrDset->FindSet(&(*vi)) == &(*vi)){
                            //parent of a new set
                            parentVertexofSet.push_back(&(*vi));
                            setCounter++;
                        }
                    }
                }
                matrix allSetsWithVertices(setCounter,
std::vector<CVertexO*>(0));
                // *****************************  Create a vector of
vectors containing our sets / vertices
                vector<CVertexO*>::iterator itVect;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is in set 'F', then find what set it
belongs to
                    if(setFBV[vi] == 'F') {
                        //find offset in parent vector--that is the
vertex set number
                        itVect = find(parentVertexofSet.begin(),
parentVertexofSet.end(), ptrDset->FindSet(&(*vi)));
                        if( itVect != parentVertexofSet.end() ) {
                            int offset =
std::distance(parentVertexofSet.begin(), itVect);
```

```
                            //store the set number in the vertex
                            FSetNum[vi] = offset;

allSetsWithVertices[offset].push_back(&(*vi));
                        }
                    }
                }
                //delete the disjoint set
                delete ptrDset;
                //delete sets of D if the number of vertices is less
than .01 D
                for(int pos=0; pos < allSetsWithVertices.size(); pos++)
                {
                    if(allSetsWithVertices[pos].size() <=
deletePercentOfF*blueCounter) {
                        for(int i=0; i <
allSetsWithVertices[pos].size(); i++) {
                            (*(allSetsWithVertices[pos][i])).C() =
Color4b(Color4b::White);
                            setFBV[(*(allSetsWithVertices[pos][i]))] =
'V';
                        }
                    }
                }
                //************************ Color the different
segments 1st Time and then delete F lines that have same color on both
sides
                //** This clears up a lot of the extraneous connecting
points before we close gaps.
                if(showSegments1st){
                    //***** New DisjointSet
                    vcg::DisjointSet<CVertexO>* DisjointSetsToColor =
new vcg::DisjointSet<CVertexO>();
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //Every vertex not in set 'F' starts as
disjoint set
                        if(setFBV[vi] != 'F') {
                            //put the vertex in the set D
                            DisjointSetsToColor->MakeSet(&(*vi));
                        }
                    }
                    //***** Merge neighboring Sets
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not a feature, then find 1-ring and
merge sets
                        if(setFBV[vi] != 'F') {
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature then merge sets
```

```
                                if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                    //if they are not already in the
same set, merge them
                                    if(DisjointSetsToColor-
>FindSet(&(*vi)) != DisjointSetsToColor->FindSet(tempVertexPointer)){
                                        DisjointSetsToColor-
>Union(&(*vi),tempVertexPointer);
                                    }
                                }
                            }
                        }
                    }
                    //Find out how many distinct segment sets there are
******************************
                    int segmentSetCounter = 0;
                    vector <CVertexO*> parentVertexofDisjointSets;
                    //***** If a vertex is the parent of a DisjointSet,
it is a new Segment
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if (DisjointSetsToColor->FindSet(&(*vi)) ==
&(*vi)){
                                //parent of a new set

parentVertexofDisjointSets.push_back(&(*vi));
                                segmentSetCounter++;
                            }
                        }
                    }
                    //******* find biggest set
                    typedef  std::vector < std::vector <CVertexO*> >
crmatrix;
                    crmatrix
vectorOfDisjointSetsWithVertices(segmentSetCounter,
std::vector<CVertexO*>(0));
                    vector<CVertexO*>::iterator itVect;
                    // ******************************  Create a vector
of vectors containing our sets / vertices
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is in set 'F', then find what set it
belongs to
                        if(setFBV[vi] != 'F') {
                            //find offset in parent vector--that is the
vertex set number
                            itVect =
find(parentVertexofDisjointSets.begin(),
parentVertexofDisjointSets.end(), DisjointSetsToColor-
>FindSet(&(*vi)));
                            if( itVect !=
parentVertexofDisjointSets.end() ) {
                                int offset =
std::distance(parentVertexofDisjointSets.begin(), itVect);
```

```
                                //store the set number in the vertex
                                FSetNum[vi] = offset;

vectorOfDisjointSetsWithVertices[offset].push_back(&(*vi));
                            }
                        }
                    }
                    //*********** For each set, sum the positive
curvature and store in new vector
                    vector<float>
vectorOfSetPosCurvatureSums(vectorOfDisjointSetsWithVertices.size(),0);
                    for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                    {
                        float tempSumOfCurvature = 0.0;
                        for(int numInSet=0; numInSet <
vectorOfDisjointSetsWithVertices[pos].size(); numInSet++){
                            CVertexO* tempVertex =
vectorOfDisjointSetsWithVertices[pos][numInSet];
                            float tempCurvature =
curvature[tempVertex];
                            //if it's positive, add it up
                            if(tempCurvature > 0.0) {
                                tempSumOfCurvature = tempSumOfCurvature
+ tempCurvature;
                            }
                        }
                        vectorOfSetPosCurvatureSums[pos] =
tempSumOfCurvature;
                    }
                    //Find the set with the most vertices
                    int maxInSet = 0;
                    int maxOffset = 0;
                    int secondBiggest = 0;
                    int secondBiggestOffset = 0;
                    for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                    {
                        if(vectorOfDisjointSetsWithVertices[pos].size()
> maxInSet) {
                            secondBiggest = maxInSet;
                            secondBiggestOffset = maxOffset;
                            maxInSet =
vectorOfDisjointSetsWithVertices[pos].size();
                            maxOffset = pos;
                        }
                    }
                    int numBigSets = 0;
                    int numLittleSets = 0;
                    for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                    {
                        if(pos != maxOffset) {

if(vectorOfDisjointSetsWithVertices[pos].size() >
(deletePercentOfBiggestSegment * secondBiggest)){
                                numBigSets++;
```

```
                            }
                            else{
                                numLittleSets++;
                            }
                        }
                    }
                    //color the different sets
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).C() = Color4b(Color4b::Blue);
                        }
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if(FSetNum[vi] == maxOffset) {
                                (*vi).C() = Color4b(Color4b::White);
                            }
                            else {
                                //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                                int ScatterSize =
vectorOfDisjointSetsWithVertices.size();
                                Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                                (*vi).C()=BaseColor;
                            }
                        }
                    }
                    delete DisjointSetsToColor;
                }//show segments
                if(removeLines1st) {
                    //Check if color same on both sides of F line
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            Color4b color1;
                            Color4b color2;
                            Color4b color3;
                            Color4b whiteColor =
Color4b(Color4b::White);
                            Color4b blueColor = Color4b(Color4b::Blue);
                            Color4b tempColor;
                            int colorCounter = 0;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if (i == 0) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    color1 = (*tempVertexPointer).C();
```

```
                                    colorCounter++;
                                }
                                if(i > 0 && i < OneRing.allV.size()) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    tempColor =
(*tempVertexPointer).C();
                                    if(colorCounter == 1) {
                                        if(tempColor != color1) {
                                            color2 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                    if(colorCounter == 2) {
                                        if(tempColor != color1 &&
tempColor != color2) {
                                            color3 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                }
                            }
                            //delete it from feature
                            if(colorCounter == 2 || colorCounter == 1){
                                if(colorCounter == 2){
                                    if(color1 != whiteColor && color2
!= whiteColor) {
                                        //delete from feature if
surrounded by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                        if(color2 != blueColor) {
                                            (*vi).C() = color2;
                                        }
                                    }
                                }
                                if(colorCounter == 1) {
                                    if(color1 != whiteColor) {
                                        //delete from feature if
surrounded by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                    }
                                }
                            }
                        }
                    }//check if color on both sides of line same
                }//remove lines
                //******************************** Mark the borders
after deleting features
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
```

```
                    //if it is a feature, then find 1-ring and mark set
'B'
                    if(setFBV[vi] == 'F') {
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            if(setFBV[&(*tempVertexPointer)] == 'V'){
                                //if the neighbor of the 'F' vertex is
just a vertex it is on the border
                                setFBV[&(*tempVertexPointer)] = 'B';
                            }
                        }
                    }
                }
                // ******************************  Estimate distance
from feature region
                tri::Geo<CMeshO> g;
                //create a vector of starting vertices in set 'B'
(border of feature)
                std::vector<CVertexO*> fro;
                bool ret;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                    if( setFBV[vi] == 'B')
                        fro.push_back(&(*vi));
                if(!fro.empty()) {
                    ret = g.DistanceFromFeature(m.cm, fro,
distanceConstraint);
                }
                float maxdist = distanceConstraint;
                float mindist = 0;
                //the distance is now stored in the quality, transfer
the quality to the distance
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if((*vi).Q() < distanceConstraint) {
                        if( setFBV[vi] == 'F') {
                            disth[vi] = -((*vi).Q());
                        }
                        if( setFBV[vi] == 'B') {
                            disth[vi] = 0;
                        }
                        if( setFBV[vi] == 'V') {
                            disth[vi] = (*vi).Q();
                        }
                    }
                    else {
                        disth[vi] = std::numeric_limits<float>::max();
                    }
                }
                //filter the distances
                if(filterOn){
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
```

```
                        //new distance to vertex is average of
neighborhood distances
                        if( disth[vi] < maxdist) {
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            float numberOfNeigbors = 0.0;
                            float sumOfDistances = 0.0;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                float tempDist =
disth[(*tempVertexPointer)];
                                if(tempDist !=
std::numeric_limits<float>::max()) {
                                    sumOfDistances = sumOfDistances +
tempDist;
                                    numberOfNeigbors++;
                                }
                            }
                            float inverseNumNeighbors = 1.0 /
numberOfNeigbors;
                            disth[vi] = inverseNumNeighbors *
sumOfDistances;
                        }
                    }
                }
                /****************************** Connecting points by
angle *************************/
                vector <CVertexO*> vectCPoints;
                int CPointCounter = 0;
                int vertexCounter = 0;
                if(findConnectingPoints) {
                    //Find Connecting Points
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if( setFBV[vi] == 'F' ) {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            char currentSet = 'a';
                            char initialSet = 'a';
                            int numberOfChanges = 0;
                            float numberOfNeighbors = 0.0;
                            float sumOfDivergence = 0.0;
                            float angle = 0.0;
                            bool addAngle = false;
                            CVertexO* firstVertex;
                            CVertexO* previousVertex;
                            char previousSet = 'a';
                            char firstSet = 'a';
                            int borderNeighborCounter = 0;
                            int featureNeighborCounter = 0;
```

```
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                char tempSetOfThisNeighbor =
(setFBV[&(*tempVertexPointer)]);
                                if(tempSetOfThisNeighbor == 'F') {
                                    featureNeighborCounter++;
                                    //don't add angle, we are still in
'F'
                                    addAngle = false;
                                }
                                if(tempSetOfThisNeighbor == 'B'){
                                    borderNeighborCounter++;
                                    //add angle, we are still in 'F'
                                    addAngle = true;
                                }
                                if(i == 0){
                                    //initialize the starting set
                                    currentSet =
setFBV[&(*tempVertexPointer)];
                                    initialSet =
setFBV[&(*tempVertexPointer)];
                                    firstVertex = tempVertexPointer;
                                    firstSet =
setFBV[&(*tempVertexPointer)];
                                    previousVertex = tempVertexPointer;
                                    previousSet =
setFBV[&(*tempVertexPointer)];
                                }
                                if(tempSetOfThisNeighbor !=
currentSet){
                                    //changed sets, update the
numberOfChanges
                                    currentSet =
setFBV[&(*tempVertexPointer)];
                                    numberOfChanges++;
                                }
                                //check if last point is in same set as
first point
                                if(i == (OneRing.allV.size() - 1)) {
                                    if(tempSetOfThisNeighbor !=
initialSet){
                                        numberOfChanges++;
                                    }
                                    if(firstSet == 'F' && currentSet ==
'F'){
                                        //don't add the angle
                                    }
                                    else{
                                        //calculate and add the angle
between tempVertexPointer and firstVertex
                                        vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                        //vector from gradient field of
w
```

```
                                        vcg::Point3f v2 =
(*firstVertex).P() - (*vi).P();
                                        double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                        double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                        v1 = v1 * (1.0/v1_magnitude);
                                        v2 = v2 * (1.0/v2_magnitude);
                                        double cosTheta = 0.0;
                                        if(v1_magnitude*v2_magnitude ==
0) {
                                            cosTheta = 0.0;
                                        }
                                        else{
                                            cosTheta = v1.X()*v2.X() +
v1.Y()*v2.Y() + v1.Z()*v2.Z();
                                        }
                                        float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                        angle = angle +
tempAngleinDegrees;
                                    }
                                }
                                if(i > 0){
                                    if(previousSet == 'F' && currentSet
== 'F'){
                                        //don't add the angle
                                    }
                                    else{
                                        //calculate and add the angle
between previousVertex and tempVertexPointer
                                        vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                        //vector from gradient field of
w
                                        vcg::Point3f v2 =
(*previousVertex).P() - (*vi).P();
                                        double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                        double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                        v1 = v1 * (1.0/v1_magnitude);
                                        v2 = v2 * (1.0/v2_magnitude);
                                        double cosTheta = 0.0;
                                        if(v1_magnitude*v2_magnitude ==
0) {
                                            cosTheta = 0.0;
                                        }
                                        else{
                                            cosTheta = v1.X()*v2.X() +
v1.Y()*v2.Y() + v1.Z()*v2.Z();
                                        }
                                        float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                        angle = angle +
tempAngleinDegrees;
                                    }
```

```
                                    //update the pointers
                                    previousVertex = tempVertexPointer;
                                    previousSet =
setFBV[&(*tempVertexPointer)];
                                }
                                float tempDivergence =
divfval[&(*tempVertexPointer)];
                                sumOfDivergence = sumOfDivergence +
tempDivergence;
                                numberOfNeighbors++;
                            }
                            //Candidate connecting point
                            if(numberOfChanges <= 2){
                                //not element of Fp, may be connecting
point
                                if(angle > connectingPointAngle) {
                                    (*vi).C() =
Color4b(Color4b::Yellow);
                                    CpointOwner[vi] = CPointCounter;
                                    source[vi] = &(*vi);
                                    vectCPoints.push_back(&(*vi));
                                    CPointCounter++;
                                }
                            }
                        }//in set 'F'
                        vertexCounter++;
                    }//go through vertices
                }//find connecting points and color them yellow
                //Find Mid Points
                std::vector<CMeshO::VertexPointer>  midPointVector;
                if(findMidPoints){
                    //************************************  Find
connecting paths from connecting points
                    //setup the seed vector of connecting points
                    std::vector<VertDist> seedVec;
                    std::vector<CVertexO*>::iterator fi;
                    for( fi  = vectCPoints.begin(); fi !=
vectCPoints.end() ; ++fi)
                    {
                        seedVec.push_back(VertDist(*fi,0.0));
                    }
                    std::vector<VertDist> frontier;
                    CMeshO::VertexPointer curr;
                    CMeshO::ScalarType unreached  =
std::numeric_limits<CMeshO::ScalarType>::max();
                    CMeshO::VertexPointer pw;
                    TempDataType TD(m.cm.vert, unreached);
                    std::vector <VertDist >::iterator ifr;
                    for(ifr = seedVec.begin(); ifr != seedVec.end();
++ifr){
                        TD[(*ifr).v].d = 0.0;
                        (*ifr).d = 0.0;
                        TD[(*ifr).v].source  = (*ifr).v;
                        frontier.push_back(VertDist((*ifr).v,0.0));
                    }
                    // initialize Heap
                    make_heap(frontier.begin(),frontier.end(),pred());
```

```
                    std::vector<int> doneConnectingPoints;
                    CMeshO::ScalarType curr_d,d_curr = 0.0,d_heap;
                    CMeshO::VertexPointer curr_s = NULL;
                    CMeshO::PerVertexAttributeHandle
<CMeshO::VertexPointer> * vertSource = NULL;
                    CMeshO::ScalarType max_distance=0.0;
                    std::vector<VertDist >:: iterator iv;
                    int connectionCounter = 0;
                    while(!frontier.empty() && max_distance < maxdist)
                    {

pop_heap(frontier.begin(),frontier.end(),pred());
                        curr = (frontier.back()).v;
                        int tempCpointOwner = CpointOwner[curr];
                        int tempFSetNum = FSetNum[curr];
                        curr_s = TD[curr].source;
                        if(vertSource!=NULL)
                            (*vertSource)[curr] = curr_s;
                        d_heap = (frontier.back()).d;
                        frontier.pop_back();
                        assert(TD[curr].d <= d_heap);
                        assert(curr_s != NULL);
                        if(TD[curr].d < d_heap )// a vertex whose
distance has been improved after it was inserted in the queue
                            continue;
                        assert(TD[curr].d == d_heap);
                        d_curr =  TD[curr].d;
                        if(d_curr > max_distance) {
                            max_distance = d_curr;
                        }
                        //check the vertices around the current point
                        (*curr).SetV();
                        vcg::tri::Nring<CMeshO> OneRing(&(*curr),
&m.cm);
                        OneRing.insertAndFlag1Ring(&(*curr));
                        (*curr).ClearV();
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            pw = OneRing.allV.at(i);
                            //just find the shortest distance between
points
                            curr_d = d_curr + vcg::Distance(pw-
>cP(),curr->cP());
                            //check if we are still searching from this
connecting point
                            std::vector<int>::iterator it;
                            it =
std::find(doneConnectingPoints.begin(), doneConnectingPoints.end(),
CpointOwner[curr]);
                            //we are still searching from this
connecting point
                            if (it == doneConnectingPoints.end()){
                                //This point has been explored by a
different Cpoint before
                                //deleted the bit about not connecting
to your own set: && FSetNum[pw] != FSetNum[curr]
                                if(CpointOwner[pw] != CpointOwner[curr]
&& CpointOwner[pw] != -1 && setFBV[&(*pw)] != 'F'){
```

```
                                    //This point has been explored
before, we may have a connection
                                    //(check if the CpointOwner is
active or already connected up with someone else
                                    //std::vector<int>::iterator chk;
                                    //chk =
std::find(doneConnectingPoints.begin(), doneConnectingPoints.end(),
CpointOwner[pw]);
                                    //we are still searching from the
CpointSource of the point we found
                                    //if(chk ==
doneConnectingPoints.end()) {
                                    (*pw).C() =
Color4b(Color4b::Magenta);
                                    connectionCounter++;
                                    source1[pw] = curr;
                                    //store the midpoint so we can make
the connecting path later
                                    midPointVector.push_back(pw);
                                    //add CpointOwner to vector of
doneConnectingPoints

doneConnectingPoints.push_back(CpointOwner[curr]);

doneConnectingPoints.push_back(CpointOwner[pw]);
                                    // }
                                    /* else{
                                }*/
                                }
                                //deleted  && curvature[pw] <
maxCurvatureInPath
                                else if(TD[(pw)].d > curr_d && curr_d <
maxdist && setFBV[&(*pw)] != 'F'){
                                    //This point has not been explored
before, keep looking
                                    //update source, Fsetnum,
CpointOwner
                                    CpointOwner[pw] = tempCpointOwner;
                                    source[pw] = curr;
                                    FSetNum[pw] = tempFSetNum;
                                    TD[(pw)].d = curr_d;
                                    TD[pw].source = curr;

frontier.push_back(VertDist(pw,curr_d));

push_heap(frontier.begin(),frontier.end(),pred());
                                    }
                                }
                                else {
                                    //not searching from this connecting
point anymore
                                }
                            }
                        }// end while
                    }//find mid points and color magenta
                    if(findMidPoints && findPathstoConnect) {
                        //********************** Make the connecting path
```

```
                    vector<CMeshO::VertexPointer>::iterator iter;
                    for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                        //track back source to beginning
                        CMeshO::VertexPointer tempVertexPointer =
*iter;
                        CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Green);
                            // setFBV[tempVertexPointer] = 'F';
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                        //track back source1 to beginning
                        tempVertexPointer = *iter;
                        tempVertexSource = source1[tempVertexPointer];
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Cyan);
                            //setFBV[tempVertexPointer] = 'F';
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                    }//make connecting path
                }//find paths to connect and color green and cyan
                if(addPathstoFeatureRegion){
                    //********************** Make the connecting path
                    vector<CMeshO::VertexPointer>::iterator iter;
                    for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                        //track back source to beginning
                        CMeshO::VertexPointer tempVertexPointer =
*iter;
                        CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                            setFBV[tempVertexPointer] = 'F';
                            newF[tempVertexPointer] = 1;
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                        //track back source1 to beginning
                        tempVertexPointer = *iter;
                        tempVertexSource = source1[tempVertexPointer];
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                            setFBV[tempVertexPointer] = 'F';
                            newF[tempVertexPointer] = 1;
                            tempVertexPointer = tempVertexSource;
```

```
                            tempVertexSource =
source[tempVertexPointer];
                        }
                    }
                }//add paths to feature region
                //******************************** get ready to
skeletonize and prune by changing sets to either F or V
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is not a feature
                    if(setFBV[vi] != 'F') {
                        setFBV[vi] = 'V';
                    }
                }
                //***************************** Skeletonize and prune
                //Apply Skeletonize operator
                if(skeletonizeAfter){
                    bool changes = false;
                    //if still changes loop through and skeletonize
                    do {
                        changes = false;
                        //go through the vertices and if 'F' check for
centers and discs
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                //check for centers and discs
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                bool neighborsAllF = true;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, vi not a center
                                    if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                        neighborsAllF = false;
                                    }
                                }
                                if(neighborsAllF){
                                    //mark as a center
                                    center[vi] = 1;
                                    //mark all neighbors as disc
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                        disc[&(*tempVertexPointer)] =
1;
                                    }
                                }
                            }//in set F
```

```
                        }//go through vertices
                        //go through vertices again, if not a center
and is a disc, check complexity and delete if not complex
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F' && center[vi] == 0 &&
disc[vi] == 1){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //delete from feature
                                    setFBV[vi] = 'V';
                                    center[vi] = 0;
                                    disc[vi] = 0;
                                    changes = true;
                                }
                            }//not a center, yes a disc
                        }//go through vertices
                        //******************************** reset disc
and center data for next iteration
```

```
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it is a feature, reset center and disc
data
                            if(setFBV[vi] == 'F') {
                                center[vi] = 0;
                                disc[vi] = 0;
                            }
                        }
                    } while (changes == true);//loop if changes
                }//skeletonize operator
                //prune
                if(pruneAfter){
                    //pune a certain number of times
                    for(int pruneIter = 0; pruneIter <
pruneAfterLength; pruneIter++){
                        int pruned = 0;
                        vertexCounter = 0;
                        //go through the vertices and if 'F' check
complexity
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F'){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
```

```
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //prune from feature
                                    setFBV[vi] = 'V';
                                    pruned++;
                                }
                            }//set
                            vertexCounter++;
                        }//go through vertices
                    }//number of times we prune
                }//prune after
                /*
                if(showFinalFeature) {
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                          (*vi).C() = Color4b(Color4b::Blue);
                        }
                        if(setFBV[vi] != 'F') {
                          (*vi).C() = Color4b(Color4b::White);
                        }
                    }
                }
                */
                if(showSegments){
                    //************************ Color the different
segments 2nd Time
                    //***** New DisjointSet
                    vcg::DisjointSet<CVertexO>* ptrDsetSegment = new
vcg::DisjointSet<CVertexO>();
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //Every vertex not in set 'F' starts as
disjoint set
                        if(setFBV[vi] != 'F') {
                            //put the vertex in the set D
                            ptrDsetSegment->MakeSet(&(*vi));
                        }
                    }
                    //***** Merge neighboring Sets
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not a feature, then find 1-ring and
merge sets
                        if(setFBV[vi] != 'F') {
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature then merge sets
```

```
                                if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                    //if they are not already in the
same set, merge them
                                    if(ptrDsetSegment->FindSet(&(*vi))
!= ptrDsetSegment->FindSet(tempVertexPointer)){
                                        ptrDsetSegment-
>Union(&(*vi),tempVertexPointer);
                                    }
                                }
                            }
                        }
                    }
                    //Find out how many distinct segment sets there are
******************************
                    int segmentSetCounter = 0;
                    vector <CVertexO*> parentVertexofSegmentSet;
                    //***** If a vertex is the parent of a DisjointSet,
it is a new Segment
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if (ptrDsetSegment->FindSet(&(*vi)) ==
&(*vi)){
                                //parent of a new set

parentVertexofSegmentSet.push_back(&(*vi));
                                segmentSetCounter++;
                            }
                        }
                    }
                    //******* find biggest set
                    typedef  std::vector < std::vector <CVertexO*> >
crmatrix;
                    crmatrix
vectorOfSetsWithVertices(segmentSetCounter, std::vector<CVertexO*>(0));
                    // ******************************  Create a vector
of vectors containing our sets / vertices
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is in set 'F', then find what set it
belongs to
                        if(setFBV[vi] != 'F') {
                            //find offset in parent vector--that is the
vertex set number
                            itVect =
find(parentVertexofSegmentSet.begin(), parentVertexofSegmentSet.end(),
ptrDsetSegment->FindSet(&(*vi)));
                            if( itVect !=
parentVertexofSegmentSet.end() ) {
                                int offset =
std::distance(parentVertexofSegmentSet.begin(), itVect);
                                //store the set number in the vertex
                                FSetNum[vi] = offset;
```

```
vectorOfSetsWithVertices[offset].push_back(&(*vi));
                            }
                        }
                    }
                    //Find the set with the most vertices
                    int maxInSet = 0;
                    int maxOffset = 0;
                    int secondBiggest = 0;
                    int secondBiggestOffset = 0;
                    for(int pos=0; pos <
vectorOfSetsWithVertices.size(); pos++)
                    {
                        if(vectorOfSetsWithVertices[pos].size() >
maxInSet) {
                            secondBiggest = maxInSet;
                            secondBiggestOffset = maxOffset;
                            maxInSet =
vectorOfSetsWithVertices[pos].size();
                            maxOffset = pos;
                        }
                    }
                    int numBigSets = 0;
                    int numLittleSets = 0;
                    for(int pos=0; pos <
vectorOfSetsWithVertices.size(); pos++)
                    {
                        if(pos != maxOffset) {
                            if(vectorOfSetsWithVertices[pos].size() >
(deletePercentOfBiggestSegment * secondBiggest)){
                                numBigSets++;
                            }
                            else{
                                numLittleSets++;
                            }
                        }
                    }
                    totalNumBigSets = numBigSets;
                    totalNumLittleSets = numLittleSets;
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).C() = Color4b(Color4b::Blue);
                        }
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if(FSetNum[vi] == maxOffset) {
                                (*vi).C() = Color4b(Color4b::White);
                            }
                            else {
                                //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                                int ScatterSize =
vectorOfSetsWithVertices.size();
```

```
                                Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                                (*vi).C()=BaseColor;
                            }
                        }
                    }
                    delete ptrDsetSegment;
                }//show segments
                if(removeLines) {
                    //Check if color same on both sides of F line
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            Color4b color1;
                            Color4b color2;
                            Color4b color3;
                            Color4b blueColor = Color4b(Color4b::Blue);
                            Color4b tempColor;
                            int colorCounter = 0;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if (i == 0) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    color1 = (*tempVertexPointer).C();
                                    colorCounter++;
                                }
                                if(i > 0 && i < OneRing.allV.size()) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    tempColor =
(*tempVertexPointer).C();
                                    if(colorCounter == 1) {
                                        if(tempColor != color1) {
                                            color2 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                    if(colorCounter == 2) {
                                        if(tempColor != color1 &&
tempColor != color2) {
                                            color3 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                }
                            }
                            //delete it from feature
                            if(colorCounter == 2 || colorCounter == 1){
                                if(colorCounter == 2){
```

```
                                        //delete from feature if surrounded
by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                        if(color2 != blueColor) {
                                            (*vi).C() = color2;
                                        }
                                    }
                                    if(colorCounter == 1) {
                                        //delete from feature if surrounded
by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                    }
                                }
                            }
                        }//check if color on both sides of line same
                    }//remove lines
                    //binary search stuff
                    if (numTeeth > totalNumBigSets)
                        first = intEpsilon + 1;  // repeat search in top
half.
                    else if (numTeeth < totalNumBigSets)
                        last = intEpsilon - 1; // repeat search in bottom
half.
                    else
                        foundIt = true;
                }//while first <= last
                if(foundIt == false)
                    if(foundIt == true) {
                    }
                if(foundIt == true) {
                    //now, if we found it, start moving epsilon smaller
till we know we have the smallest value
                    do {
                        //try the next smallest epsilon
                        epsilon = epsilon - .01;
                        /* ********************  Calculate curvature:
curvature.h computes the discrete gaussian curvature.
             <vcg/complex/algorithms/update/curvature.h>
             For further details, please, refer to:
             Discrete Differential-Geometry Operators for Triangulated 2-
Manifolds Mark Meyer,
             Mathieu Desbrun, Peter Schroder, Alan H. Barr VisMath '02,
Berlin </em>*/
                        // results stored in (*vi).Kh() (mean) and
(*vi).Kg() (guassian)

tri::UpdateCurvature<CMeshO>::MeanAndGaussian(m.cm);
                        // ********************  Put the curvature in the
quality: <vcg/complex/algorithms/update/quality.h>

tri::UpdateQuality<CMeshO>::VertexFromMeanCurvature(m.cm);
```

```
                    //****************** ComputePerVertexQualityMinMax
15 and 85 percent
                    std::pair<float,float> minmax =
std::make_pair(std::numeric_limits<float>::max(),-
std::numeric_limits<float>::max());
                    CMeshO::VertexIterator vi;
                    std::vector<float> QV;
                    QV.reserve(m.cm.vn);
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                        if(!(*vi).IsD()) QV.push_back((*vi).Q());
                    //bottom 15% maps to -1

std::nth_element(QV.begin(),QV.begin()+15*(m.cm.vn/100),QV.end());
                    float newmin=*(QV.begin()+m.cm.vn/100);
                    //top 15% maps to 1
                    std::nth_element(QV.begin(),QV.begin()+m.cm.vn-
15*(m.cm.vn/100),QV.end());
                    float newmax=*(QV.begin()+m.cm.vn-m.cm.vn/100);
                    minmax.first = newmin;
                    minmax.second = newmax;
                    // ***************************
MapCurvatureRangetoOneNegOne
                    float B = 8.0;
                    float min = minmax.first;
                    float max = minmax.second;
                    float upperBound = 0.0;
                    if(min < 0)
                        min = min * -1;
                    if(min > max)
                        upperBound = min;
                    else
                        upperBound = max;
                    float K = 0.5 * log10((1 + ((pow(2.0,B)-
2)/(pow(2.0,B)-1))) / ((1 - ((pow(2.0,B)-2)/(pow(2.0,B)-1)))));
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                        if(!(*vi).IsD())
                        {
                            float remapped = tanh((*vi).Q() * (K /
upperBound));
                            (*vi).Q() = remapped;
                            //Log( "Value to be remapped: %f
remapped: %f\n", (*vi).Q(), remapped );
                            // Log( "inside mapping");
                        }
                    vcg::DisjointSet<CVertexO>* ptrDset = new
vcg::DisjointSet<CVertexO>();
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //all vertices start as set V for Vertex and
unmarked
                        setFBV[vi] = 'V';
                        marked[vi] = 0;
                        disc[vi] = 0;
                        center[vi] = 0;
                        complex[vi] = 0;
```

```
                        newF[vi] = 0;
                        //all vertices start as -1 FSetNum
                        FSetNum[vi] = -1;
                        //none have been visited to connect C points,
so set CpointOwner to -1
                        CpointOwner[vi] = -1;
                        //color it White initially
                        (*vi).C() = Color4b(Color4b::White);
                        curvature[vi] = (*vi).Q();
                        //feature region, less than defined min
curvature
                        if(curvature[vi] < epsilon) {
                            //in set F for feature
                            setFBV[vi] = 'F';
                        }
                    }
                    // apply morphological opperators dilate and then
erode ****************************
                    if(morphologicalOperations){
                        //dilate 1-ring neighbor
                        for(int k = 0; k < 1; k++){
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                if(setFBV[vi] != 'V') {
                                    (*vi).SetV();
                                    vcg::tri::Nring<CMeshO>
OneRing(&(*vi), &m.cm);
                                    OneRing.insertAndFlag1Ring(&(*vi));
                                    (*vi).ClearV();
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                        //if the neighbor of the
current vertex is not a feature, mark the neighbor to be one
                                        //if it is negative curvature

if(setFBV[&(*tempVertexPointer)] == 'V') {
                                            //mark it to be 'N'

marked[&(*tempVertexPointer)] = 1;
                                        }
                                    }
                                }
                            }
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                //if it was marked, change it
                                if(marked[vi] == 1){
                                    setFBV[vi] = 'F';
                                    marked[vi] = 0;
                                }
                            }
                        }//grow once
                        //erode 1-ring neighbor
                        for(int k = 0; k < 1; k++){
```

```
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                if(setFBV[vi] != 'V') {
                                    (*vi).SetV();
                                    vcg::tri::Nring<CMeshO>
OneRing(&(*vi), &m.cm);
                                    OneRing.insertAndFlag1Ring(&(*vi));
                                    (*vi).ClearV();
                                    bool neighborsAllF = true;
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                        //if the neighbor of the
current vertex is not a feature, we will erode this vertex

if(setFBV[&(*tempVertexPointer)] == 'V') {
                                            neighborsAllF = false;
                                        }
                                    }
                                    if(neighborsAllF){
                                        //mark to not erode this vertex
this round
                                        marked[vi] = 1;
                                    }
                                }
                            }
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                //if it was marked to not erode, keep
it
                                if(marked[vi] == 1){
                                    //setFBV[vi] = 'F';
                                    marked[vi] = 0;
                                }
                                else{
                                    setFBV[vi] = 'V';
                                }
                            }
                        }//erode once
                    }
                    //Apply Skeletonize operator
                    if(skeletonize){
                        int debugCounter = 1;
                        bool changes = false;
                        //if still changes loop through and skeletonize
                        do {
                            changes = false;
                            //go through the vertices and if 'F' check
for centers and discs
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                if(setFBV[vi] == 'F') {
                                    //check for centers and discs
                                    (*vi).SetV();
                                    vcg::tri::Nring<CMeshO>
OneRing(&(*vi), &m.cm);
```

```
                                    OneRing.insertAndFlag1Ring(&(*vi));
                                    (*vi).ClearV();
                                    bool neighborsAllF = true;
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                        //if the neighbor of the
current vertex is not a feature, vi not a center

if(setFBV[&(*tempVertexPointer)] != 'F') {
                                            neighborsAllF = false;
                                        }
                                    }
                                    if(neighborsAllF){
                                        //mark as a center
                                        center[vi] = 1;
                                        //mark all neighbors as disc
                                        for (int i = 0; i <
OneRing.allV.size(); i++){
                                            CVertexO* tempVertexPointer
= OneRing.allV.at(i);
                                            disc[&(*tempVertexPointer)]
= 1;
                                        }
                                    }
                                }//in set F
                            }//go through vertices
                            //go through vertices again, if not a
center and is a disc, check complexity and delete if not complex
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                if(center[vi] == 0 && disc[vi] == 1){
                                    //check complexity
                                    (*vi).SetV();
                                    vcg::tri::Nring<CMeshO>
OneRing(&(*vi), &m.cm);
                                    OneRing.insertAndFlag1Ring(&(*vi));
                                    (*vi).ClearV();
                                    int numberOfChanges = 0;
                                    char iVertex;
                                    char iPlus1Vertex;
                                    CVertexO* tempVertexPointer;
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        if(i < (OneRing.allV.size() -
1)) {
                                            tempVertexPointer =
OneRing.allV.at(i);
                                            iVertex =
(setFBV[&(*tempVertexPointer)]);
                                            tempVertexPointer =
OneRing.allV.at(i+1);
                                            iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                        }
                                        else {
```

```
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from
feature
                                if(numberOfChanges < 4){
                                    //delete from feature
                                    setFBV[vi] = 'V';
                                    center[vi] = 0;
                                    disc[vi] = 0;
                                    changes = true;
                                }
                            }//not a center, yes a disc
                        }//go through vertices
                        //******************************** reset
disc and center data for next iteration
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                center[vi] = 0;
                                disc[vi] = 0;
                            }
                        }
                        /* if(debugCounter == 1){
                   changes = false;
               }*/
                    } while (changes == true);//loop if changes
                }//skeletonize operator
                //prune
                if(prune){
                    //prune a certain number of times
                    for(int pruneIter = 0; pruneIter < pruneLength;
pruneIter++){
                        int pruned = 0;
                        //go through the vertices and if 'F' check
complexity
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F'){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO>
OneRing(&(*vi), &m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
```

```
                                    char iVertex;
                                    char iPlus1Vertex;
                                    CVertexO* tempVertexPointer;
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        if(i < (OneRing.allV.size() -
1)) {
                                            tempVertexPointer =
OneRing.allV.at(i);
                                            iVertex =
(setFBV[&(*tempVertexPointer)]);
                                            tempVertexPointer =
OneRing.allV.at(i+1);
                                            iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                        }
                                        else {
                                            tempVertexPointer =
OneRing.allV.at(i);
                                            iVertex =
(setFBV[&(*tempVertexPointer)]);
                                            tempVertexPointer =
OneRing.allV.at(0);
                                            iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                        }
                                        //change noted
                                        if(iVertex != iPlus1Vertex){
                                            numberOfChanges++;
                                        }
                                    }
                                    //not complex, delete it from
feature
                                    if(numberOfChanges < 4){
                                        //prune from feature
                                        setFBV[vi] = 'V';
                                        pruned++;
                                    }
                                }//checking current vertex
                            }//go through vertices
                        }//number of times we prune
                    }//prune before
                    //***************************** Initialize the
Disjoint sets of 'F' and delete any that are small
*****************************
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F'){
                            //put the vertex in the set D
                            ptrDset->MakeSet(&(*vi));
                        }
                    }
                    /*
            if(showMinAndMaxFeature || onlyShowFeature){
                //go through the vertices and color them according to
sets
                for(vi=m.cm.vert.begin();vi!=m.cm.vert.end();++vi) {
```

```
                    if(setFBV[&(*vi)] == 'F')
                        (*vi).C() = Color4b(Color4b::Blue);
                    if(setFBV[&(*vi)] == 'N')
                        (*vi).C() = Color4b(Color4b::Green);
                      //more than defined maximum curvature
                   // if(!onlyShowFeature) {
                     //   if(curvature[vi] > eta)
                       //    (*vi).C() = Color4b(Color4b::Red);
                   // }
                }
            }
        */
                    //print number of sets
                    //ptrDset->printNumberOfSets();
                    //Go through vertices again--if in set 'F',
******************************
                    //then find 1-Ring of vertices and merge sets if
orginal vertex is 'F'
                    //and new 1-ring vertex is 'F', define set 'B'
                    int blueCounter = 0;
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is a feature, then find 1-ring and
merge sets
                        if(setFBV[vi] == 'F') {
                            blueCounter++;
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is a feature then merge sets
                                if(setFBV[&(*tempVertexPointer)] ==
'F') {
                                    //if they are not already in the
same set, merge them
                                    if(ptrDset->FindSet(&(*vi)) !=
ptrDset->FindSet(tempVertexPointer)){
                                        ptrDset-
>Union(&(*vi),tempVertexPointer);
                                    }
                                }
                                else if(setFBV[&(*tempVertexPointer)]
== 'V'){
                                    //if the neighbor of the blue
vertex is just a vertex it is on the border
                                    //we do this after we delete the
small sets of 'F'
                                    //setFBV[&(*tempVertexPointer)] =
'B';
                                }
                            }
                        }
                    }
```

```
                    //Find out how many distinct sets there are
******************************
                    int setCounter = 0;
                    vector <CVertexO*> parentVertexofSet;
                    typedef  std::vector < std::vector <CVertexO*> >
matrix;
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is in set 'F', then find what set it
belongs to
                        if(setFBV[vi] == 'F') {
                            if (ptrDset->FindSet(&(*vi)) == &(*vi)){
                                //parent of a new set
                                parentVertexofSet.push_back(&(*vi));
                                setCounter++;
                            }
                        }
                    }
                    matrix allSetsWithVertices(setCounter,
std::vector<CVertexO*>(0));
                    // ******************************  Create a vector
of vectors containing our sets / vertices
                    vector<CVertexO*>::iterator itVect;
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is in set 'F', then find what set it
belongs to
                        if(setFBV[vi] == 'F') {
                            //find offset in parent vector--that is the
vertex set number
                            itVect = find(parentVertexofSet.begin(),
parentVertexofSet.end(), ptrDset->FindSet(&(*vi)));
                            if( itVect != parentVertexofSet.end() ) {
                                int offset =
std::distance(parentVertexofSet.begin(), itVect);
                                //store the set number in the vertex
                                FSetNum[vi] = offset;

allSetsWithVertices[offset].push_back(&(*vi));
                            }
                        }
                    }
                    //delete the disjoint set
                    delete ptrDset;
                    //print out the stored set data
                    /*
          for(int pos=0; pos < allSetsWithVertices.size(); pos++)
          {
          }*/
                    //delete sets of D if the number of vertices is
less than .01 D
                    for(int pos=0; pos < allSetsWithVertices.size();
pos++)
                    {
                        if(allSetsWithVertices[pos].size() <=
deletePercentOfF*blueCounter) {
```

```
                            for(int i=0; i <
allSetsWithVertices[pos].size(); i++) {
                                (*(allSetsWithVertices[pos][i])).C() =
Color4b(Color4b::White);

setFBV[(*(allSetsWithVertices[pos][i]))] = 'V';
                            }
                        }
                    }
                    //************************ Color the different
segments 1st Time and then delete F lines that have same color on both
sides
                    //** This clears up a lot of the extraneous
connecting points before we close gaps.
                    if(showSegments1st){
                        //***** New DisjointSet
                        vcg::DisjointSet<CVertexO>* DisjointSetsToColor
= new vcg::DisjointSet<CVertexO>();
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //Every vertex not in set 'F' starts as
disjoint set
                            if(setFBV[vi] != 'F') {
                                //put the vertex in the set D
                                DisjointSetsToColor->MakeSet(&(*vi));
                            }
                        }
                        //***** Merge neighboring Sets
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it is not a feature, then find 1-ring
and merge sets
                            if(setFBV[vi] != 'F') {
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature then merge sets
                                    if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                        //if they are not already in
the same set, merge them
                                        if(DisjointSetsToColor-
>FindSet(&(*vi)) != DisjointSetsToColor->FindSet(tempVertexPointer)){
                                            DisjointSetsToColor-
>Union(&(*vi),tempVertexPointer);
                                        }
                                    }
                                }
                            }
                        }
                        //Find out how many distinct segment sets there
are ******************************
```

```
                        int segmentSetCounter = 0;
                        vector <CVertexO*> parentVertexofDisjointSets;
                        //***** If a vertex is the parent of a
DisjointSet, it is a new Segment
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it is not in set 'F', then find what
set it belongs to
                            if(setFBV[vi] != 'F') {
                                if (DisjointSetsToColor-
>FindSet(&(*vi)) == &(*vi)){
                                    //parent of a new set

parentVertexofDisjointSets.push_back(&(*vi));
                                    segmentSetCounter++;
                                }
                            }
                        }
                        //******* find biggest set
                        typedef  std::vector < std::vector <CVertexO*>
> crmatrix;
                        crmatrix
vectorOfDisjointSetsWithVertices(segmentSetCounter,
std::vector<CVertexO*>(0));
                        vector<CVertexO*>::iterator itVect;
                        // ******************************  Create a
vector of vectors containing our sets / vertices
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it is in set 'F', then find what set
it belongs to
                            if(setFBV[vi] != 'F') {
                                //find offset in parent vector--that is
the vertex set number
                                itVect =
find(parentVertexofDisjointSets.begin(),
parentVertexofDisjointSets.end(), DisjointSetsToColor-
>FindSet(&(*vi)));
                                if( itVect !=
parentVertexofDisjointSets.end() ) {
                                    int offset =
std::distance(parentVertexofDisjointSets.begin(), itVect);
                                    //store the set number in the
vertex
                                    FSetNum[vi] = offset;

vectorOfDisjointSetsWithVertices[offset].push_back(&(*vi));
                                }
                            }
                        }
                        //*********** For each set, sum the positive
curvature and store in new vector
                        vector<float>
vectorOfSetPosCurvatureSums(vectorOfDisjointSetsWithVertices.size(),0);
                        for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                        {
```

```
                            float tempSumOfCurvature = 0.0;
                            for(int numInSet=0; numInSet <
vectorOfDisjointSetsWithVertices[pos].size(); numInSet++){
                                CVertexO* tempVertex =
vectorOfDisjointSetsWithVertices[pos][numInSet];
                                float tempCurvature =
curvature[tempVertex];
                                //if it's positive, add it up
                                if(tempCurvature > 0.0) {
                                    tempSumOfCurvature =
tempSumOfCurvature + tempCurvature;
                                }
                            }
                            vectorOfSetPosCurvatureSums[pos] =
tempSumOfCurvature;
                        }
                        //Find the set with the most vertices
                        int maxInSet = 0;
                        int maxOffset = 0;
                        int secondBiggest = 0;
                        int secondBiggestOffset = 0;
                        for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                        {

if(vectorOfDisjointSetsWithVertices[pos].size() > maxInSet) {
                                secondBiggest = maxInSet;
                                secondBiggestOffset = maxOffset;
                                maxInSet =
vectorOfDisjointSetsWithVertices[pos].size();
                                maxOffset = pos;
                            }
                        }
                        int numBigSets = 0;
                        int numLittleSets = 0;
                        for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                        {
                            if(pos != maxOffset) {

if(vectorOfDisjointSetsWithVertices[pos].size() >
(deletePercentOfBiggestSegment * secondBiggest)){
                                    numBigSets++;
                                }
                                else{
                                    numLittleSets++;
                                }
                            }
                        }
                        //color the different sets
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                (*vi).C() = Color4b(Color4b::Blue);
                            }
                            //if it is not in set 'F', then find what
set it belongs to
```

```
                            if(setFBV[vi] != 'F') {
                                if(FSetNum[vi] == maxOffset) {
                                    (*vi).C() =
Color4b(Color4b::White);
                                }
                                else {
                                    //color ramp
//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                                    int ScatterSize =
vectorOfDisjointSetsWithVertices.size();
                                    Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                                    (*vi).C()=BaseColor;
                                }
                            }
                        }
                        delete DisjointSetsToColor;
                    }//show segments
                    if(removeLines1st) {
                        //Check if color same on both sides of F line
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                Color4b color1;
                                Color4b color2;
                                Color4b color3;
                                Color4b whiteColor =
Color4b(Color4b::White);
                                Color4b blueColor =
Color4b(Color4b::Blue);
                                Color4b tempColor;
                                int colorCounter = 0;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if (i == 0) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        color1 =
(*tempVertexPointer).C();
                                        colorCounter++;
                                    }
                                    if(i > 0 && i <
OneRing.allV.size()) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        tempColor =
(*tempVertexPointer).C();
                                        if(colorCounter == 1) {
                                            if(tempColor != color1) {
                                                color2 = tempColor;
```

```
                                        colorCounter++;
                                    }
                                }
                                if(colorCounter == 2) {
                                    if(tempColor != color1 &&
tempColor != color2) {
                                        color3 = tempColor;
                                        colorCounter++;
                                    }
                                }
                            }
                        }
                        //delete it from feature
                        if(colorCounter == 2 || colorCounter ==
1){
                            if(colorCounter == 2){
                                if(color1 != whiteColor &&
color2 != whiteColor) {
                                    //delete from feature if
surrounded by a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                    if(color2 != blueColor) {
                                        (*vi).C() = color2;
                                    }
                                }
                            }
                            if(colorCounter == 1) {
                                if(color1 != whiteColor) {
                                    //delete from feature if
surrounded by a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                }
                            }
                        }
                    }
                }//check if color on both sides of line same
            }//remove lines
            //********************************* Mark the
borders after deleting features
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                //if it is a feature, then find 1-ring and mark
set 'B'
                if(setFBV[vi] == 'F') {
                    vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                    OneRing.insertAndFlag1Ring(&(*vi));
                    for (int i = 0; i < OneRing.allV.size();
i++){
                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
```

```
                                if(setFBV[&(*tempVertexPointer)] ==
'V'){
                                    //if the neighbor of the 'F' vertex
is just a vertex it is on the border
                                    setFBV[&(*tempVertexPointer)] =
'B';
                                }
                            }
                        }
                        // ******************************  Estimate
distance from feature region
                        tri::Geo<CMeshO> g;
                        //create a vector of starting vertices in set 'B'
(border of feature)
                        std::vector<CVertexO*> fro;
                        bool ret;
                        for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                            if( setFBV[vi] == 'B')
                                fro.push_back(&(*vi));
                        if(!fro.empty()) {
                            ret = g.DistanceFromFeature(m.cm, fro,
distanceConstraint);
                        }
                        float maxdist = distanceConstraint;
                        float mindist = 0;
                        //the distance is now stored in the quality,
transfer the quality to the distance
                        for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                            if((*vi).Q() < distanceConstraint) {
                                if( setFBV[vi] == 'F') {
                                    disth[vi] = -((*vi).Q());
                                }
                                if( setFBV[vi] == 'B') {
                                    disth[vi] = 0;
                                }
                                if( setFBV[vi] == 'V') {
                                    disth[vi] = (*vi).Q();
                                }
                            }
                            else {
                                disth[vi] =
std::numeric_limits<float>::max();
                            }
                        }
                        //filter the distances
                        if(filterOn){
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                //new distance to vertex is average of
neighborhood distances
                                if( disth[vi] < maxdist) {
                                    vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                    OneRing.insertAndFlag1Ring(&(*vi));
```

```
                                float numberOfNeigbors = 0.0;
                                float sumOfDistances = 0.0;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    float tempDist =
disth[(*tempVertexPointer)];
                                    if(tempDist !=
std::numeric_limits<float>::max()) {
                                        sumOfDistances = sumOfDistances
+ tempDist;
                                        numberOfNeigbors++;
                                    }
                                }
                                float inverseNumNeighbors = 1.0 /
numberOfNeigbors;
                                disth[vi] = inverseNumNeighbors *
sumOfDistances;
                            }
                        }
                    }
                    /****************************** Connecting points
by angle *************************/
                    vector <CVertexO*> vectCPoints;
                    int CPointCounter = 0;
                    int vertexCounter = 0;
                    if(findConnectingPoints) {
                        //Find Connecting Points
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if( setFBV[vi] == 'F' ) {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                char currentSet = 'a';
                                char initialSet = 'a';
                                int numberOfChanges = 0;
                                float numberOfNeighbors = 0.0;
                                float sumOfDivergence = 0.0;
                                float angle = 0.0;
                                bool addAngle = false;
                                CVertexO* firstVertex;
                                CVertexO* previousVertex;
                                char previousSet = 'a';
                                char firstSet = 'a';
                                int borderNeighborCounter = 0;
                                int featureNeighborCounter = 0;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    char tempSetOfThisNeighbor =
(setFBV[&(*tempVertexPointer)]);
                                    if(tempSetOfThisNeighbor == 'F') {
```

```
                                        featureNeighborCounter++;
                                        //don't add angle, we are still
in 'F'
                                        addAngle = false;
                                    }
                                    if(tempSetOfThisNeighbor == 'B'){
                                        borderNeighborCounter++;
                                        //add angle, we are still in
'F'
                                        addAngle = true;
                                    }
                                    if(i == 0){
                                        //initialize the starting set
                                        currentSet =
setFBV[&(*tempVertexPointer)];
                                        initialSet =
setFBV[&(*tempVertexPointer)];
                                        firstVertex =
tempVertexPointer;
                                        firstSet =
setFBV[&(*tempVertexPointer)];
                                        previousVertex =
tempVertexPointer;
                                        previousSet =
setFBV[&(*tempVertexPointer)];
                                    }
                                    if(tempSetOfThisNeighbor !=
currentSet){
                                        //changed sets, update the
numberOfChanges
                                        currentSet =
setFBV[&(*tempVertexPointer)];
                                        numberOfChanges++;
                                    }
                                    //check if last point is in same
set as first point
                                    if(i == (OneRing.allV.size() - 1))
{
                                        if(tempSetOfThisNeighbor !=
initialSet){
                                            numberOfChanges++;
                                        }
                                        if(firstSet == 'F' &&
currentSet == 'F'){
                                            //don't add the angle
                                        }
                                        else{
                                            //calculate and add the
angle between tempVertexPointer and firstVertex
                                            vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                            //vector from gradient
field of w
                                            vcg::Point3f v2 =
(*firstVertex).P() - (*vi).P();
                                            double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
```

```
                                                double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                                v1 = v1 *
(1.0/v1_magnitude);
                                                v2 = v2 *
(1.0/v2_magnitude);
                                                double cosTheta = 0.0;

if(v1_magnitude*v2_magnitude == 0) {
                                                    cosTheta = 0.0;
                                                }
                                                else{
                                                    cosTheta =
v1.X()*v2.X() + v1.Y()*v2.Y() + v1.Z()*v2.Z();
                                                }
                                                float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                                angle = angle +
tempAngleinDegrees;
                                            }
                                        }
                                        if(i > 0){
                                            if(previousSet == 'F' &&
currentSet == 'F'){
                                                //don't add the angle
                                            }
                                            else{
                                                //calculate and add the
angle between previousVertex and tempVertexPointer
                                                vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                                //vector from gradient
field of w
                                                vcg::Point3f v2 =
(*previousVertex).P() - (*vi).P();
                                                double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                                double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                                v1 = v1 *
(1.0/v1_magnitude);
                                                v2 = v2 *
(1.0/v2_magnitude);
                                                double cosTheta = 0.0;

if(v1_magnitude*v2_magnitude == 0) {
                                                    cosTheta = 0.0;
                                                }
                                                else{
                                                    cosTheta =
v1.X()*v2.X() + v1.Y()*v2.Y() + v1.Z()*v2.Z();
                                                }
                                                float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                                angle = angle +
tempAngleinDegrees;
                                            }
```

```
                                        //update the pointers
                                        previousVertex =
tempVertexPointer;
                                        previousSet =
setFBV[&(*tempVertexPointer)];
                                    }
                                    float tempDivergence =
divfval[&(*tempVertexPointer)];
                                    sumOfDivergence = sumOfDivergence +
tempDivergence;
                                    numberOfNeighbors++;
                                }
                                //Candidate connecting point
                                if(numberOfChanges <= 2){
                                    //not element of Fp, may be
connecting point
                                    if(angle > connectingPointAngle) {
                                        (*vi).C() =
Color4b(Color4b::Yellow);
                                        CpointOwner[vi] =
CPointCounter;
                                        source[vi] = &(*vi);
                                        vectCPoints.push_back(&(*vi));
                                        CPointCounter++;
                                    }
                                }
                            }//in set 'F'
                            vertexCounter++;
                        }//go through vertices
                    }//find connecting points and color them yellow
                    //Find Mid Points
                    std::vector<CMeshO::VertexPointer>  midPointVector;
                    if(findMidPoints){
                        //************************************  Find
connecting paths from connecting points
                        //setup the seed vector of connecting points
                        std::vector<VertDist> seedVec;
                        std::vector<CVertexO*>::iterator fi;
                        for( fi  = vectCPoints.begin(); fi !=
vectCPoints.end() ; ++fi)
                        {
                            seedVec.push_back(VertDist(*fi,0.0));
                        }
                        std::vector<VertDist> frontier;
                        CMeshO::VertexPointer curr;
                        CMeshO::ScalarType unreached  =
std::numeric_limits<CMeshO::ScalarType>::max();
                        CMeshO::VertexPointer pw;
                        TempDataType TD(m.cm.vert, unreached);
                        std::vector <VertDist >::iterator ifr;
                        for(ifr = seedVec.begin(); ifr !=
seedVec.end(); ++ifr){
                            TD[(*ifr).v].d = 0.0;
                            (*ifr).d = 0.0;
                            TD[(*ifr).v].source  = (*ifr).v;
                            frontier.push_back(VertDist((*ifr).v,0.0));
                        }
```

```
                        // initialize Heap

make_heap(frontier.begin(),frontier.end(),pred());
                        std::vector<int> doneConnectingPoints;
                        CMeshO::ScalarType curr_d,d_curr = 0.0,d_heap;
                        CMeshO::VertexPointer curr_s = NULL;
                        CMeshO::PerVertexAttributeHandle
<CMeshO::VertexPointer> * vertSource = NULL;
                        CMeshO::ScalarType max_distance=0.0;
                        std::vector<VertDist >:: iterator iv;
                        int connectionCounter = 0;
                        while(!frontier.empty() && max_distance <
maxdist)
                        {

pop_heap(frontier.begin(),frontier.end(),pred());
                            curr = (frontier.back()).v;
                            int tempCpointOwner = CpointOwner[curr];
                            int tempFSetNum = FSetNum[curr];
                            curr_s = TD[curr].source;
                            if(vertSource!=NULL)
                                (*vertSource)[curr] = curr_s;
                            d_heap = (frontier.back()).d;
                            frontier.pop_back();
                            assert(TD[curr].d <= d_heap);
                            assert(curr_s != NULL);
                            if(TD[curr].d < d_heap )// a vertex whose
distance has been improved after it was inserted in the queue
                                continue;
                            assert(TD[curr].d == d_heap);
                            d_curr =  TD[curr].d;
                            if(d_curr > max_distance) {
                                max_distance = d_curr;
                            }
                            //check the vertices around the current
point
                            (*curr).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*curr),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*curr));
                            (*curr).ClearV();
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                pw = OneRing.allV.at(i);
                                //just find the shortest distance
between points
                                curr_d = d_curr + vcg::Distance(pw-
>cP(),curr->cP());
                                //check if we are still searching from
this connecting point
                                std::vector<int>::iterator it;
                                it =
std::find(doneConnectingPoints.begin(), doneConnectingPoints.end(),
CpointOwner[curr]);
                                //we are still searching from this
connecting point
                                if (it == doneConnectingPoints.end()){
```

```
                                        //This point has been explored by a
different Cpoint before
                                        //deleted the bit about not
connecting to your own set: && FSetNum[pw] != FSetNum[curr]
                                        if(CpointOwner[pw] !=
CpointOwner[curr] && CpointOwner[pw] != -1 && setFBV[&(*pw)] != 'F'){
                                            //This point has been explored
before, we may have a connection
                                            //(check if the CpointOwner is
active or already connected up with someone else
                                            //std::vector<int>::iterator
chk;
                                            //chk =
std::find(doneConnectingPoints.begin(), doneConnectingPoints.end(),
CpointOwner[pw]);
                                            //we are still searching from
the CpointSource of the point we found
                                            //if(chk ==
doneConnectingPoints.end()) {
                                            (*pw).C() =
Color4b(Color4b::Magenta);
                                            connectionCounter++;
                                            source1[pw] = curr;
                                            //store the midpoint so we can
make the connecting path later
                                            midPointVector.push_back(pw);
                                            //add CpointOwner to vector of
doneConnectingPoints

doneConnectingPoints.push_back(CpointOwner[curr]);

doneConnectingPoints.push_back(CpointOwner[pw]);
                                            // }
                                            /* else{
                                    }*/
                                        }
                                        //deleted  && curvature[pw] <
maxCurvatureInPath
                                        else if(TD[(pw)].d > curr_d &&
curr_d < maxdist && setFBV[&(*pw)] != 'F'){
                                            //This point has not been
explored before, keep looking
                                            //update source, Fsetnum,
CpointOwner
                                            CpointOwner[pw] =
tempCpointOwner;
                                            source[pw] = curr;
                                            FSetNum[pw] = tempFSetNum;
                                            TD[(pw)].d = curr_d;
                                            TD[pw].source = curr;

frontier.push_back(VertDist(pw,curr_d));

push_heap(frontier.begin(),frontier.end(),pred());
                                        }
                                    }
                                    else {
```

```
                                    //not searching from this
connecting point anymore
                                }
                            }
                        }// end while
                    }//find mid points and color magenta
                    if(findMidPoints && findPathstoConnect) {
                        //********************** Make the connecting
path
                        vector<CMeshO::VertexPointer>::iterator iter;
                        for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                            //track back source to beginning
                            CMeshO::VertexPointer tempVertexPointer =
*iter;
                            CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                            while(tempVertexPointer !=
tempVertexSource){
                                (*tempVertexPointer).C() =
Color4b(Color4b::Green);
                                // setFBV[tempVertexPointer] = 'F';
                                tempVertexPointer = tempVertexSource;
                                tempVertexSource =
source[tempVertexPointer];
                            }
                            //track back source1 to beginning
                            tempVertexPointer = *iter;
                            tempVertexSource =
source1[tempVertexPointer];
                            while(tempVertexPointer !=
tempVertexSource){
                                (*tempVertexPointer).C() =
Color4b(Color4b::Cyan);
                                //setFBV[tempVertexPointer] = 'F';
                                tempVertexPointer = tempVertexSource;
                                tempVertexSource =
source[tempVertexPointer];
                            }
                        }//make connecting path
                    }//find paths to connect and color green and cyan
                    if(addPathstoFeatureRegion){
                        //********************** Make the connecting
path
                        vector<CMeshO::VertexPointer>::iterator iter;
                        for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                            //track back source to beginning
                            CMeshO::VertexPointer tempVertexPointer =
*iter;
                            CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                            while(tempVertexPointer !=
tempVertexSource){
                                (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                                setFBV[tempVertexPointer] = 'F';
```

```
                                newF[tempVertexPointer] = 1;
                                tempVertexPointer = tempVertexSource;
                                tempVertexSource =
source[tempVertexPointer];
                            }
                            //track back source1 to beginning
                            tempVertexPointer = *iter;
                            tempVertexSource =
source1[tempVertexPointer];
                            while(tempVertexPointer !=
tempVertexSource){
                                (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                                setFBV[tempVertexPointer] = 'F';
                                newF[tempVertexPointer] = 1;
                                tempVertexPointer = tempVertexSource;
                                tempVertexSource =
source[tempVertexPointer];
                            }
                        }
                    }//add paths to feature region
                    //******************************** get ready to
skeletonize and prune by changing sets to either F or V
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not a feature
                        if(setFBV[vi] != 'F') {
                            setFBV[vi] = 'V';
                        }
                    }
                    //***************************** Skeletonize and
prune
                    //Apply Skeletonize operator
                    if(skeletonizeAfter){
                        bool changes = false;
                        //if still changes loop through and skeletonize
                        do {
                            changes = false;
                            //go through the vertices and if 'F' check
for centers and discs
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                if(setFBV[vi] == 'F') {
                                    //check for centers and discs
                                    (*vi).SetV();
                                    vcg::tri::Nring<CMeshO>
OneRing(&(*vi), &m.cm);
                                    OneRing.insertAndFlag1Ring(&(*vi));
                                    (*vi).ClearV();
                                    bool neighborsAllF = true;
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                        //if the neighbor of the
current vertex is not a feature, vi not a center
```

```
if(setFBV[&(*tempVertexPointer)] != 'F') {
                                        neighborsAllF = false;
                                    }
                                }
                                if(neighborsAllF){
                                    //mark as a center
                                    center[vi] = 1;
                                    //mark all neighbors as disc
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer
= OneRing.allV.at(i);
                                        disc[&(*tempVertexPointer)]
= 1;
                                    }
                                }
                            }//in set F
                        }//go through vertices
                        //go through vertices again, if not a
center and is a disc, check complexity and delete if not complex
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F' && center[vi] == 0
&& disc[vi] == 1){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO>
OneRing(&(*vi), &m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() -
1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
```

```
                                        }
                                        //change noted
                                        if(iVertex != iPlus1Vertex){
                                            numberOfChanges++;
                                        }
                                    }
                                    //not complex, delete it from
feature
                                    if(numberOfChanges < 4){
                                        //delete from feature
                                        setFBV[vi] = 'V';
                                        center[vi] = 0;
                                        disc[vi] = 0;
                                        changes = true;
                                    }
                                }//not a center, yes a disc
                            }//go through vertices
                            //******************************** reset
disc and center data for next iteration
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                //if it is a feature, reset center and
disc data
                                if(setFBV[vi] == 'F') {
                                    center[vi] = 0;
                                    disc[vi] = 0;
                                }
                            }
                        } while (changes == true);//loop if changes
                    }//skeletonize operator
                    //prune
                    if(pruneAfter){
                        //pune a certain number of times
                        for(int pruneIter = 0; pruneIter <
pruneAfterLength; pruneIter++){
                            int pruned = 0;
                            vertexCounter = 0;
                            //go through the vertices and if 'F' check
complexity
                            for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                                if(setFBV[vi] == 'F'){
                                    //check complexity
                                    (*vi).SetV();
                                    vcg::tri::Nring<CMeshO>
OneRing(&(*vi), &m.cm);
                                    OneRing.insertAndFlag1Ring(&(*vi));
                                    (*vi).ClearV();
                                    int numberOfChanges = 0;
                                    char iVertex;
                                    char iPlus1Vertex;
                                    CVertexO* tempVertexPointer;
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        if(i < (OneRing.allV.size() -
1)) {
```

```
                                            tempVertexPointer =
OneRing.allV.at(i);
                                            iVertex =
(setFBV[&(*tempVertexPointer)]);
                                            tempVertexPointer =
OneRing.allV.at(i+1);
                                            iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                        }
                                        else {
                                            tempVertexPointer =
OneRing.allV.at(i);
                                            iVertex =
(setFBV[&(*tempVertexPointer)]);
                                            tempVertexPointer =
OneRing.allV.at(0);
                                            iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                        }
                                        //change noted
                                        if(iVertex != iPlus1Vertex){
                                            numberOfChanges++;
                                        }
                                    }
                                    //not complex, delete it from
feature
                                    if(numberOfChanges < 4){
                                        //prune from feature
                                        setFBV[vi] = 'V';
                                        pruned++;
                                    }
                                }//set
                                vertexCounter++;
                            }//go through vertices
                        }//number of times we prune
                    }//prune after
                    /*
                if(showFinalFeature) {
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                          (*vi).C() = Color4b(Color4b::Blue);
                        }
                        if(setFBV[vi] != 'F') {
                          (*vi).C() = Color4b(Color4b::White);
                        }
                    }
                }
                */
                    if(showSegments){
                        //************************ Color the different
segments 2nd Time
                        //***** New DisjointSet
                        vcg::DisjointSet<CVertexO>* ptrDsetSegment =
new vcg::DisjointSet<CVertexO>();
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
```

```
                            //Every vertex not in set 'F' starts as
disjoint set
                            if(setFBV[vi] != 'F') {
                                //put the vertex in the set D
                                ptrDsetSegment->MakeSet(&(*vi));
                            }
                        }
                        //***** Merge neighboring Sets
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it is not a feature, then find 1-ring
and merge sets
                            if(setFBV[vi] != 'F') {
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature then merge sets
                                    if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                        //if they are not already in
the same set, merge them
                                        if(ptrDsetSegment-
>FindSet(&(*vi)) != ptrDsetSegment->FindSet(tempVertexPointer)){
                                            ptrDsetSegment-
>Union(&(*vi),tempVertexPointer);
                                        }
                                    }
                                }
                            }
                        }
                        //Find out how many distinct segment sets there
are ******************************
                        int segmentSetCounter = 0;
                        vector <CVertexO*> parentVertexofSegmentSet;
                        //***** If a vertex is the parent of a
DisjointSet, it is a new Segment
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it is not in set 'F', then find what
set it belongs to
                            if(setFBV[vi] != 'F') {
                                if (ptrDsetSegment->FindSet(&(*vi)) ==
&(*vi)){
                                    //parent of a new set

parentVertexofSegmentSet.push_back(&(*vi));
                                    segmentSetCounter++;
                                }
                            }
                        }
                        //******* find biggest set
```

```
                        typedef  std::vector < std::vector <CVertexO*>
> crmatrix;
                        crmatrix
vectorOfSetsWithVertices(segmentSetCounter, std::vector<CVertexO*>(0));
                        // ******************************  Create a
vector of vectors containing our sets / vertices
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it is in set 'F', then find what set
it belongs to
                            if(setFBV[vi] != 'F') {
                                //find offset in parent vector--that is
the vertex set number
                                itVect =
find(parentVertexofSegmentSet.begin(), parentVertexofSegmentSet.end(),
ptrDsetSegment->FindSet(&(*vi)));
                                if( itVect !=
parentVertexofSegmentSet.end() ) {
                                    int offset =
std::distance(parentVertexofSegmentSet.begin(), itVect);
                                    //store the set number in the
vertex
                                    FSetNum[vi] = offset;

vectorOfSetsWithVertices[offset].push_back(&(*vi));
                                }
                            }
                        }
                        //Find the set with the most vertices
                        int maxInSet = 0;
                        int maxOffset = 0;
                        int secondBiggest = 0;
                        int secondBiggestOffset = 0;
                        for(int pos=0; pos <
vectorOfSetsWithVertices.size(); pos++)
                        {
                            if(vectorOfSetsWithVertices[pos].size() >
maxInSet) {
                                secondBiggest = maxInSet;
                                secondBiggestOffset = maxOffset;
                                maxInSet =
vectorOfSetsWithVertices[pos].size();
                                maxOffset = pos;
                            }
                        }
                        int numBigSets = 0;
                        int numLittleSets = 0;
                        for(int pos=0; pos <
vectorOfSetsWithVertices.size(); pos++)
                        {
                            if(pos != maxOffset) {
                                if(vectorOfSetsWithVertices[pos].size()
> (deletePercentOfBiggestSegment * secondBiggest)){
                                    numBigSets++;
                                }
                                else{
                                    numLittleSets++;
```

```
                                }
                            }
                        }
                        totalNumBigSets = numBigSets;
                        totalNumLittleSets = numLittleSets;
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                (*vi).C() = Color4b(Color4b::Blue);
                            }
                            //if it is not in set 'F', then find what
set it belongs to
                            if(setFBV[vi] != 'F') {
                                if(FSetNum[vi] == maxOffset) {
                                    (*vi).C() =
Color4b(Color4b::White);
                                }
                                else {
                                    //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                                    int ScatterSize =
vectorOfSetsWithVertices.size();
                                    Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                                    (*vi).C()=BaseColor;
                                }
                            }
                        }
                        delete ptrDsetSegment;
                    }//show segments
                    if(removeLines) {
                        //Check if color same on both sides of F line
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);

                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                Color4b color1;
                                Color4b color2;
                                Color4b color3;
                                Color4b blueColor =
Color4b(Color4b::Blue);
                                Color4b tempColor;
                                int colorCounter = 0;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if (i == 0) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        color1 =
(*tempVertexPointer).C();
                                        colorCounter++;
```

```
                                    }
                                    if(i > 0 && i <
OneRing.allV.size()) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        tempColor =
(*tempVertexPointer).C();
                                        if(colorCounter == 1) {
                                            if(tempColor != color1) {
                                                color2 = tempColor;
                                                colorCounter++;
                                            }
                                        }
                                        if(colorCounter == 2) {
                                            if(tempColor != color1 &&
tempColor != color2) {
                                                color3 = tempColor;
                                                colorCounter++;
                                            }
                                        }
                                    }
                                }
                                //delete it from feature
                                if(colorCounter == 2 || colorCounter ==
1){
                                    if(colorCounter == 2){
                                        //delete from feature if
surrounded by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                        if(color2 != blueColor) {
                                            (*vi).C() = color2;
                                        }
                                    }
                                    if(colorCounter == 1) {
                                        //delete from feature if
surrounded by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                    }
                                }
                            }
                        }//check if color on both sides of line same
                    }//remove lines
                } while (totalNumBigSets == numTeeth);//while
                epsilon = epsilon + .01;
                /* ******************  Calculate curvature:
curvature.h computes the discrete gaussian curvature.
         <vcg/complex/algorithms/update/curvature.h>
         For further details, please, refer to:
         Discrete Differential-Geometry Operators for Triangulated 2-
Manifolds Mark Meyer,
```

```
        Mathieu Desbrun, Peter Schroder, Alan H. Barr VisMath '02,
Berlin </em>*/
                // results stored in (*vi).Kh() (mean) and (*vi).Kg()
(guassian)
                tri::UpdateCurvature<CMeshO>::MeanAndGaussian(m.cm);
                // ********************  Put the curvature in the
quality: <vcg/complex/algorithms/update/quality.h>

tri::UpdateQuality<CMeshO>::VertexFromMeanCurvature(m.cm);
                //****************** ComputePerVertexQualityMinMax 15
and 85 percent
                std::pair<float,float> minmax =
std::make_pair(std::numeric_limits<float>::max(),-
std::numeric_limits<float>::max());
                CMeshO::VertexIterator vi;
                std::vector<float> QV;
                QV.reserve(m.cm.vn);
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                    if(!(*vi).IsD()) QV.push_back((*vi).Q());
                //bottom 15% maps to -1

std::nth_element(QV.begin(),QV.begin()+15*(m.cm.vn/100),QV.end());
                float newmin=*(QV.begin()+m.cm.vn/100);
                //top 15% maps to 1
                std::nth_element(QV.begin(),QV.begin()+m.cm.vn-
15*(m.cm.vn/100),QV.end());
                float newmax=*(QV.begin()+m.cm.vn-m.cm.vn/100);
                minmax.first = newmin;
                minmax.second = newmax;
                // ***************************
MapCurvatureRangetoOneNegOne
                float B = 8.0;
                float min = minmax.first;
                float max = minmax.second;
                float upperBound = 0.0;
                if(min < 0)
                    min = min * -1;
                if(min > max)
                    upperBound = min;
                else
                    upperBound = max;
                float K = 0.5 * log10((1 + ((pow(2.0,B)-2)/(pow(2.0,B)-
1))) / ((1 - ((pow(2.0,B)-2)/(pow(2.0,B)-1)))));
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                    if(!(*vi).IsD())
                    {
                        float remapped = tanh((*vi).Q() * (K /
upperBound));
                        (*vi).Q() = remapped;
                        //Log( "Value to be remapped: %f   remapped:
%f\n", (*vi).Q(), remapped );
                        // Log( "inside mapping");
                    }
                vcg::DisjointSet<CVertexO>* ptrDset = new
vcg::DisjointSet<CVertexO>();
```

```
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //all vertices start as set V for Vertex and
unmarked
                    setFBV[vi] = 'V';
                    marked[vi] = 0;
                    disc[vi] = 0;
                    center[vi] = 0;
                    complex[vi] = 0;
                    newF[vi] = 0;
                    //all vertices start as -1 FSetNum
                    FSetNum[vi] = -1;
                    //none have been visited to connect C points, so
set CpointOwner to -1
                    CpointOwner[vi] = -1;
                    //color it White initially
                    (*vi).C() = Color4b(Color4b::White);
                    curvature[vi] = (*vi).Q();
                    //feature region, less than defined min curvature
                    if(curvature[vi] < epsilon) {
                        //in set F for feature
                        setFBV[vi] = 'F';
                    }
                }
                // apply morphological opperators dilate and then erode
****************************
                if(morphologicalOperations){
                    //dilate 1-ring neighbor
                    for(int k = 0; k < 1; k++){
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] != 'V') {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, mark the neighbor to be one
                                    //if it is negative curvature
                                    if(setFBV[&(*tempVertexPointer)] ==
'V') {
                                        //mark it to be 'N'
                                        marked[&(*tempVertexPointer)] =
1;
                                    }
                                }
                            }
                        }
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it was marked, change it
                            if(marked[vi] == 1){
```

```
                                setFBV[vi] = 'F';
                                marked[vi] = 0;
                            }
                        }
                    }//grow once
                    //erode 1-ring neighbor
                    for(int k = 0; k < 1; k++){
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] != 'V') {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                bool neighborsAllF = true;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, we will erode this vertex
                                    if(setFBV[&(*tempVertexPointer)] ==
'V') {
                                        neighborsAllF = false;
                                    }
                                }
                                if(neighborsAllF){
                                    //mark to not erode this vertex
this round
                                    marked[vi] = 1;
                                }
                            }
                        }
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it was marked to not erode, keep it
                            if(marked[vi] == 1){
                                //setFBV[vi] = 'F';
                                marked[vi] = 0;
                            }
                            else{
                                setFBV[vi] = 'V';
                            }
                        }
                    }//erode once
                }
                //Apply Skeletonize operator
                if(skeletonize){
                    int debugCounter = 1;
                    bool changes = false;
                    //if still changes loop through and skeletonize
                    do {
                        changes = false;
                        //go through the vertices and if 'F' check for
centers and discs
```

```
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                //check for centers and discs
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                bool neighborsAllF = true;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, vi not a center
                                    if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                        neighborsAllF = false;
                                    }
                                }
                                if(neighborsAllF){
                                    //mark as a center
                                    center[vi] = 1;
                                    //mark all neighbors as disc
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                        disc[&(*tempVertexPointer)] =
1;
                                    }
                                }
                            }//in set F
                        }//go through vertices
                        //go through vertices again, if not a center
and is a disc, check complexity and delete if not complex
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(center[vi] == 0 && disc[vi] == 1){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
```

```
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //delete from feature
                                    setFBV[vi] = 'V';
                                    center[vi] = 0;
                                    disc[vi] = 0;
                                    changes = true;
                                }
                            }//not a center, yes a disc
                        }//go through vertices
                        //******************************** reset disc
and center data for next iteration
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                center[vi] = 0;
                                disc[vi] = 0;
                            }
                        }
                        /* if(debugCounter == 1){
                      changes = false;
                   }*/
                    } while (changes == true);//loop if changes
                }//skeletonize operator
                //prune
                if(prune){
                    //prune a certain number of times
                    for(int pruneIter = 0; pruneIter < pruneLength;
pruneIter++){
                        int pruned = 0;
                        //go through the vertices and if 'F' check
complexity
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F'){
                                //check complexity
                                (*vi).SetV();
```

```
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //prune from feature
                                    setFBV[vi] = 'V';
                                    pruned++;
                                }
                            }//checking current vertex
                        }//go through vertices
                    }//number of times we prune
                }//prune before
                //****************************** Initialize the
Disjoint sets of 'F' and delete any that are small
*****************************
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if(setFBV[vi] == 'F'){
                        //put the vertex in the set D
                        ptrDset->MakeSet(&(*vi));
                    }
                }
                /*
            if(showMinAndMaxFeature || onlyShowFeature){
```

```
                //go through the vertices and color them according to
sets
                for(vi=m.cm.vert.begin();vi!=m.cm.vert.end();++vi) {
                    if(setFBV[&(*vi)] == 'F')
                        (*vi).C() = Color4b(Color4b::Blue);
                    if(setFBV[&(*vi)] == 'N')
                        (*vi).C() = Color4b(Color4b::Green);
                      //more than defined maximum curvature
                   // if(!onlyShowFeature) {
                     //   if(curvature[vi] > eta)
                       //    (*vi).C() = Color4b(Color4b::Red);
                   // }
                }
            }
        */
                //print number of sets
                //ptrDset->printNumberOfSets();
                //Go through vertices again--if in set 'F',
******************************
                //then find 1-Ring of vertices and merge sets if
orginal vertex is 'F'
                //and new 1-ring vertex is 'F', define set 'B'
                int blueCounter = 0;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is a feature, then find 1-ring and merge
sets
                    if(setFBV[vi] == 'F') {
                        blueCounter++;
                        vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                        OneRing.insertAndFlag1Ring(&(*vi));
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                            //if the neighbor of the current vertex is
a feature then merge sets
                            if(setFBV[&(*tempVertexPointer)] == 'F') {
                                //if they are not already in the same
set, merge them
                                if(ptrDset->FindSet(&(*vi)) != ptrDset-
>FindSet(tempVertexPointer)){
                                    ptrDset-
>Union(&(*vi),tempVertexPointer);
                                }
                            }
                            else if(setFBV[&(*tempVertexPointer)] ==
'V'){
                                //if the neighbor of the blue vertex is
just a vertex it is on the border
                                //we do this after we delete the small
sets of 'F'
                                //setFBV[&(*tempVertexPointer)] = 'B';
                            }
                        }
                    }
                }
```

```
                //Find out how many distinct sets there are
****************************
                int setCounter = 0;
                vector <CVertexO*> parentVertexofSet;
                typedef  std::vector < std::vector <CVertexO*> >
matrix;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is in set 'F', then find what set it
belongs to
                    if(setFBV[vi] == 'F') {
                        if (ptrDset->FindSet(&(*vi)) == &(*vi)){
                            //parent of a new set
                            parentVertexofSet.push_back(&(*vi));
                            setCounter++;
                        }
                    }
                }
                matrix allSetsWithVertices(setCounter,
std::vector<CVertexO*>(0));
                // ****************************  Create a vector of
vectors containing our sets / vertices
                vector<CVertexO*>::iterator itVect;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is in set 'F', then find what set it
belongs to
                    if(setFBV[vi] == 'F') {
                        //find offset in parent vector--that is the
vertex set number
                        itVect = find(parentVertexofSet.begin(),
parentVertexofSet.end(), ptrDset->FindSet(&(*vi)));
                        if( itVect != parentVertexofSet.end() ) {
                            int offset =
std::distance(parentVertexofSet.begin(), itVect);
                            //store the set number in the vertex
                            FSetNum[vi] = offset;

allSetsWithVertices[offset].push_back(&(*vi));
                        }
                    }
                }
                //delete the disjoint set
                delete ptrDset;
                //print out the stored set data
                /*
          for(int pos=0; pos < allSetsWithVertices.size(); pos++)
          {
          }*/
                //delete sets of D if the number of vertices is less
than .01 D
                for(int pos=0; pos < allSetsWithVertices.size(); pos++)
                {
                    if(allSetsWithVertices[pos].size() <=
deletePercentOfF*blueCounter) {
                        for(int i=0; i <
allSetsWithVertices[pos].size(); i++) {
```

```
                            (*(allSetsWithVertices[pos][i])).C() =
Color4b(Color4b::White);
                            setFBV[(*(allSetsWithVertices[pos][i]))] =
'V';
                        }
                    }
                }
                //************************ Color the different
segments 1st Time and then delete F lines that have same color on both
sides
                //** This clears up a lot of the extraneous connecting
points before we close gaps.
                if(showSegments1st){
                    //***** New DisjointSet
                    vcg::DisjointSet<CVertexO>* DisjointSetsToColor =
new vcg::DisjointSet<CVertexO>();
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //Every vertex not in set 'F' starts as
disjoint set
                        if(setFBV[vi] != 'F') {
                            //put the vertex in the set D
                            DisjointSetsToColor->MakeSet(&(*vi));
                        }
                    }
                    //***** Merge neighboring Sets
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not a feature, then find 1-ring and
merge sets
                        if(setFBV[vi] != 'F') {
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature then merge sets
                                if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                    //if they are not already in the
same set, merge them
                                    if(DisjointSetsToColor-
>FindSet(&(*vi)) != DisjointSetsToColor->FindSet(tempVertexPointer)){
                                        DisjointSetsToColor-
>Union(&(*vi),tempVertexPointer);
                                    }
                                }
                            }
                        }
                    }
                    //Find out how many distinct segment sets there are
*****************************
                    int segmentSetCounter = 0;
                    vector <CVertexO*> parentVertexofDisjointSets;
```

```
                    //***** If a vertex is the parent of a DisjointSet,
it is a new Segment
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if (DisjointSetsToColor->FindSet(&(*vi)) ==
&(*vi)){
                                //parent of a new set

parentVertexofDisjointSets.push_back(&(*vi));
                                segmentSetCounter++;
                            }
                        }
                    }
                    //******* find biggest set
                    typedef  std::vector < std::vector <CVertexO*> >
crmatrix;
                    crmatrix
vectorOfDisjointSetsWithVertices(segmentSetCounter,
std::vector<CVertexO*>(0));
                    vector<CVertexO*>::iterator itVect;
                    // ******************************  Create a vector
of vectors containing our sets / vertices
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is in set 'F', then find what set it
belongs to
                        if(setFBV[vi] != 'F') {
                            //find offset in parent vector--that is the
vertex set number
                            itVect =
find(parentVertexofDisjointSets.begin(),
parentVertexofDisjointSets.end(), DisjointSetsToColor-
>FindSet(&(*vi)));
                            if( itVect !=
parentVertexofDisjointSets.end() ) {
                                int offset =
std::distance(parentVertexofDisjointSets.begin(), itVect);
                                //store the set number in the vertex
                                FSetNum[vi] = offset;

vectorOfDisjointSetsWithVertices[offset].push_back(&(*vi));
                            }
                        }
                    }
                    //*********** For each set, sum the positive
curvature and store in new vector
                    vector<float>
vectorOfSetPosCurvatureSums(vectorOfDisjointSetsWithVertices.size(),0);
                    for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                    {
                        float tempSumOfCurvature = 0.0;
                        for(int numInSet=0; numInSet <
vectorOfDisjointSetsWithVertices[pos].size(); numInSet++){
```

```
                            CVertexO* tempVertex =
vectorOfDisjointSetsWithVertices[pos][numInSet];
                            float tempCurvature =
curvature[tempVertex];
                            //if it's positive, add it up
                            if(tempCurvature > 0.0) {
                                tempSumOfCurvature = tempSumOfCurvature
+ tempCurvature;
                            }
                        }
                        vectorOfSetPosCurvatureSums[pos] =
tempSumOfCurvature;
                    }
                    //Find the set with the most vertices
                    int maxInSet = 0;
                    int maxOffset = 0;
                    int secondBiggest = 0;
                    int secondBiggestOffset = 0;
                    for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                    {
                        if(vectorOfDisjointSetsWithVertices[pos].size()
> maxInSet) {
                            secondBiggest = maxInSet;
                            secondBiggestOffset = maxOffset;
                            maxInSet =
vectorOfDisjointSetsWithVertices[pos].size();
                            maxOffset = pos;
                        }
                    }
                    int numBigSets = 0;
                    int numLittleSets = 0;
                    for(int pos=0; pos <
vectorOfDisjointSetsWithVertices.size(); pos++)
                    {
                        if(pos != maxOffset) {

if(vectorOfDisjointSetsWithVertices[pos].size() >
(deletePercentOfBiggestSegment * secondBiggest)){
                                numBigSets++;
                            }
                            else{
                                numLittleSets++;
                            }
                        }
                    }
                    //color the different sets
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).C() = Color4b(Color4b::Blue);
                        }
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if(FSetNum[vi] == maxOffset) {
                                (*vi).C() = Color4b(Color4b::White);
```

```
                            }
                            else {
                                //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                                int ScatterSize =
vectorOfDisjointSetsWithVertices.size();
                                Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                                (*vi).C()=BaseColor;
                            }
                        }
                    }
                    delete DisjointSetsToColor;
                }//show segments
                if(removeLines1st) {
                    //Check if color same on both sides of F line
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            Color4b color1;
                            Color4b color2;
                            Color4b color3;
                            Color4b whiteColor =
Color4b(Color4b::White);
                            Color4b blueColor = Color4b(Color4b::Blue);
                            Color4b tempColor;
                            int colorCounter = 0;
                            CVertexO* tempVertexPointer;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                if (i == 0) {
                                    tempVertexPointer =
OneRing.allV.at(i);

                                    color1 = (*tempVertexPointer).C();
                                    colorCounter++;
                                }
                                if(i > 0 && i < OneRing.allV.size()) {
                                    tempVertexPointer =
OneRing.allV.at(i);
                                    tempColor =
(*tempVertexPointer).C();
                                    if(colorCounter == 1) {
                                        if(tempColor != color1) {
                                            color2 = tempColor;
                                            colorCounter++;
                                        }
                                    }
                                    if(colorCounter == 2) {
                                        if(tempColor != color1 &&
tempColor != color2) {
                                            color3 = tempColor;
```

```
                                        colorCounter++;
                                    }
                                }
                            }
                        }
                        //delete it from feature
                        if(colorCounter == 2 || colorCounter == 1){
                            if(colorCounter == 2){
                                if(color1 != whiteColor && color2
!= whiteColor) {
                                    //delete from feature if
surrounded by a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                    if(color2 != blueColor) {
                                        (*vi).C() = color2;
                                    }
                                }
                            }
                            if(colorCounter == 1) {
                                if(color1 != whiteColor) {
                                    //delete from feature if
surrounded by a color besides white
                                    setFBV[vi] = 'V';
                                    if(color1 != blueColor) {
                                        (*vi).C() = color1;
                                    }
                                }
                            }
                        }
                    }
                }//check if color on both sides of line same
            }//remove lines
            //******************************** Mark the borders
after deleting features
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                //if it is a feature, then find 1-ring and mark set
'B'
                if(setFBV[vi] == 'F') {
                    vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*vi));
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                        if(setFBV[&(*tempVertexPointer)] == 'V'){
                            //if the neighbor of the 'F' vertex is
just a vertex it is on the border
                            setFBV[&(*tempVertexPointer)] = 'B';
                        }
                    }
                }
            }
            // ****************************  Estimate distance
from feature region
```

```
                tri::Geo<CMeshO> g;
                //create a vector of starting vertices in set 'B'
(border of feature)
                std::vector<CVertexO*> fro;
                bool ret;
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi)
                    if( setFBV[vi] == 'B')
                        fro.push_back(&(*vi));
                if(!fro.empty()) {
                    ret = g.DistanceFromFeature(m.cm, fro,
distanceConstraint);
                }
                float maxdist = distanceConstraint;
                float mindist = 0;
                //the distance is now stored in the quality, transfer
the quality to the distance
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    if((*vi).Q() < distanceConstraint) {
                        if( setFBV[vi] == 'F') {
                            disth[vi] = -((*vi).Q());
                        }
                        if( setFBV[vi] == 'B') {
                            disth[vi] = 0;
                        }
                        if( setFBV[vi] == 'V') {
                            disth[vi] = (*vi).Q();
                        }
                    }
                    else {
                        disth[vi] = std::numeric_limits<float>::max();
                    }
                }
                //filter the distances
                if(filterOn){
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //new distance to vertex is average of
neighborhood distances
                        if( disth[vi] < maxdist) {
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            float numberOfNeigbors = 0.0;
                            float sumOfDistances = 0.0;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                float tempDist =
disth[(*tempVertexPointer)];
                                if(tempDist !=
std::numeric_limits<float>::max()) {
                                    sumOfDistances = sumOfDistances +
tempDist;
                                    numberOfNeigbors++;
```

```
                                }
                            }
                            float inverseNumNeighbors = 1.0 /
numberOfNeigbors;
                            disth[vi] = inverseNumNeighbors *
sumOfDistances;
                        }
                    }
                }
                /****************************** Connecting points by
angle ***************************/
                vector <CVertexO*> vectCPoints;
                int CPointCounter = 0;
                int vertexCounter = 0;
                if(findConnectingPoints) {
                    //Find Connecting Points
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if( setFBV[vi] == 'F' ) {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            char currentSet = 'a';
                            char initialSet = 'a';
                            int numberOfChanges = 0;
                            float numberOfNeighbors = 0.0;
                            float sumOfDivergence = 0.0;
                            float angle = 0.0;
                            bool addAngle = false;
                            CVertexO* firstVertex;
                            CVertexO* previousVertex;
                            char previousSet = 'a';
                            char firstSet = 'a';
                            int borderNeighborCounter = 0;
                            int featureNeighborCounter = 0;
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                char tempSetOfThisNeighbor =
(setFBV[&(*tempVertexPointer)]);
                                if(tempSetOfThisNeighbor == 'F') {
                                    featureNeighborCounter++;
                                    //don't add angle, we are still in
'F'
                                    addAngle = false;
                                }
                                if(tempSetOfThisNeighbor == 'B'){
                                    borderNeighborCounter++;
                                    //add angle, we are still in 'F'
                                    addAngle = true;
                                }
                                if(i == 0){
                                    //initialize the starting set
```

```
                                    currentSet =
setFBV[&(*tempVertexPointer)];
                                    initialSet =
setFBV[&(*tempVertexPointer)];
                                    firstVertex = tempVertexPointer;
                                    firstSet =
setFBV[&(*tempVertexPointer)];
                                    previousVertex = tempVertexPointer;
                                    previousSet =
setFBV[&(*tempVertexPointer)];
                                }
                                if(tempSetOfThisNeighbor !=
currentSet){
                                    //changed sets, update the
numberOfChanges
                                    currentSet =
setFBV[&(*tempVertexPointer)];
                                    numberOfChanges++;
                                }
                                //check if last point is in same set as
first point
                                if(i == (OneRing.allV.size() - 1)) {
                                    if(tempSetOfThisNeighbor !=
initialSet){
                                        numberOfChanges++;
                                    }
                                    if(firstSet == 'F' && currentSet ==
'F'){
                                        //don't add the angle
                                    }
                                    else{
                                        //calculate and add the angle
between tempVertexPointer and firstVertex
                                        vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                        //vector from gradient field of
w
                                        vcg::Point3f v2 =
(*firstVertex).P() - (*vi).P();
                                        double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                        double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                        v1 = v1 * (1.0/v1_magnitude);
                                        v2 = v2 * (1.0/v2_magnitude);
                                        double cosTheta = 0.0;
                                        if(v1_magnitude*v2_magnitude ==
0) {
                                            cosTheta = 0.0;
                                        }
                                        else{
                                            cosTheta = v1.X()*v2.X() +
v1.Y()*v2.Y() + v1.Z()*v2.Z();
                                        }
                                        float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
```

```
                                        angle = angle +
tempAngleinDegrees;
                                    }
                                }
                                if(i > 0){
                                    if(previousSet == 'F' && currentSet
== 'F'){
                                        //don't add the angle
                                    }
                                    else{
                                        //calculate and add the angle
between previousVertex and tempVertexPointer
                                        vcg::Point3f v1 =
(*tempVertexPointer).P() - (*vi).P();
                                        //vector from gradient field of
w
                                        vcg::Point3f v2 =
(*previousVertex).P() - (*vi).P();
                                        double v1_magnitude =
sqrt(v1.X()*v1.X() + v1.Y()*v1.Y() + v1.Z()*v1.Z());
                                        double v2_magnitude =
sqrt(v2.X()*v2.X() + v2.Y()*v2.Y() + v2.Z()*v2.Z());
                                        v1 = v1 * (1.0/v1_magnitude);
                                        v2 = v2 * (1.0/v2_magnitude);
                                        double cosTheta = 0.0;
                                        if(v1_magnitude*v2_magnitude ==
0) {
                                            cosTheta = 0.0;
                                        }
                                        else{
                                            cosTheta = v1.X()*v2.X() +
v1.Y()*v2.Y() + v1.Z()*v2.Z();
                                        }
                                        float tempAngleinDegrees =
acos(cosTheta) * 180.0 / PI;
                                        angle = angle +
tempAngleinDegrees;
                                    }
                                    //update the pointers
                                    previousVertex = tempVertexPointer;
                                    previousSet =
setFBV[&(*tempVertexPointer)];
                                }
                                float tempDivergence =
divfval[&(*tempVertexPointer)];
                                sumOfDivergence = sumOfDivergence +
tempDivergence;
                                numberOfNeighbors++;
                            }
                            //Candidate connecting point
                            if(numberOfChanges <= 2){
                                //not element of Fp, may be connecting
point
                                if(angle > connectingPointAngle) {
                                    (*vi).C() =
Color4b(Color4b::Yellow);
                                    CpointOwner[vi] = CPointCounter;
```

```
                                    source[vi] = &(*vi);
                                    vectCPoints.push_back(&(*vi));
                                    CPointCounter++;
                                }
                            }
                        }//in set 'F'
                        vertexCounter++;
                    }//go through vertices
                }//find connecting points and color them yellow
                //Find Mid Points
                std::vector<CMeshO::VertexPointer>  midPointVector;
                if(findMidPoints){
                    //************************************  Find
connecting paths from connecting points
                    //setup the seed vector of connecting points
                    std::vector<VertDist> seedVec;
                    std::vector<CVertexO*>::iterator fi;
                    for( fi  = vectCPoints.begin(); fi !=
vectCPoints.end() ; ++fi)
                    {
                        seedVec.push_back(VertDist(*fi,0.0));
                    }
                    std::vector<VertDist> frontier;
                    CMeshO::VertexPointer curr;
                    CMeshO::ScalarType unreached  =
std::numeric_limits<CMeshO::ScalarType>::max();
                    CMeshO::VertexPointer pw;
                    TempDataType TD(m.cm.vert, unreached);
                    std::vector <VertDist >::iterator ifr;
                    for(ifr = seedVec.begin(); ifr != seedVec.end();
++ifr){
                        TD[(*ifr).v].d = 0.0;
                        (*ifr).d = 0.0;
                        TD[(*ifr).v].source  = (*ifr).v;
                        frontier.push_back(VertDist((*ifr).v,0.0));
                    }
                    // initialize Heap
                    make_heap(frontier.begin(),frontier.end(),pred());
                    std::vector<int> doneConnectingPoints;
                    CMeshO::ScalarType curr_d,d_curr = 0.0,d_heap;
                    CMeshO::VertexPointer curr_s = NULL;
                    CMeshO::PerVertexAttributeHandle
<CMeshO::VertexPointer> * vertSource = NULL;
                    CMeshO::ScalarType max_distance=0.0;
                    std::vector<VertDist >:: iterator iv;
                    int connectionCounter = 0;
                    while(!frontier.empty() && max_distance < maxdist)
                    {

pop_heap(frontier.begin(),frontier.end(),pred());
                        curr = (frontier.back()).v;
                        int tempCpointOwner = CpointOwner[curr];
                        int tempFSetNum = FSetNum[curr];
                        curr_s = TD[curr].source;
                        if(vertSource!=NULL)
                            (*vertSource)[curr] = curr_s;
                        d_heap = (frontier.back()).d;
```

```
                        frontier.pop_back();
                        assert(TD[curr].d <= d_heap);
                        assert(curr_s != NULL);
                        if(TD[curr].d < d_heap )// a vertex whose
distance has been improved after it was inserted in the queue
                            continue;
                        assert(TD[curr].d == d_heap);
                        d_curr =  TD[curr].d;
                        if(d_curr > max_distance) {
                            max_distance = d_curr;
                        }
                        //check the vertices around the current point
                        (*curr).SetV();
                        vcg::tri::Nring<CMeshO> OneRing(&(*curr),
&m.cm);
                        OneRing.insertAndFlag1Ring(&(*curr));
                        (*curr).ClearV();
                        for (int i = 0; i < OneRing.allV.size(); i++){
                            pw = OneRing.allV.at(i);
                            //just find the shortest distance between
points
                            curr_d = d_curr + vcg::Distance(pw-
>cP(),curr->cP());
                            //check if we are still searching from this
connecting point
                            std::vector<int>::iterator it;
                            it =
std::find(doneConnectingPoints.begin(), doneConnectingPoints.end(),
CpointOwner[curr]);
                            //we are still searching from this
connecting point
                            if (it == doneConnectingPoints.end()){
                                //This point has been explored by a
different Cpoint before
                                //deleted the bit about not connecting
to your own set: && FSetNum[pw] != FSetNum[curr]
                                if(CpointOwner[pw] != CpointOwner[curr]
&& CpointOwner[pw] != -1 && setFBV[&(*pw)] != 'F'){
                                    //This point has been explored
before, we may have a connection
                                    //(check if the CpointOwner is
active or already connected up with someone else
                                    //std::vector<int>::iterator chk;
                                    //chk =
std::find(doneConnectingPoints.begin(), doneConnectingPoints.end(),
CpointOwner[pw]);
                                    //we are still searching from the
CpointSource of the point we found
                                    //if(chk ==
doneConnectingPoints.end()) {
                                    (*pw).C() =
Color4b(Color4b::Magenta);
                                    connectionCounter++;
                                    source1[pw] = curr;
                                    //store the midpoint so we can make
the connecting path later
                                    midPointVector.push_back(pw);
```

```
                                    //add CpointOwner to vector of
doneConnectingPoints

doneConnectingPoints.push_back(CpointOwner[curr]);

doneConnectingPoints.push_back(CpointOwner[pw]);
                                    // }
                                    /* else{
                                }*/
                                }
                                //deleted  && curvature[pw] <
maxCurvatureInPath
                                else if(TD[(pw)].d > curr_d && curr_d <
maxdist && setFBV[&(*pw)] != 'F'){
                                    //This point has not been explored
before, keep looking
                                    //update source, Fsetnum,
CpointOwner
                                    CpointOwner[pw] = tempCpointOwner;
                                    source[pw] = curr;
                                    FSetNum[pw] = tempFSetNum;
                                    TD[(pw)].d = curr_d;
                                    TD[pw].source = curr;

frontier.push_back(VertDist(pw,curr_d));

push_heap(frontier.begin(),frontier.end(),pred());
                                }
                            }
                            else {
                                //not searching from this connecting
point anymore
                            }
                        }
                    }// end while
                }//find mid points and color magenta
                if(findMidPoints && findPathstoConnect) {
                    //********************** Make the connecting path
                    vector<CMeshO::VertexPointer>::iterator iter;
                    for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                        //track back source to beginning
                        CMeshO::VertexPointer tempVertexPointer =
*iter;
                        CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Green);
                            // setFBV[tempVertexPointer] = 'F';
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                        //track back source1 to beginning
                        tempVertexPointer = *iter;
                        tempVertexSource = source1[tempVertexPointer];
```

```
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Cyan);
                            //setFBV[tempVertexPointer] = 'F';
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                    }//make connecting path
                }//find paths to connect and color green and cyan
                if(addPathstoFeatureRegion){
                    //********************** Make the connecting path
                    vector<CMeshO::VertexPointer>::iterator iter;
                    for ( iter = midPointVector.begin(); iter !=
midPointVector.end(); ++iter ){
                        //track back source to beginning
                        CMeshO::VertexPointer tempVertexPointer =
*iter;
                        CMeshO::VertexPointer tempVertexSource =
source[tempVertexPointer];
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                            setFBV[tempVertexPointer] = 'F';
                            newF[tempVertexPointer] = 1;
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                        //track back source1 to beginning
                        tempVertexPointer = *iter;
                        tempVertexSource = source1[tempVertexPointer];
                        while(tempVertexPointer != tempVertexSource){
                            (*tempVertexPointer).C() =
Color4b(Color4b::Blue);
                            setFBV[tempVertexPointer] = 'F';
                            newF[tempVertexPointer] = 1;
                            tempVertexPointer = tempVertexSource;
                            tempVertexSource =
source[tempVertexPointer];
                        }
                    }
                }//add paths to feature region
                //******************************** get ready to
skeletonize and prune by changing sets to either F or V
                for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                    //if it is not a feature
                    if(setFBV[vi] != 'F') {
                        setFBV[vi] = 'V';
                    }
                }
                //***************************** Skeletonize and prune
                //Apply Skeletonize operator
                if(skeletonizeAfter){
                    bool changes = false;
                    //if still changes loop through and skeletonize
```

```
                    do {
                        changes = false;
                        //go through the vertices and if 'F' check for
centers and discs
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F') {
                                //check for centers and discs
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                bool neighborsAllF = true;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                    //if the neighbor of the current
vertex is not a feature, vi not a center
                                    if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                        neighborsAllF = false;
                                    }
                                }
                                if(neighborsAllF){
                                    //mark as a center
                                    center[vi] = 1;
                                    //mark all neighbors as disc
                                    for (int i = 0; i <
OneRing.allV.size(); i++){
                                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                        disc[&(*tempVertexPointer)] =
1;
                                    }
                                }
                            }//in set F
                        }//go through vertices
                        //go through vertices again, if not a center
and is a disc, check complexity and delete if not complex
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F' && center[vi] == 0 &&
disc[vi] == 1){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
```

```
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //delete from feature
                                    setFBV[vi] = 'V';
                                    center[vi] = 0;
                                    disc[vi] = 0;
                                    changes = true;
                                }
                            }//not a center, yes a disc
                        }//go through vertices
                        //******************************** reset disc
and center data for next iteration
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            //if it is a feature, reset center and disc
data
                            if(setFBV[vi] == 'F') {
                                center[vi] = 0;
                                disc[vi] = 0;
                            }
                        }
                    } while (changes == true);//loop if changes
                }//skeletonize operator
                //prune
                if(pruneAfter){
                    //pune a certain number of times
                    for(int pruneIter = 0; pruneIter <
pruneAfterLength; pruneIter++){
                        int pruned = 0;
                        vertexCounter = 0;
                        //go through the vertices and if 'F' check
complexity
```

```
                        for(vi = m.cm.vert.begin(); vi !=
m.cm.vert.end(); ++vi) {
                            if(setFBV[vi] == 'F'){
                                //check complexity
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                                OneRing.insertAndFlag1Ring(&(*vi));
                                (*vi).ClearV();
                                int numberOfChanges = 0;
                                char iVertex;
                                char iPlus1Vertex;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i <
OneRing.allV.size(); i++){
                                    if(i < (OneRing.allV.size() - 1)) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(i+1);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    else {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        iVertex =
(setFBV[&(*tempVertexPointer)]);
                                        tempVertexPointer =
OneRing.allV.at(0);
                                        iPlus1Vertex =
(setFBV[&(*tempVertexPointer)]);
                                    }
                                    //change noted
                                    if(iVertex != iPlus1Vertex){
                                        numberOfChanges++;
                                    }
                                }
                                //not complex, delete it from feature
                                if(numberOfChanges < 4){
                                    //prune from feature
                                    setFBV[vi] = 'V';
                                    pruned++;
                                }
                            }//set
                            vertexCounter++;
                        }//go through vertices
                    }//number of times we prune
                }//prune after
                /*
                if(showFinalFeature) {
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                          (*vi).C() = Color4b(Color4b::Blue);
```

```
                        }
                        if(setFBV[vi] != 'F') {
                          (*vi).C() = Color4b(Color4b::White);
                        }
                    }
                }
                */
                if(showSegments){
                    //************************ Color the different
segments 2nd Time
                    //***** New DisjointSet
                    vcg::DisjointSet<CVertexO>* ptrDsetSegment = new
vcg::DisjointSet<CVertexO>();
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //Every vertex not in set 'F' starts as
disjoint set
                        if(setFBV[vi] != 'F') {
                            //put the vertex in the set D
                            ptrDsetSegment->MakeSet(&(*vi));
                        }
                    }
                    //***** Merge neighboring Sets
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not a feature, then find 1-ring and
merge sets
                        if(setFBV[vi] != 'F') {
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);
                            OneRing.insertAndFlag1Ring(&(*vi));
                            for (int i = 0; i < OneRing.allV.size();
i++){
                                CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                                //if the neighbor of the current vertex
is not a feature then merge sets
                                if(setFBV[&(*tempVertexPointer)] !=
'F') {
                                    //if they are not already in the
same set, merge them
                                    if(ptrDsetSegment->FindSet(&(*vi))
!= ptrDsetSegment->FindSet(tempVertexPointer)){
                                        ptrDsetSegment-
>Union(&(*vi),tempVertexPointer);
                                    }
                                }
                            }
                        }
                    }
                    //Find out how many distinct segment sets there are
*****************************
                    int segmentSetCounter = 0;
                    vector <CVertexO*> parentVertexofSegmentSet;
                    //***** If a vertex is the parent of a DisjointSet,
it is a new Segment
```

```
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if (ptrDsetSegment->FindSet(&(*vi)) ==
&(*vi)){
                                //parent of a new set

parentVertexofSegmentSet.push_back(&(*vi));
                                segmentSetCounter++;
                            }
                        }
                    }
                    //******* find biggest set
                    typedef  std::vector < std::vector <CVertexO*> >
crmatrix;
                    crmatrix
vectorOfSetsWithVertices(segmentSetCounter, std::vector<CVertexO*>(0));
                    // ******************************  Create a vector
of vectors containing our sets / vertices
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        //if it is in set 'F', then find what set it
belongs to
                        if(setFBV[vi] != 'F') {
                            //find offset in parent vector--that is the
vertex set number
                            itVect =
find(parentVertexofSegmentSet.begin(), parentVertexofSegmentSet.end(),
ptrDsetSegment->FindSet(&(*vi)));
                            if( itVect !=
parentVertexofSegmentSet.end() ) {
                                int offset =
std::distance(parentVertexofSegmentSet.begin(), itVect);
                                //store the set number in the vertex
                                FSetNum[vi] = offset;

vectorOfSetsWithVertices[offset].push_back(&(*vi));
                            }
                        }
                    }
                    //Find the set with the most vertices
                    int maxInSet = 0;
                    int maxOffset = 0;
                    int secondBiggest = 0;
                    int secondBiggestOffset = 0;
                    for(int pos=0; pos <
vectorOfSetsWithVertices.size(); pos++)
                    {
                        if(vectorOfSetsWithVertices[pos].size() >
maxInSet) {
                            secondBiggest = maxInSet;
                            secondBiggestOffset = maxOffset;
                            maxInSet =
vectorOfSetsWithVertices[pos].size();
                            maxOffset = pos;
```

```
                        }
                    }
                    int numBigSets = 0;
                    int numLittleSets = 0;
                    for(int pos=0; pos <
vectorOfSetsWithVertices.size(); pos++)
                    {
                        if(pos != maxOffset) {
                            if(vectorOfSetsWithVertices[pos].size() >
(deletePercentOfBiggestSegment * secondBiggest)){
                                numBigSets++;
                            }
                            else{
                                numLittleSets++;
                            }
                        }
                    }
                    totalNumBigSets = numBigSets;
                    totalNumLittleSets = numLittleSets;
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).C() = Color4b(Color4b::Blue);
                        }
                        //if it is not in set 'F', then find what set
it belongs to
                        if(setFBV[vi] != 'F') {
                            if(FSetNum[vi] == maxOffset) {
                                (*vi).C() = Color4b(Color4b::White);
                            }
                            else {
                                //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                                int ScatterSize =
vectorOfSetsWithVertices.size();
                                Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.3f,.9f);
                                (*vi).C()=BaseColor;
                            }
                        }
                    }
                    delete ptrDsetSegment;
                }//show segments
                if(removeLines) {
                    //Check if color same on both sides of F line
                    for(vi = m.cm.vert.begin(); vi != m.cm.vert.end();
++vi) {
                        if(setFBV[vi] == 'F') {
                            (*vi).SetV();
                            vcg::tri::Nring<CMeshO> OneRing(&(*vi),
&m.cm);

                            OneRing.insertAndFlag1Ring(&(*vi));
                            (*vi).ClearV();
                            Color4b color1;
                            Color4b color2;
                            Color4b color3;
```

```
                                Color4b blueColor = Color4b(Color4b::Blue);
                                Color4b tempColor;
                                int colorCounter = 0;
                                CVertexO* tempVertexPointer;
                                for (int i = 0; i < OneRing.allV.size();
i++){
                                    if (i == 0) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        color1 = (*tempVertexPointer).C();
                                        colorCounter++;
                                    }
                                    if(i > 0 && i < OneRing.allV.size()) {
                                        tempVertexPointer =
OneRing.allV.at(i);
                                        tempColor =
(*tempVertexPointer).C();
                                        if(colorCounter == 1) {
                                            if(tempColor != color1) {
                                                color2 = tempColor;
                                                colorCounter++;
                                            }
                                        }
                                        if(colorCounter == 2) {
                                            if(tempColor != color1 &&
tempColor != color2) {
                                                color3 = tempColor;
                                                colorCounter++;
                                            }
                                        }
                                    }
                                }
                                //delete it from feature
                                if(colorCounter == 2 || colorCounter == 1){
                                    if(colorCounter == 2){
                                        //delete from feature if surrounded
by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                        if(color2 != blueColor) {
                                            (*vi).C() = color2;
                                        }
                                    }
                                    if(colorCounter == 1) {
                                        //delete from feature if surrounded
by a color besides white
                                        setFBV[vi] = 'V';
                                        if(color1 != blueColor) {
                                            (*vi).C() = color1;
                                        }
                                    }
                                }
                            }
                        }//check if color on both sides of line same
                    }//remove lines
```

```
            }//if foundIt == true
        }
        //**********************Redo the disjoint set info and delete
the little ones by removing the biggest connecting line with another
set
        if(deletePercentOfLargestSegment) {
            CMeshO::VertexIterator vi;
            vector<CVertexO*>::iterator itVect;
            //Find the set with the most vertices
            //************************ Color the different segments
after deleting lines in a segment
            vcg::DisjointSet<CVertexO>* SecondDsetSegment = new
vcg::DisjointSet<CVertexO>();
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //Every vertex not in set 'F' starts as disjoint set
                if(setFBV[vi] != 'F') {
                    //put the vertex in the set D
                    SecondDsetSegment->MakeSet(&(*vi));
                }
            }
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is not a feature, then find 1-ring and merge
sets
                if(setFBV[vi] != 'F') {
                    vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*vi));
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                        //if the neighbor of the current vertex is not
a feature then merge sets
                        if(setFBV[&(*tempVertexPointer)] != 'F') {
                            //if they are not already in the same set,
merge them
                            if(SecondDsetSegment->FindSet(&(*vi)) !=
SecondDsetSegment->FindSet(tempVertexPointer)){
                                SecondDsetSegment-
>Union(&(*vi),tempVertexPointer);
                            }
                        }
                    }
                }
            }
            //Find out how many distinct segment sets there are
*****************************
            int SecondsegmentSetCounter = 0;
            vector <CVertexO*> SecondparentVertexofSegmentSet;
            typedef  std::vector < std::vector <CVertexO*> >
colormatrix;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is not in set 'F', then find what set it
belongs to
                if(setFBV[vi] != 'F') {
                    if (SecondDsetSegment->FindSet(&(*vi)) == &(*vi)){
                        //parent of a new set

SecondparentVertexofSegmentSet.push_back(&(*vi));
```

```
                        SecondsegmentSetCounter++;
                    }
                }
            }
            colormatrix
allColorSetsWithVertices(SecondsegmentSetCounter,
std::vector<CVertexO*>(0));
            // ******************************  Create a vector of
vectors containing our sets / vertices
            //vector<CVertexO*>::iterator itVect;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is in set 'F', then find what set it belongs to
                if(setFBV[vi] != 'F') {
                    //find offset in parent vector--that is the vertex
set number
                    itVect =
find(SecondparentVertexofSegmentSet.begin(),
SecondparentVertexofSegmentSet.end(), SecondDsetSegment-
>FindSet(&(*vi)));
                    if( itVect != SecondparentVertexofSegmentSet.end()
) {
                        int offset =
std::distance(SecondparentVertexofSegmentSet.begin(), itVect);
                        //store the set number in the vertex
                        FSetNum[vi] = offset;

allColorSetsWithVertices[offset].push_back(&(*vi));
                    }
                }
            }
            delete SecondDsetSegment;
            //Find the set with the most vertices
            int maxInSet = 0;
            int maxOffset = 0;
            int secondBiggest = 0;
            int secondBiggestOffset = 0;
            for(int pos=0; pos < allColorSetsWithVertices.size();
pos++)
            {
                if(allColorSetsWithVertices[pos].size() > maxInSet){
                    secondBiggest = maxInSet;
                    secondBiggestOffset = maxOffset;
                    maxInSet = allColorSetsWithVertices[pos].size();
                    maxOffset = pos;
                }
            }
            //**********Make a vector of FSetnum for small sets (less
than .01 D)
            vector <int> vectorSmallSetNum;
            vector<int>::iterator itVectNum;
            for(int pos=0; pos < allColorSetsWithVertices.size();
pos++)
            {
                if(allColorSetsWithVertices[pos].size() <=
deletePercentOfBiggestSegment*secondBiggest) {
                    CVertexO* temporaryVertex =
allColorSetsWithVertices[pos][0];
```

```
                    int temporaryNum = FSetNum[temporaryVertex];
                    vectorSmallSetNum.push_back(temporaryNum);
                }
            }
            //Delete the boundary between the small and big sets
(delete the line where they touch the most)
            for(int pos=0; pos < allColorSetsWithVertices.size();
pos++)
            {
                //if it's a small set
                if(allColorSetsWithVertices[pos].size() <=
deletePercentOfBiggestSegment*secondBiggest) {
                    //find out what our setNum is
                    CVertexO* temporaryVertex =
allColorSetsWithVertices[pos][0];
                    int thisSmallSetNum = FSetNum[temporaryVertex];
                    vector <int> vectorNeighborSetNums;
                    //go through every vertex in this small set
                    for(int i=0; i <
allColorSetsWithVertices[pos].size(); i++) {
                        //here is the temp vertex in the small set
                        CVertexO* centerVertex =
allColorSetsWithVertices[pos][i];
                        (*vi).SetV();
                        vcg::tri::Nring<CMeshO>
OneRingAroundVinSmallSet(centerVertex, &m.cm);

OneRingAroundVinSmallSet.insertAndFlag1Ring(centerVertex);
                        (*vi).ClearV();
                        //go through the 1-ring neighborhood of
centerVertex and check for an 'F' to see if you should turn it into a
"V"
                        for (int i = 0; i <
OneRingAroundVinSmallSet.allV.size(); i++){
                            //this is a neighbor of centerVertex in the
small set
                            CVertexO* tempVertexPointer =
OneRingAroundVinSmallSet.allV.at(i);
                            //Check the one-ring of the 'F' and see if
any other small vertices are nearby, if so, delete it
                            if(setFBV[&(*tempVertexPointer)] == 'F') {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO>
OneRingARoundF(tempVertexPointer, &m.cm);

OneRingARoundF.insertAndFlag1Ring(tempVertexPointer);
                                (*vi).ClearV();
                                for (int j = 0; j <
OneRingARoundF.allV.size(); j++){
                                    CVertexO*
tempVertexPointerNeighborOfF = OneRingARoundF.allV.at(j);
                                    //ignore other 'F's--they have a
different FSetNum system AND members of this small set

if(setFBV[&(*tempVertexPointerNeighborOfF)] != 'F' &&
FSetNum[tempVertexPointerNeighborOfF] != thisSmallSetNum) {
```

```
                                        //add the setNum of this
neighbor to the vector of setNums
                                        int tempSetNumOfNeighborOfF =
FSetNum[tempVertexPointerNeighborOfF];

vectorNeighborSetNums.push_back(tempSetNumOfNeighborOfF);
                                    }
                                }
                            }//found a 'F'
                        }
                    }//go through every vertex in this small set
                    //find the Mode (most often occuring) set from the
vectorNeighborSetNums
                    int prev_count = 0;
                    int curr_count = 0;
                    int mode = 0;
                    for (int i=0; i<vectorNeighborSetNums.size(); ++i)
{ // for each element of the vector "numbers"
                        for (int j=0; j<vectorNeighborSetNums.size();
++j) { // compare it to all its other elements
                            if (vectorNeighborSetNums[i] ==
vectorNeighborSetNums[j]) // if 2 elements equal, increment the count
(curr_count)
                                curr_count++;
                        }
                        if (curr_count>prev_count) { // if the current
count greater than previous count
                            mode = vectorNeighborSetNums[i]; // the
mode is set to equal that element of the vector
                            prev_count = curr_count; // set previous
count == current count & repeat above steps
                        }
                        curr_count = 0; // set current count to zero
after finishing the check on each of the elements
                    }
                    /*
                    //print out vector for debugging
                    for(int i = 0; i < vectorNeighborSetNums.size();
i++){
                    }
                    */
                    //Delete an 'F' it is touches a mode set
                    //go through every vertex in this small set
                    for(int i=0; i <
allColorSetsWithVertices[pos].size(); i++) {
                        //here is the temp vertex in the small set
                        CVertexO* centerVertex =
allColorSetsWithVertices[pos][i];
                        (*vi).SetV();
                        vcg::tri::Nring<CMeshO>
OneRingAroundVinSmallSet(centerVertex, &m.cm);

OneRingAroundVinSmallSet.insertAndFlag1Ring(centerVertex);
                        (*vi).ClearV();
                        //go through the 1-ring neighborhood of
centerVertex and check for an 'F' to see if you should turn it into a
"V"
```

```
                        for (int i = 0; i <
OneRingAroundVinSmallSet.allV.size(); i++){
                            //this is a neighbor of centerVertex in the
small set
                            CVertexO* tempVertexPointer =
OneRingAroundVinSmallSet.allV.at(i);
                            //Check the one-ring of the 'F' and see if
any other small vertices are nearby, if so, delete it
                            if(setFBV[&(*tempVertexPointer)] == 'F') {
                                (*vi).SetV();
                                vcg::tri::Nring<CMeshO>
OneRingARoundF(tempVertexPointer, &m.cm);

OneRingARoundF.insertAndFlag1Ring(tempVertexPointer);
                                (*vi).ClearV();
                                bool isSmall = false;
                                bool intersection = false;
                                for (int j = 0; j <
OneRingARoundF.allV.size(); j++){
                                    CVertexO*
tempVertexPointerNeighborOfF = OneRingARoundF.allV.at(j);
                                    //ignore other 'F's--they have a
different FSetNum system

if(setFBV[&(*tempVertexPointerNeighborOfF)] != 'F') {
                                        //check the setNum of this
neighbor--if it's mode, delete the vertex
                                        int tempSetNumOfNeighborOfF =
FSetNum[tempVertexPointerNeighborOfF];

if(FSetNum[tempVertexPointerNeighborOfF] != mode &&
FSetNum[tempVertexPointerNeighborOfF] != thisSmallSetNum )
                                            intersection = true;
                                        if(
setFBV[&(*tempVertexPointerNeighborOfF)] != 'F' &&
FSetNum[tempVertexPointerNeighborOfF] == mode) {
                                            isSmall = true;
                                        }
                                    }
                                }
                                if(isSmall == true && intersection ==
false){
                                    //change to 'V'
                                    setFBV[&(*tempVertexPointer)] =
'V';
                                    (*tempVertexPointer).C() =
Color4b(Color4b::Black);
                                    FSetNum[&(*tempVertexPointer)] =
thisSmallSetNum;
                                }
                            }//found a 'F'
                        }
                    }//go though every vertex in this small set again
and delete if touches mode
                }//if we are looking at a small set
            }
        }
```

```
        CMeshO::VertexIterator vi;
        vector<CVertexO*>::iterator itVect;
        //show segments again
        if(showSegmentsAgain) {
            //make new disjoint set
            vcg::DisjointSet<CVertexO>* FinalDsetSegment = new
vcg::DisjointSet<CVertexO>();
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //Every vertex not in set 'F' starts as disjoint set
                if(setFBV[vi] != 'F') {
                    //put the vertex in the set D
                    FinalDsetSegment->MakeSet(&(*vi));
                }
            }
            //merge new 'V's with old disjoint sets, merge old disjoint
sets with each other
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is not a feature, then find 1-ring and merge
sets
                if(setFBV[vi] != 'F') {
                    vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*vi));
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        CVertexO* tempVertexPointer =
OneRing.allV.at(i);
                        //if the neighbor of the current vertex is not
a feature then merge sets
                        if(setFBV[&(*tempVertexPointer)] != 'F') {
                            //if they are not already in the same set,
merge them
                            if(FinalDsetSegment->FindSet(&(*vi)) !=
FinalDsetSegment->FindSet(tempVertexPointer)){
                                FinalDsetSegment-
>Union(&(*vi),tempVertexPointer);
                            }
                        }
                    }
                }
            }
            //Find out how many distinct segment sets there are
*****************************
            int FinalsegmentSetCounter = 0;
            vector <CVertexO*> FinalparentVertexofSegmentSet;
            typedef  std::vector < std::vector <CVertexO*> >
colormatrix;
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is not in set 'F', then find what set it
belongs to
                if(setFBV[vi] != 'F') {
                    if (FinalDsetSegment->FindSet(&(*vi)) == &(*vi)){
                        //parent of a new set

FinalparentVertexofSegmentSet.push_back(&(*vi));
                        FinalsegmentSetCounter++;
                    }
                }
            }
```

```
            //******* find biggest set
            typedef  std::vector < std::vector <CVertexO*> > crmatrix;
            crmatrix
FinalvectorOfSetsWithVertices(FinalsegmentSetCounter,
std::vector<CVertexO*>(0));
            // ****************************  Create a vector of
vectors containing our sets / vertices
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                //if it is in set 'F', then find what set it belongs to
                if(setFBV[vi] != 'F') {
                    //find offset in parent vector--that is the vertex
set number
                    itVect =
find(FinalparentVertexofSegmentSet.begin(),
FinalparentVertexofSegmentSet.end(), FinalDsetSegment-
>FindSet(&(*vi)));
                    if( itVect != FinalparentVertexofSegmentSet.end() )
{
                        int offset =
std::distance(FinalparentVertexofSegmentSet.begin(), itVect);
                        //store the set number in the vertex
                        FSetNum[vi] = offset;

FinalvectorOfSetsWithVertices[offset].push_back(&(*vi));
                    }
                }
            }
            delete FinalDsetSegment;
            //Find the set with the most vertices
            int maxInSet = 0;
            int maxOffset = 0;
            for(int pos=0; pos < FinalvectorOfSetsWithVertices.size();
pos++)
            {
                if(FinalvectorOfSetsWithVertices[pos].size() >
maxInSet) {
                    maxInSet =
FinalvectorOfSetsWithVertices[pos].size();
                    maxOffset = pos;
                }
            }
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                if(setFBV[vi] == 'F') {
                    (*vi).C() = Color4b(Color4b::Blue);
                }
                //if it is not in set 'F', then find what set it
belongs to
                if(setFBV[vi] != 'F') {
                    if(FSetNum[vi] == maxOffset) {
                        (*vi).C() = Color4b(Color4b::White);
                    }
                    else {
                        //color ramp

//(*vi).C().ColorRamp(0,vectorOfSetsWithVertices.size(),FSetNum[vi]);
                        int ScatterSize =
FinalvectorOfSetsWithVertices.size();
```

```
                        Color4b BaseColor =
Color4b::Scatter(ScatterSize, FSetNum[vi],.7f,.9f);
                        (*vi).C()=BaseColor;
                    }
                }
            }//go through vertices
            //remove lines surrounded by one color
            //Check if color same on both sides of F line
            for(vi = m.cm.vert.begin(); vi != m.cm.vert.end(); ++vi) {
                if(setFBV[vi] == 'F') {
                    (*vi).SetV();
                    vcg::tri::Nring<CMeshO> OneRing(&(*vi), &m.cm);
                    OneRing.insertAndFlag1Ring(&(*vi));
                    (*vi).ClearV();
                    Color4b color1;
                    Color4b color2;
                    Color4b color3;
                    Color4b blueColor = Color4b(Color4b::Blue);
                    Color4b tempColor;
                    int colorCounter = 0;
                    CVertexO* tempVertexPointer;
                    for (int i = 0; i < OneRing.allV.size(); i++){
                        if (i == 0) {
                            tempVertexPointer = OneRing.allV.at(i);
                            color1 = (*tempVertexPointer).C();
                            colorCounter++;
                        }
                        if(i > 0 && i < OneRing.allV.size()) {
                            tempVertexPointer = OneRing.allV.at(i);
                            tempColor = (*tempVertexPointer).C();
                            if(colorCounter == 1) {
                                if(tempColor != color1) {
                                    color2 = tempColor;
                                    colorCounter++;
                                }
                            }
                            if(colorCounter == 2) {
                                if(tempColor != color1 && tempColor !=
color2) {
                                    color3 = tempColor;
                                    colorCounter++;
                                }
                            }
                        }
                    }
                    //delete it from feature
                    if(colorCounter == 2 || colorCounter == 1){
                        if(colorCounter == 2){
                            //delete from feature if surrounded by a
color
                            setFBV[vi] = 'V';
                            if(color1 != blueColor) {
                                (*vi).C() = color1;
                            }
                            if(color2 != blueColor) {
                                (*vi).C() = color2;
                            }
```

```
                        }
                        if(colorCounter == 1) {
                            //delete from feature if surrounded by a
color
                            setFBV[vi] = 'V';
                            if(color1 != blueColor) {
                                (*vi).C() = color1;
                            }
                        }
                    }
                }
            }//check if color on both sides of line same
        }//show segments again
        m.updateDataMask(MeshModel::MM_VERTQUALITY);
        m.updateDataMask(MeshModel::MM_VERTCOLOR);
        // delete attibute by name

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<int>(m.cm,marked)
;

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<float>(m.cm,disth
);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<char>(m.cm,setFBV
);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<float>(m.cm,divfv
al);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<int>(m.cm,FSetNum
);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<int>(m.cm,CpointO
wner);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<CMeshO::VertexPoi
nter>(m.cm,source);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<CMeshO::VertexPoi
nter>(m.cm,source1);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<float>(m.cm,curva
ture);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<int>(m.cm,disc);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<int>(m.cm,center)
;

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<int>(m.cm,complex
);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<int>(m.cm,newF);

vcg::tri::Allocator<CMeshO>::DeletePerVertexAttribute<float>(m.cm,sumPo
sSetcurvature);
        break;
```

```
    }
    break;
    }
    return true;
}
MeshFilterInterface::FilterClass
ExtraMeshcolorizeteethPlugin::getClass(QAction *a){
    switch(ID(a)){
    case   CP_DISCRETE_CURVATURETEETH:
        return MeshFilterInterface::VertexColoring;
    default:
        assert(0);
        return MeshFilterInterface::Generic;
    }
}
int ExtraMeshcolorizeteethPlugin::getPreConditions(QAction *a) const{
    switch(ID(a)){
    case CP_DISCRETE_CURVATURETEETH:
        return MeshModel::MM_FACENUMBER;
    default: assert(0);
        return MeshModel::MM_NONE;
    }
}
int ExtraMeshcolorizeteethPlugin::postCondition( QAction* a ) const{
    switch(ID(a)){
    case CP_DISCRETE_CURVATURETEETH:
        return MeshModel::MM_VERTCOLOR | MeshModel::MM_VERTQUALITY |
MeshModel::MM_VERTNUMBER;
    default: assert(0);
        return MeshModel::MM_NONE;
    }
}
Q_EXPORT_PLUGIN(ExtraMeshcolorizeteethPlugin)
```

APPENDIX B

CHANGE TO VCG'S NRING.H SOURCE CODE

In order to assure that the vertices comprising the n-ring neighborhood of a central vertex are explored in order (without skipping around) and in a consistent direction (clockwise vs. counterclockwise) the native VCG nring.h file was changed to the following:

```
#ifndef RINGWALKER_H
#define RINGWALKER_H
#include <vcg/simplex/face/jumping_pos.h>
#include <vcg/complex/allocate.h>
#include <vcg/complex/algorithms/update/flag.h>
#include <vector>
#include <common/interfaces.h>
namespace vcg
{
namespace tri
{
        /** \addtogroup trimesh */
        /*@{*/
        /*@{*/
        /** Class Mesh.
         This is class for extracting n-ring of vertexes or faces,
starting from a vertex of a mesh.
         */
template <class MeshType>
class Nring
{
public:
    typedef typename MeshType::FaceType   FaceType;
    typedef typename MeshType::VertexType VertexType;
    typedef typename MeshType::ScalarType ScalarType;
    typedef typename MeshType::FaceIterator FaceIterator;
    typedef typename MeshType::VertexIterator VertexIterator;
    typedef typename MeshType::CoordType CoordType;
    std::vector<VertexType*> allV;
    std::vector<FaceType*> allF;
    std::vector<VertexType*> lastV;
    std::vector<FaceType*> lastF;
    MeshType* m;
    Nring(VertexType* v, MeshType* m) : m(m)
    {
        assert((unsigned)(v - &*m->vert.begin()) < m->vert.size());
        insertAndFlag(v);
```

```
    }
    ~Nring()
    {
                clear();
    }
    void insertAndFlag1Ring(VertexType* v)
    {
        insertAndFlag(v);
        typename face::Pos<FaceType> p(v->VFp(),v);
        assert(p.V() == v);
        int count = 0;
        face::Pos<FaceType> ori = p;
        int numFaces = 0;
        do
        {
            insertAndFlag(p.F(), v, numFaces);
            p.FlipF();
            p.FlipE();
            assert(count++ < 100);
            numFaces++;
        } while (ori != p);
    }
    void insertAndFlag(FaceType* f, VertexType* v, int numFaces)
    {
        if (!f->IsV())
        {
            allF.push_back(f);
            lastF.push_back(f);
            f->SetV();
            //CVertexO* tempV = v;
            //CVertexO* tempVertex = f->V(0);
            //CVertexO* tempVertex1 = f->V(1);
            //CVertexO* tempVertex2 = f->V(2);
            VertexType* tempVertex = f->V(0);
            VertexType* tempVertex1 = f->V(1);
            VertexType* tempVertex2 = f->V(2);
            //qDebug() << "****************** New Face with these
vertices:";
            //qDebug() << "f->V(0): " <<  (*tempVertex).P().X() << " "
<< (*tempVertex).P().Y() << " " << (*tempVertex).P().Z();
            //qDebug() << "f->V(1): " <<  (*tempVertex1).P().X() << " "
<< (*tempVertex1).P().Y() << " " << (*tempVertex1).P().Z();
            //qDebug() << "f->V(2): " <<  (*tempVertex2).P().X() << " "
<< (*tempVertex2).P().Y() << " " << (*tempVertex2).P().Z();
            //deal with special case that makes the first face appear
to be clockwise
            if(numFaces == 0) {
               // if((*tempVertex1).P().X() == v.X() &&
(*tempVertex1).P().Y() == tempV.P().Y() && (*tempVertex1).P().Z() ==
tempV.P().Z()) {
                  if(tempVertex1 == v) {
                    //change the order you send them in (send them in
counterclockwise)
                    //qDebug() << "correction made";
                    insertAndFlag(f->V(2));
                    insertAndFlag(f->V(0));
                    insertAndFlag(f->V(1));
```

```
                }
                else {
                      insertAndFlag(f->V(0));
                      insertAndFlag(f->V(1));
                      insertAndFlag(f->V(2));
                }
            }
            else {
                insertAndFlag(f->V(0));
                insertAndFlag(f->V(1));
                insertAndFlag(f->V(2));
            }
        }
    }
    void insertAndFlag(VertexType* v)
    {
        if (!v->IsV())
        {
            allV.push_back(v);
            lastV.push_back(v);
            v->SetV();
        }
    }
    static void clearFlags(MeshType* m)
    {
        tri::UpdateFlags<MeshType>::VertexClearV(*m);
        tri::UpdateFlags<MeshType>::FaceClearV(*m);
    }
    void clear()
    {
        for(unsigned i=0; i< allV.size(); ++i)
            allV[i]->ClearV();
        for(unsigned i=0; i< allF.size(); ++i)
            allF[i]->ClearV();
        allV.clear();
        allF.clear();
    }
    void expand()
    {
      std::vector<VertexType*> lastVtemp = lastV;
        lastV.clear();
        lastF.clear();
        for(typename std::vector<VertexType*>::iterator it =
lastVtemp.begin(); it != lastVtemp.end(); ++it)
        {
            insertAndFlag1Ring(*it);
        }
    }
    void expand(int k)
    {
        for(int i=0;i<k;++i)
            expand();
    }
};
}} // end namespace NAMESPACE
#endif // RINGWALKER_H
```

APPENDIX C

IRB APPROVAL FORM

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

Institutional Review Board for Human Use

Form 4: IRB Approval Form
Identification and Certification of Research
Projects Involving Human Subjects

UAB's Institutional Review Boards for Human Use (IRBs) have an approved Federalwide Assurance with the Office for Human Research Protections (OHRP). The Assurance number is FWA00005960 and it expires on August 29, 2016. The UAB IRBs are also in compliance with 21 CFR Parts 50 and 56.

Principal Investigator:	MOURITSEN, DAVID A
Co-Investigator(s):	
Protocol Number:	**E111123002**
Protocol Title:	*Segmentation of Teeth in Digital Dental Models*

The above project was reviewed on 1-24-12. The review was conducted in accordance with UAB's Assurance of Compliance approved by the Department of Health and Human Services. This project qualifies as an exemption as defined in 45CF46.101, paragraph 4.

This project received EXEMPT review.

IRB Approval Date: 1-24-12

Date IRB Approval Issued: 1-24-12

Marilyn Doss

Marilyn Doss, M.A.
Vice Chair of the Institutional Review
Board for Human Use (IRB)

Investigators please note:

IRB approval is given for one year unless otherwise noted. For projects subject to annual review research activities may not continue past the one year anniversary of the IRB approval date.

Any modifications in the study methodology, protocol and/or consent form must be submitted for review and approval to the IRB prior to implementation.

Adverse Events and/or unanticipated risks to subjects or others at UAB or other participating institutions must be reported promptly to the IRB.

470 Administration Building
701 20th Street South
205.934.3789
Fax 205.934.1301
irb@uab.edu

The University of
Alabama at Birmingham
Mailing Address:
AB 470
1530 3RD AVE S
BIRMINGHAM AL 35294-0104