



Implementation of Morphological Operators for Surface Segmentation

Marcel Alcoverro

► To cite this version:

Marcel Alcoverro. Implementation of Morphological Operators for Surface Segmentation. 2007. hal-00749115

HAL Id: hal-00749115

<https://hal.science/hal-00749115>

Submitted on 6 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Implementation of Morphological Operators for Surface Segmentation

Marcel Alcoverro Vidal

Supervisor : Sylvie Philipp-Foliguet
Ecole Nationale Supérieure de l'Electronique et de ses Applications

September 10, 2007

Acknowledgements

Dedico aquest treball a la Lara, sense tu no hagués estat possible. Gràcies per haver-me escoltat els rotllos dels watersheds i skeletons, per haver-me animat en els moments difícils. També el vull dedicar als meus pares, que sou els qui heu aguantat tants maldecaps, tants anys. Ja s'ha acabat per fi la carrera infinita !

Je veux remercier Sylvie Phillipp, Michel Jordan, Laurent Najman et Jean Cousty pour m'accueillir dans leur équipes. Merci pour m'avoir découvert le monde de la recherche et me permettre participer de très bonnes discussions. Merci Jean pour consacrer ton temps à m'aider a ne me perdre pas dans les chemins de la morphologie.

Resum

Aquest treball forma part del projecte Eros3d, el qual tracta la gestió de bases de dades d'obres d'art. El C2RMF (Centre de Recherche et de Restauration des Musees de France) ha organitzat durant uns anys les tasques de digitalització dels museus francesos i actualment es disposa d'una base de dades de 650 objectes 3D. El seu objectiu és dissenyar una arquitectura de software capaç d'emmagatzemar i manejar (visualitzar, cercar, comparar) les dades per diferents nivells d'usuari. Concretament el treball realitzat forma part d'un sistema de recuperació de dades segons el contingut.

Un sistema de recuperació de dades segons el contingut (*content-based retrieval system*) recupera els objectes de la base de dades basant-se en la similaritat entre els objectes. La base de dades ha d'indexar-se per tal de dur a terme la classificació i cerca, i per realitzar la indexació s'utilitzen vectors de característiques per a cada objecte. El plantejament adoptat per a l'obtenció de les característiques de cada objecte és la segmentació de la superfície del model en diferents regions significatives. Posteriorment es realitza un càlcul de diferents trets característics per a cada regió. Aquestes diferents característiques obtingudes per a cada model 3D serviran tant per a comparacions basades en l'objecte sencer, com també per realitzar consultes parcials consistents en la cerca a partir d'un conjunt de regions de la superfície de l'objecte.

La divisió de la superfície de l'objecte es realitza partint d'un mapa de curvatura calculat sobre la malla triangular que representa l'objecte. L'operador utilitzat per fer la partició és la transformada *watershed* (línia divisòria d'aigües) aplicada sobre el mapa de curvatura. El watershed és un operador morfològic molt utilitzat en el processament d'imatge. Si imaginem una funció com un relleu topogràfic on els valors de cada punt són l'alçada d'aquest punt en el relleu, un regió seria el conjunt de punts on si i caigués una gota d'aigua, aquesta baixaria fins al fons d'una mateixa vall. Partint d'una funció en un espai determinat, el watershed obté regions que extenen els mínims regionals de la funció. En el nostre cas, representem la funció curvatura sobre la malla 3D amb un graf valuat en les seves arestes. Apliquem la transformada watershed sobre aquest graf utilitzant l'algorisme presentat per Jean Cousty [5] de càlcul de watersheds en grafs valuats en les arestes.

La transformada watershed aplicada sobre una funció obté un nombre de regions equivalent al nombre de mínims regionals de la funció de partida. El resultat ens pot portar a una sobre-segmentació no desitjada. Per tal de controlar el nombre de regions obtingudes pel watershed així com les característiques d'aquestes regions, abans del càlcul d'aquest, filtrem la funció original per tal de reduir-ne el nombre de mínims regionals. L'operador que implementem per al filtratge de la funció de curvatura original és l'*arbre de components*. La combinació *arbre de components + watershed* ha estat ampliament utilitzada en processat d'imatge, vídeo i senyal, i aporta robustesa als resultats de segmentació obtinguts amb el watershed. L'algorisme implementat és una adaptació original de l'algorisme per al càlcul de l'arbre de components presentat per Najman i Couprie [14].

L'arbre de components és una estructura que ordena els components connexos d'un graf. Considerem un graf valuat G . Un component connex d'aquest graf G és un subgraf de G connex en que els seus nodes o arestes tenen el mateix valor. L'arbre de compo-

nents estableix una jerarquia entre els diferents components connexos d'un graf. Aleshores mitjançant aquest arbre podem modificar els valors del graf de manera que eliminem els components connexos que no desitgem. En el nostre cas els valors del graf són els valors de curvatura de la superfície de l'objecte que analitzem. Si per exemple considerem un component connex petit de valor molt petit, aquest correspondrà a una zona de la superfície de l'objecte petita amb molta curvatura. Aquesta zona pot correspondre a una irregularitat poc significativa de la superfície, o a sorolls en el procés d'obtenció del model. Mitjançant l'arbre de components els valors de curvatura d'aquesta zona prendran els valors dels punts adjacents, de manera que la irregularitat de la funci curvatura desapareix. Mentre la transformada watershed obtindria una regió per aquesta zona petita del mapa de curvatura original, al aplicar-la al mapa filtrat per l'arbre de components aquesta zona no significativa ja no apareix.

Un cop obtingudes les diferents regions de la superfície es calculen tres característiques diferents per cada regió. Per una banda un histograma de la longitud de les cordes (distància d'un punt al baricentre) i angle amb els eixos principals de l'objecte. Un altre s un histograma de valors de curvatura. Per últim, l'histograma EGI (*Extended Gaussian Images*). Considerem una esfera de Gauss de l'objecte, l'histograma EGI compta les projeccions de cada punt de l'objecte en aquesta esfera. En el nostre cas el recompte es realitza per cada regió.

A partir dels histogrames de cada regió es poden construir els vectors de característiques necessaris per la indexació de la base de dades d'objectes. El mètode presenta avantatges respecte tècniques d'extracció de característiques geomètriques globals ja que permet trobar similituds i diferències en objectes que tenen la mateixa forma i on els trets característics es troben en detalls de la superfície. Aquest és el cas per exemple de les escultures antigues que trobem a la base Eros3d provinents del Museu del Louvre, on la forma d'aquestes és pràcticament cilíndrica, mentre que les particularitats es troben esculpides sobre la superfície.

Abstract

A content-based retrieval system needs feature vectors for database indexation. We adopt the surface segmentation approach to obtain several features for a 3D object which can be used to retrieve objects in a database from a partial request composed from a set of regions. To achieve the segmentation of the surface into several regions we apply the watershed transform on a curvature map computed on the 3D surface mesh. The watershed applied directly on the original curvature map produces an over-segmentation of the object surface. Thus, we previously filter the original curvature map by using the component tree. After this filtering, the watershed transform is computed on the filtered curvature map and we obtain the desired number of regions. Then we proceed by computing some features for each region obtained, which will serve as feature vectors for a content-based search and retrieval system. The techniques we apply on the surface of 3D objects have been presented for image applications.

Consider a 3D triangular mesh (a set of points, triangles and sides of triangles). We build an edge weighted graph to represent the mesh. Weights on edges of the graph correspond to the curvature map computed on the mesh. The component tree is a structure to order the connected components of a weighted graph. We implement an original algorithm to build the component tree of an edge weighted graph based on the one presented by Najman and Couprie [14]. This structure allows to reduce the number of minima of the original map on edges of the graph. We implement the algorithm proposed by Cousty [5] to compute the watershed transform on an edge weighted graph. By using the watershed transform we obtain a number of regions of the map which equals the number of minima of the input map.

Once we obtain the partition of the mesh into several regions we compute features of each region. These features consist in histograms for each region considering three different approaches: Extended Gaussian Images (EGI) [8]; a cords histogram (considering the cords lenght and a principal angle); a curvature histogram (considering principal curvatures). These histograms form feature vectors for each region which will help the database indexation and classification.

Contents

1	Introduction	7
2	Mesh characterization and surface analysis	7
2.1	Edge-weighted graphs	8
2.2	Simplicial complexes	9
2.3	Curvature	10
2.4	Vertex per-face graph	12
2.5	Mesh Repair	12
3	Watershed	14
3.1	Extensions and graph cuts	15
3.2	Watersheds and catchment basins	15
3.3	Minimum spanning forests and watershed optimality	17
3.3.1	Minimum spanning forests and minimum spanning trees	18
3.4	Watershed algorithm	19
3.5	Border thinning on simplicial complexes	21
4	Component tree	23
4.1	Connected components notions	24
4.2	Component tree definition	25
4.3	Component tree and minimum spanning tree	25
4.4	The Union-Find method	27
4.5	Component tree algorithm	28
4.5.1	High-level view	28
4.5.2	Detailed view	29
4.5.3	Example	32
4.6	Node Attributes	35
4.7	Filtering	37
5	Region attributes	38
5.1	Cords histogram	38
5.2	Extended Gaussian Images	39
5.3	Curvature histogram	40
6	Experimental results	40
7	Conclusion	42

1 Introduction

The recent development of 3D object acquisition methods involve a need to handle this kind of information. Nowadays 3D object databases appear in various areas for leisure as well as for scientific applications (medical, industrial part catalogues, cultural heritage, etc.). Large database can be quickly populated using 3D mesh acquisition and reconstruction tools which have become easy to use. As database size is growing, tools to retrieve information as indexing methods, search algorithms and data classification techniques become more and more important.

A significant amount of work has been done in the past two decades on text-based document retrieval. The Google search engine has become a standard as text-based search engine. Indexing by keywords and search achieved through text retrieval techniques has the advantage that it is “high level” (semantic level), but keywords are external information which is often manually assigned. More recently content-based retrieval systems have been developed for images, audio, and video to automatically index and retrieve information from digital libraries. Content-based retrieval systems retrieve objects based on the integral similarity of objects.

In search-by-similarity, the goal is to find objects which are “close” to the example. It is done with respect to a given similarity measurement and thanks to object *indexes* computed on object *features*. These features may be of various kinds (points, segments, regions, etc.) and may have different properties such as scale invariance, rotation invariance, etc.

Different techniques can be used to extract features of the 3D object, that can be obtained from a shape representation (Global feature-based techniques, graph-based techniques, recognition-based techniques). Surface segmentation can be applied also for feature detection.

This work is part of the Eros3d project. The Eros3d project deals with artwork database management. C2RMF (Centre de Recherche et de Restauration des Musees de France) has organized digitalization lobbying in french museums for some years. 650 3D objects are available in the database. Its aim is to design a software architecture that is supposed to store and handle (view, research, compare) data at different user levels.

2 Mesh characterization and surface analysis

Segmentation of a polygonal mesh is a method of splitting the mesh into regions in a “meaningful” manner. A mesh consists of a set of n points ($x_i \in \mathbb{E}^3; 0 \leq i < n$) and a set of planar convex polygons made up of these points. A first question to consider in order to partition the mesh is to define which components of the mesh will form the regions and how their boundaries will be defined. Two different approaches that have been used before are summarized in [17], called vertex based method and edge based method.

The vertex based methods consider a value associated at each vertex (*i.e* curvature) and define the segmentation as regions that consist of connected vertices that have the same property. A major drawback in this approach is that no boundaries are created

for the regions and each vertex has its own region information. Therefore triangles have multi-region information, so the three vertices of a triangle can be part of three different regions, whereas the triangle itself would not belong to any region.

The edge based methods define an edge as boundary if it is shared by two planes whose normal vectors make an angle greater than a certain threshold. This results in disconnected boundary edges and thereby open regions.

Our approach defines regions as sets of connected faces with edges as their boundaries, that leads to obtain a surface divided in closed independent regions consisting of connected triangles after the segmentation process. We illustrate our approach in fig. 3.

We represent the mesh by a graph, and the curvature will be defined as a weight function on edges, so we will introduce some notations for edge-weighted graphs. Also we will introduce some notions on simplicial complexes, since it is a structure that allows to describe the topological properties of a mesh.

2.1 Edge-weighted graphs

We define a *graph* as a pair $X = (V(X), E(X))$ where $V(X)$ is a finite set and $E(X)$ is composed of unordered pairs of $V(X)$, i.e., $E(X)$ is a subset of $\{\{x, y\} \subseteq V^2(X) \mid x \neq y\}$. Each element of $V(X)$ is called a *vertex* and each element of $E(X)$ is called an *edge*.

Let X be a graph. Let x and y be vertices of X . We say that x and y are *adjacent* if $\{x, y\}$ is an edge of X . A sequence $\pi = \langle x_0, \dots, x_l \rangle$ of vertices of X is a *path in X (from x_0 to x_l)* if x_i and x_{i+1} are adjacent for each $i = 0, \dots, l-1$. We say that X is *connected* if, for any pair of vertices (x, y) in X , there is a path in X from x to y .

Let X and Y be two graphs. If $V(Y) \subseteq V(X)$ and $E(Y) \subseteq E(X)$, we say that Y is a *subgraph of X* and we write $Y \subseteq X$.

Let X and Y be two graphs and $Y \subseteq X$, Y a subgraph of X . We say that Y is a *connected component of X* , or simply a *component of X* , if Y is a connected subgraph of X which is maximal for this property, i.e., for any connected graph Z , $Y \subseteq Z \subseteq X$ implies $Z = Y$.

Throughout this paper G denotes a connected graph.

We denote by \mathcal{F} the set of all maps from E to \mathbb{R} . Let $F \in \mathcal{F}$. If u is an edge of G , $F(u)$ is the *altitude* of u . In our application the curvature will define the altitude of the edges of the graph. We also will denote by F the map from V to \mathbb{R} such that for any $x \in V$, $F(x)$ is the minimal altitude of an edge at x , i.e., $F(x) = \min\{F(u) \mid u \in E, x \in u\}$.

Let $X \subseteq G$ and $k \in \mathbb{R}$. A subgraph X of G is a *minimum of F (at altitude k)* if:

- X is connected;
- k is the altitude of any edge of X ;

- the altitude of any edge adjacent to X is strictly greater than k .

We denote by $M(F)$ the graph whose edge set is the union of the edge sets of all minima of F .

2.2 Simplicial complexes

We extract from [2] and [3] some notions and notations of complexes.

A (*finite simplicial*) *complex* X is a finite family composed of finite nonempty sets such that, if f is an element of X , then every nonempty subset of f is an element of X . Each element of a complex is called *face*. The *dimension* of a face f is the number of its elements minus one. We call an m -*face* a face of dimension m . We denote by \mathbb{K} the collection of all complexes.

In fig. 1(a) we illustrate a graphical representation of a 0-face, a 1-face and a 2-face.

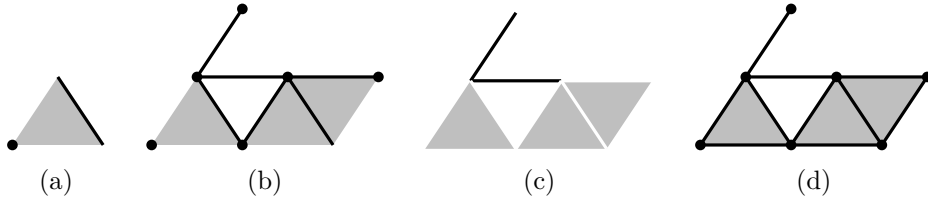


Figure 1: (a) A set of faces (one 2-face, one 1-face, one 0-face). (b) A set of faces X which is not a complex. (c) The set X^+ , all the facets of X . (d) The set X^- , the closure of X , which is a complex.

Let f be a finite nonempty set. We set $\hat{f} = \{g | g \subseteq f, g \neq \emptyset\}$ and $\hat{f}^* = \hat{f} \setminus \{f\}$. Any $g \in \hat{f}$ is a *face of f* , and any $g \in \hat{f}^*$ is a *proper face of f* . If X is a finite set of faces in \mathbb{F}_2^n , we write $X^- = \cup\{\hat{f} | f \in X\}$, X^- is the *closure of X* . Thus, a finite family X of finite nonempty sets is a complex if and only if $X = X^-$.

Any subset Y of a complex X which is also a complex is a *subcomplex of X* , and we write $Y \preceq X$. If X is a complex in \mathbb{K} we also denote $X \preceq \mathbb{K}$.

Let $X \preceq \mathbb{K}$. A face $f \in X$ is a *facet of X* if there is no $g \in X$ such that $f \in \hat{g}^*$. We denote by X^+ the set composed of all facets.

In fig. 1 we illustrate these notions. The set of faces X of fig. 1(b) are not a complex. As it can be observed, X does not equal its closure X^- depicted in fig. 1(d).

2.3 Curvature

The computation of the curvature has been done with the software Trimesh provided by Princeton University [18]. This software provides a method based on computing first the curvature per-face and then estimates the value at each vertex as a weighted average over the immediately adjacent faces.

The *normal curvature* κ_n of a surface in some direction is the reciprocal of the radius of the circle that best approximates a normal slice of surface in that direction. The normal curvature can be expressed as $\kappa_n = \kappa_1 s^2 + \kappa_2 t^2$ where κ_1 and κ_2 are the *principal curvatures* and (s, t) are the *principal directions*, which are the directions where the normal curvature reaches its minimum and maximum. These directions are ortogonal.

The *Gaussian curvature* K is equal to the product of the principal curvatures: $K = \kappa_1 \kappa_2$, and the *mean curvature* H is their average: $H = (\kappa_1 + \kappa_2)/2$.

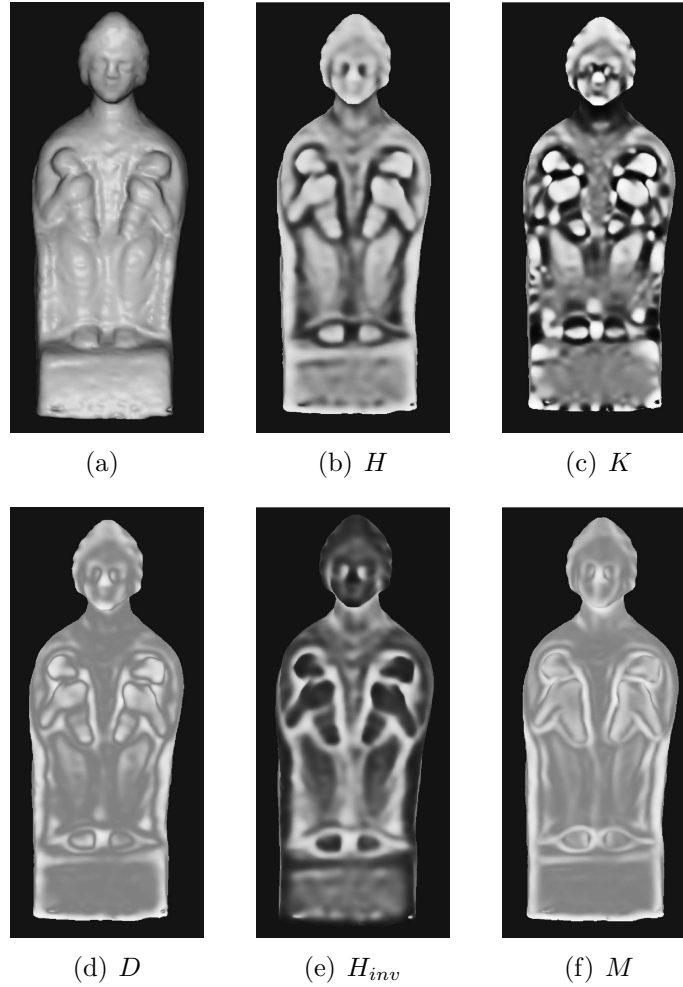


Figure 2: Curvature scalar functions of the object in (a) in grayscale.

After the computation of Trimesh algorithm on a 3D mesh we obtain the values κ_1 and

κ_2 on each vertex of the mesh. These values increase with the convexity of the surface. They decrease into negative values on concave zones, getting low absolute values on flat zones. Considering the combination of the principal curvatures κ_1 and κ_2 on a surface we have *convex zones* when both values are great positive, *concave zones* when both are great negative and *saddle zones* when one value is great positive and the other great negative, meaning convexity in one direction and concavity in the other. The *flat zones* have both values low.

As the curvature map will be used to partition the surface by using the watershed operator, a single scalar function is desired. Several approaches can be done to obtain this height function by combining values κ_1 and κ_2 . Mainly, the choice will depend on the desired further applications. Also the class of objects or their shape characteristics may determine which are the “meaningful” regions. For example, if we are dealing with objects made of flat smooth polygonal parts (cars, manufactured pieces, furniture, etc.), we should be interested in obtaining regions of this flat parts, thus the divide line between regions would be placed on high curvature edges. In the case of art objects, the pieces could be characterized by their carved features, thus it would be interesting to place the lines on the concave zones dividing convex parts.

Mangan and Whitaker [11] use as magnitude for curvature the *deviation from flatness*

$$D = \sqrt{4H^2 - 2K^2}$$

where H is the mean curvature and K the gaussian curvature. This function gives high values on convex and concave zones, while it is low on flat and saddle zones.

Other approach we adopted is to use mean curvature H in the form

$$H_{inv} = \arctan(-H + \pi/2)$$

This function has the behavior of the inverse of the mean curvature, but taking always positive values. It gives high values to concave zones and low values to convex zones.

We consider also a *max curvature* as

$$M = \max(\kappa_1^2, \kappa_2^2)$$

and that gives high values on convex and concave zones, as H^2 . The max curvature has also high values on zones that are flat in one direction, and convex or concave in the other. This zones are commonly the edges that divide planes of an object, as the division between the roof and the doors of a car.

We have used this different treatments of the principal curvatures and, for the art objects we deal with, the H_{inv} function is the one with which we obtained the best results, while the max curvature M gives better results for manufactured objects.

In fig. 2 are depicted the values of these scalar functions in grayscale for the sculpture in fig. 2(a). Low values are black, while great values are white.

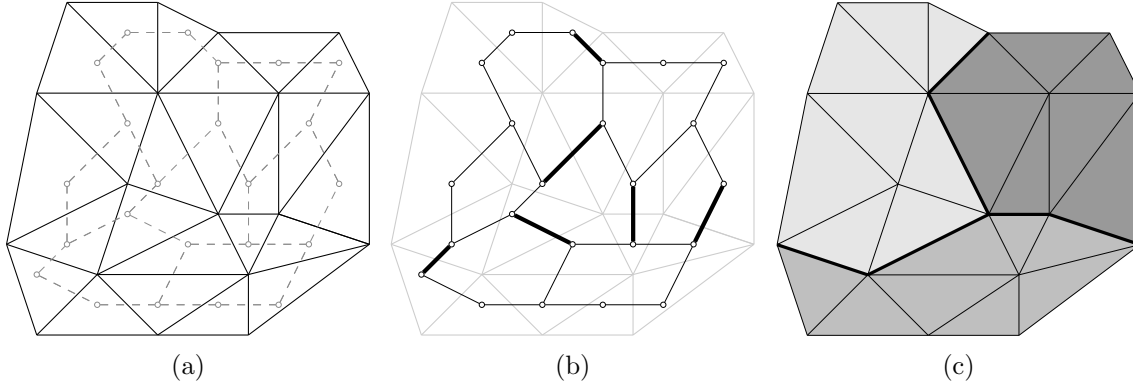


Figure 3: (a) A triangle mesh and a vertex per-face graph. (b) Segmentation on edges of the graph (in bold). (c) Segmentation on the mesh.

2.4 Vertex per-face graph

Consider a 3D surface mesh M (composed of *points, triangles, sides of the triangles*) so that for any side e in M there is exactly one pair of triangles (g, h) such that $e \in g$ and $e \in h$. We build a graph $G = (V, E, F)$ with one vertex for each face of M and an edge connecting two vertices if the corresponding two faces share a side. We will call this graph a *vertex per-face graph*. An example of a vertex per-face graph is depicted in fig. 3(a).

Let be e any side of a triangle in M and (x, y) the pair of points such that $e = \{x, y\}$. As described in section 2.3 we have computed the curvature values in each point of the mesh. We denote them as κ_{1x} , κ_{2x} and κ_{1y} , κ_{2y} for the points x and y respectively. Then we will compute for each e in M ,

$$\kappa_1 = \frac{\kappa_{1x} + \kappa_{1y}}{2} \quad \kappa_2 = \frac{\kappa_{2x} + \kappa_{2y}}{2}$$

Considering then the scalar curvature functions explained above (section 2.3) we obtain then a map from E into \mathbb{R} that we denote by F , that will represent the curvature between each two adjacent faces of the mesh.

2.5 Mesh Repair

The triangle meshes obtained from acquisition of real-objects and also CAD generated models often have defects that may cause problems in further processing. In our case the condition that will allow us to proceed properly would be a triangular surface mesh that forms a manifold without boundaries. Thus, the set of points, edges and triangles that form a mesh should be a complex K , in which \forall 1-face $u \in K$ there exists exactly two 2-faces $f \in K$, $g \in K$, such that $f \cap g = u$. If this is accomplished we can obtain the vertex per-face graph described previously.

The degeneracies that usually appear may be holes, tubes, duplicated faces, intersecting faces or borders, and not all of them can be solved in the same way. We have tried different

approaches, which have been based on available software, as it was out of our scope to implement a new application for mesh repairing purposes.

The application ReMESH [1] provides an interactive environment for repairing meshes. A visualization tool is provided, and the software allows to detect several degeneracies, as duplicated faces, holes, intersecting triangles. It provides also tools to remove the defects, to fill holes after, and also has utilities to build again the mesh. We have not used this application for repairing our objects, as with it meshes should be repaired manually, and for our purposes the approach should be automatic. Even though, it has been useful to visualize the kind of degeneracies we deal with, to plan other approaches and to test if the other approaches worked well.

In order to get the proper meshes, we use two different automatic approaches. Both of them rebuild the mesh as it assures that the definitive mesh accomplishes our conditions. Also for both of them the steps involved have been the same, *voxelization*, *isosurface extraction*, *mesh smoothing* and *mesh size optimization* similar as it is presented by Nooruddin and Turk [15]. The differences are on the techniques adopted on the voxelization step.

Voxelization Voxelization means converting a polygonal model into a volume. The first approach used to reach the volume from the original mesh has been using the library Pink [4] to proceed with the following steps:

- obtain the points (*i.e* vertices of each triangle) of the mesh, and build a 3D grid where we place the points.
- calculate a distance map of this grid. Each cell of the grid get a value which is the minimum distance to a point, while point cells get 0 value.
- apply a watershed segmentation on the inverse of the distance map. We use markers for the watershed that are: a point in the interior of the object; a frame of the grid as marker for the exterior.

The watershed operator produces a division of the grid into two regions, that are the interior and the exterior of the object. We found problems in this approach due to the need for automatisation. One problem comes from obtaining a point of the interior of the object as marker for the watershed. A implementation of a method in a step before the extraction of the vertices of the mesh into the grid is needed. We used the barycenter of the object as marker, but it fails as it is not always in the interior. Other problem is that the resulting volume may have broken parts corresponding to thin parts of the original mesh.

The other approach we tested for the voxelization is the one presented at [15], which has been implemented at Princeton [13]. The method used is called *parity count* which consists in classify a voxel V by counting the number of times that a ray with its origin at the center of V intersects polygons of the model. An odd number of intersections means that V is interior to the model and even number means that it is outside. To improve the technique for models that have cracks or holes, that will cause a bad classification, a voting scheme is adopted, by using more directions for the rays, and classifying by a

majority vote. This approach worked well, even though it creates some tunnels due to degeneracies not solved by the voting scheme. At [15] is proposed to apply morphological operators to fill this tunnels, as erosion and dilation, but it has not been tested.

Isosurface extraction After the voxelization step we extract an isosurface from the volumetric representation to create a manifold polygonal model. We used the library Pink for this computing, which implements a topologically correct Marching Cubes algorithm [10].

Mesh smoothing The isosurfaces extracted after the marching cubes algorithm result in a staircase surface. We use Taubin’s smoothing technique implemented in Trimesh [18] which creates a new surface smoother than the original by using a low-pass filter over the position of the vertices.

Mesh size optimization The surface mesh produced by the marching cubes algorithm is usually over-tessellated. We used the software Yams [6] to optimize the triangle count of the mesh. The remeshing algorithm produce a mesh where elements size is related to local curvature.

After these steps we obtain a new mesh free of degeneracies that accomplishes our conditions of a manifold without boundaries. Even though, the method should be improved to perform better in some cases, as when it generates tunnels in the resulting object, or broken thin parts. Also the automatisation of resolution related parameters should be improved.

3 Watershed

The watershed transform has been widely used as a fundamental step in many segmentation procedures. This algorithm operates on a height function which is defined over the corresponding domain. It was introduced for image segmentation and also has been used on 3D surfaces [11]. We use the algorithm introduced by Cousty [5] which is defined on the framework of edge-weighted graphs.

The watershed method derives the name from the manner how it segments its height function domain in *catchment basins*. Imagine a drop of water falling on a relief described by the height function. It will follow the steepest descent path until it reaches a local minimum. The set of points whose steepest descent paths terminate at the same minimum of the function forms a catchment basin. The watershed line will be the line that separates the different catchment basins.

In our case the height function F is the curvature defined on the edges of a graph $G = (V, E, F)$ and the regions of the watershed are computed as a extension of the minima of that graph.

3.1 Extensions and graph cuts

The notions of extension and graph cut are important to define the method used to compute the watershed. The following definitions are extracted from [5] and formalize both notions for the framework of graphs.

Definition 1. Let X and Y be two non empty subgraphs of G . We say that Y is an extension of X (in G) if $X \subseteq Y$ and if any component of Y contains exactly one component of X .

The subgraphs drawn in bold in Fig.4(b) and Fig.4(c) are extensions of the subgraph in Fig.4(a).

Producing an extension until it covers all the vertices of the graph will form a separation that is called a graph cut.

Definition 2. Let $X \subseteq G$ and $S \subseteq E$. We say that S is a (graph) cut for X if \overline{S} is an extension of X and if S is minimal for this property, i.e., $T \subseteq S$ and \overline{T} is an extension of X imply $T = S$.

The set S depicted in Fig.4(d) is a cut for the subgraph X in Fig.4(a). \overline{S} in Fig.4(c) is an extension of X , and it can be seen that it is a maximal extension.

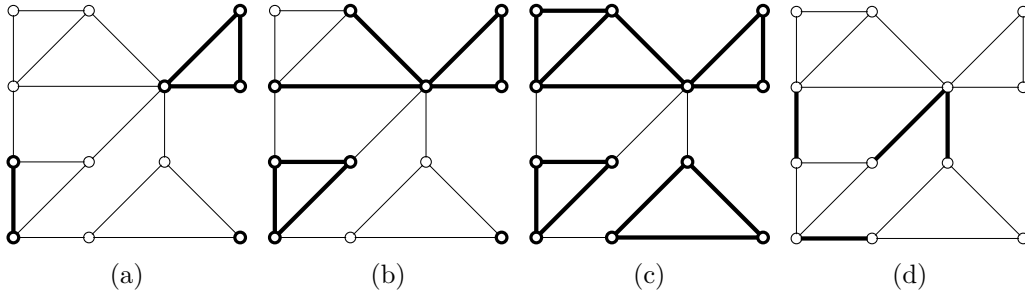


Figure 4: A graph G where in bold there is: (a) a subgraph X of G . (b) an extension of X . (c) an extension Y of X which is maximal. (d) a cut S for X such that $\overline{S} = Y$.

3.2 Watersheds and catchment basins

The notion of a watershed can be defined by its catchment basins or by the properties of the divide line. Usually the first approach is used on a vertex-weighted graph framework, and the catchment basin of a minimum M is defined as the points from which M is reached by a steepest descent path. Following this definition, several catchment basins may overlap each other, as some points could have a steepest descent path to different minimas. To avoid this situation, some authors define the catchment basins as the points from which only a unique minimum is reached by a path with steepest descent, but in this latter definition some sets of points appear not belonging to any catchment basin.

From the notions of extension and graph cut we will introduce the watershed cut, as the “lines” that separate catchment basins, and their properties will be formalized by the drop of water principle. Later we will show that the watershed can be defined by the divide lines or the catchment basins. The following definitions are extracted from [5].

Let $\pi = \langle x_0, \dots, x_l \rangle$ be a path in G . Path π is *descending* (for F) if, for any $i \in [1, l-1]$, $F(\{x_{i-1}, x_i\}) \geq F(\{x_i, x_{i+1}\})$.

Definition 3 (drop of water principle). Let $S \subseteq E$. We say that S satisfies the drop of water principle (for F) if \overline{S} is an extension of $M(F)$ and if for any $u = \{x_0, y_0\} \in S$, there exist $\pi_1 = \langle x_0, \dots, x_n \rangle$ and $\pi_2 = \langle y_0, \dots, y_m \rangle$ which are two descending paths in \overline{S} such that:

- x_n and y_m are vertices of two distinct minima of F ; and
- $F(u) \geq F(\{x_0, x_1\})$ (resp. $F(u) \geq F(\{y_0, y_1\})$), whenever π_1 (resp. π_2) is not trivial.

If S satisfies the drop of water principle, we say that S is a watershed cut, or simply a watershed of F .

We will illustrate the previous definition with the graph G and the map F depicted in Fig. 5. In Fig. 5(a) it can be observed the set that composes the minima $M(F)$ (in bold). Consider the set of edges S at Fig. 5(b). It can be seen that \overline{S} (Fig. 5(c)) is an extension of $M(F)$. Let $s_0 = \{e, f\} \in S$ the edge at altitude 6 at the corner bottom left of the graph G . There exists a descending path $\pi_1 = \langle f, k, o \rangle$ with o a vertex of the minima at altitude 1. Also there is the path $\pi_2 = \langle e, a \rangle$ with a a vertex of a different minima at altitude 3. The first edge of π_1 (resp. π_2) is lower than s_0 , $F(s_0) > F(\{f, k\})$ (resp. $F(s_0) > F(\{e, a\})$). As this properties can be verified for each edge in S , S satisfies the drop of water principle. Thus S is a watershed cut of F .

In the framework of edge-weighted graph, we define a *catchment basin* as a component of the complementary of a watershed. In the following we will show that the watershed can be defined equivalently by the catchment basins or by the divide lines. We will introduce the notion of a path with steepest descent in order to later establish the consistency of the watershed cuts.

From now on, we also denote by F the map from V to \mathbb{R} such that for any $x \in V$, $F(x)$ is the minimal altitude of an edge which contains x .

Let $\pi = \langle x_0, \dots, x_l \rangle$ be a path in G . The path π is a *path with steepest descent* for F if, for any $i \in [1, l]$, $F(\{x_{i-1}, x_i\}) = F(x_{i-1})$.

Definition 4 (steepest descent). Let $S \subseteq E$ be a cut for $M(S)$. We say that S is a basin cut of F if, from each point of V to $M(F)$, there exists, in the graph induced by \overline{S} , a path with steepest descent for F .

Theorem 1 (consistency). Let $S \subseteq E$. The set S is a basin cut of F if and only if S is a watershed cut of F .

Consider the three components of the set depicted in bold in Fig. 5(c). It can be seen that there is a path with steepest descent from each vertex V to the minima, thus these

X if:

i) Y is an extension of X ; and

ii) for any extension $Z \subseteq Y$ of X , we have $Z = Y$ whenever $V(Z) = V(Y)$.

We say that Y is a *spanning forest relative to X (for G)* if Y is a forest relative to X and $V(Y) = V$.

Usually the notion of forest is defined as a graph which does not contain any cycle.

Let X a subgraph of G , the weight of X (for F) is the value $F(X) = \sum_{u \in E(X)} F(u)$.

Definition 5. Let X and Y be two subgraphs of G . We say that Y is a minimum spanning forest (MSF) relative to X (for F , in G) if Y is a spanning forest relative to X and if the weight of Y is less than or equal to the weight of any other spanning forest relative to X . In this case, we also say that Y is a relative MSF.

The minimum spanning forest (MSF) relative to the minima (depicted in bold in fig. 5(a)) is depicted in bold in the graph G of fig. 5(d).

The following statement will help to intuitively establish later the optimality of watersheds.

Let X be a spanning forest relative to $M(F)$. The graph X is a MSF relative to $M(F)$ if and only if, for any x in V , there exists a path in X from x to $M(F)$ which is a path with steepest descent for F .

Let X be a subgraph of G and let Y be a spanning forest relative to X . There exists a unique cut S for Y and this cut is also a cut for X . We say that this unique cut is the *cut induced by Y* . Furthermore, if Y is a MSF relative to X , we say that S is a *MSF cut for X* .

Theorem 2 (optimality). Let $S \subseteq E$. The set S is a MSF cut for $M(F)$ if and only if S is a watershed cut of F .

In fig. 5(d) dashed edges represent the cut induced by the MSF, which is also the watershed of G (fig.5(b)).

3.3.1 Minimum spanning forests and minimum spanning trees

A tree is usually defined [21] as a connected graph with no circuits, and a spanning tree of a connected graph G is a tree in G which contains all nodes of G . If we define the weight of a tree as the sum of the weights of its constituent edges, the minimum spanning tree of a graph G is a spanning tree whose weight is minimum among all spanning trees of G .

We can derive a definition of the tree from the notion of forest.

Let $X \subseteq G$. We say that X is a *tree* (resp. *spanning tree*) if X is a forest (resp. spanning forest) relative to the subgraph $(\{x\}, \emptyset)$, x being any vertex of X .

Let $X \subseteq G$. The graph X is a *minimum spanning tree (for F , in G)* if X is a MSF relative to the subgraph $(\{x\}, \emptyset)$, x being any vertex of X .

We will present a construction that allows to give an equivalence of finding the minimum spanning tree and finding a MSF rooted on any subgraph X of G .

Let us consider first a subgraph $X \subseteq G$ composed of isolated vertices. Let us construct a new graph $G' = (V', E')$ which contains an additional vertex $z \notin V$ linked by an edge to each vertex of X , thus $V' = V \cup \{z\}$ and $E' = E \cup E_z$, where $E_z = \{\{x, z\} | x \in V(X)\}$. Let us consider the map F' from E' to \mathbb{R} such that, for any $u \in E$, $F'(u) = F(u)$, while for any $u \in E_z$, $F'(u) = k_{\min} - 1$, where k_{\min} is the minimum value of F .

Let Y be any subgraph of G and let Y' be the graph such that $V(Y') = V(Y) \cup \{z\}$ and $E(Y') = E(Y) \cup E_z$. It may be seen that Y' is a minimum spanning tree for F' in G' if and only if Y is a MSF relative to X for F in G .

Following this construction we can extend it to the case of X being any subgraph of G . In this case, first we will contract each component of X into a single vertex, and if two vertices of this resulting graph must be linked by more than one edge we will keep only the one with minimal value. After we will proceed with the construction explained above.

The minimum spanning tree computational construction has been widely studied. Considering this construction the relative MSF can be computed using any minimum spanning tree algorithm.

3.4 Watershed algorithm

In order to introduce the method used to compute the watershed we will define first an edge classification based on local properties. Then we will also define the lowering operation and the thinning that will let us understand the strategy used to compute the watershed.

As we said before if x is a vertex of G , $F(x)$ is the minimal altitude of an edge at x .

Definition 6. Let $u = \{x, y\} \in E$.

We say that u is locally separating (for F) if $(F(u) > \max(F(x), F(y)))$.

We say that u is border (for F) if $(F(u) = \max(F(x), F(y)))$ and $(F(u) > \min(F(x), F(y)))$.

We say that u is minimum-border (for F), written M-border if u is border and if exactly one of the vertices in u is a vertex of $M(F)$.

We say that u is inner (for F) if $F(x) = F(y) = F(u)$.

Fig. 6 illustrates this classification. In fig. 5(a) edges $\{k, f\}$ or $\{l, g\}$ are examples of border edges, $\{e, f\}$ or $\{m, i\}$ are locally-separating, $\{k, o\}$ or $\{q, n\}$ are m-border, and $\{a, b\}$ or $\{w, q\}$ are examples of inner edges.

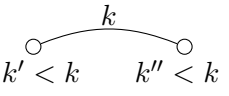
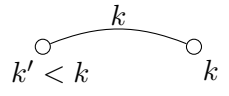
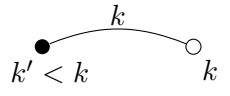
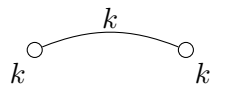
locally-separating	border	m-border	inner
			

Figure 6: Edge classification in a weighted graph. In the m-border case the black vertex means that belongs to a minimum.

Definition 7. Let $u \in E$. The lowering of F at u is the map in \mathcal{F} , denoted by $[F \setminus u]$, such that:

$$\begin{aligned} &-[F \setminus u](u) = \min_{x \in u} \{F(x)\}; \text{ and} \\ &-[F \setminus u](v) = F(v) \text{ for any edge } v \in E \setminus \{u\}. \end{aligned}$$

Definition 8 (border cut (and M-border cut)). *Let $H \in \mathcal{F}$. We say that H is a border thinning of (resp. M-border thinning of) F if:*

i) $H = F$; or

ii) there exists $I \in \mathcal{F}$ a border thinning of F (resp. M -border thinning of F) such that H is the lowering of I at a border (resp. M -border) edge for I .

If there is no border (resp. M -border) edge for H , we say that H is a border kernel (resp. M -border kernel). If H is a border thinning (resp. M -border thinning) of F and if it is a border kernel (resp. M -border kernel), we say that H is a border kernel of F (resp. border kernel of F).

If H is a border kernel (resp. M -border kernel) of F , any cut for $M(H)$ is called a border cut for F (resp. M -border cut for F).

Consider the illustrations at fig. 7. The map H of fig. 7(b) is a border thinning of the map F of fig. 7(a). It is also a M-border thinning as the edge $\{d, e\}$ lowered has the vertex $d \in M(F)$. The map I of fig. 7(c) depicts a M-border kernel of F . Edges in dashed show the M-border cut for F induced by $M(I)$.

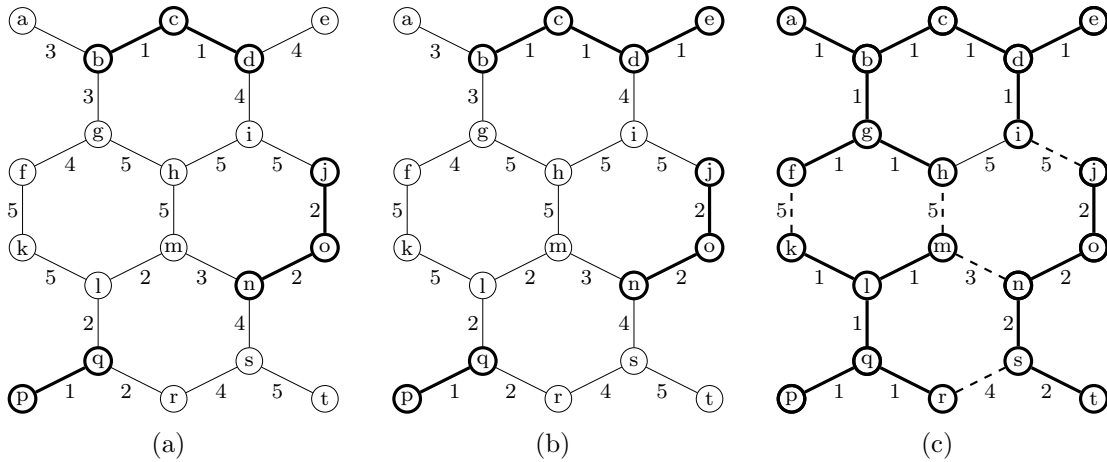


Figure 7: Graphs where the minima of the corresponding functions is depicted in bold. (b) A M-border thinning of (a). (c) An M-border kernel I of (a), where dashed edges correspond to the cut induced by $M(I)$.

It can be proved [5] that the border thinning transformation preserves some MSF relative to the minima of the original map. The border kernels allow the extraction of the MSF relative to the minima, as the minima of a border kernel is itself a MSF relative to the minima of the original map. As we state above (theorem 2) a MSF relative to the minima induces a unique cut that is a watershed cut of the original map. Thus, obtaining

a border kernel, as it is a sequence of local operations, is a promising approach to produce a globally optimal structure as the MSF relative to the minima, hence the watershed cut.

A possible algorithm to compute the border kernel would be: 1) take an edge of the graph, check if it is a border, and lower its value; 2) repeat step 1) until no border edge remains. Due to the cost of this approach, the particular case of border kernel, M-border kernel, is considered. As an edge which is in a minimum in a given step of the border thinning sequence, will never become border again in later steps, the strategy will be to lower first the edges adjacent to the minima.

As well as the minimum of a border kernel of a map F is a MSF relative to $M(F)$, the minimum of a M-border kernel is also a MSF relative to $M(F)$. Hence, it can be stated the following theorem.

Theorem 3. *Let $S \subseteq E$. The following statements are equivalent:*

- (i) S is a M-border cut for F ;
- (ii) S is a border cut for F ;
- (iii) S is a watershed cut for F ;

The following algorithm presented by Cousty [5] computes the M-border kernel, hence the watershed cut, using these previous notions.

Algorithm 1: M-Border

Data: (V, E, F) - edge-weighted graph

Result: F , a M-border kernel of the input map and M its minima.

```

1  $L \leftarrow \emptyset$ ;
2 Compute  $M(F) = (V_M, E_M)$  and  $F(x)$  for each  $x \in V$ ;
3 foreach  $u \in E$  outgoing from  $(V_M, E_M)$  do  $L \leftarrow L \cup \{u\}$ ;
4 while there exists  $u \in L$  do
5    $L \leftarrow L \setminus \{u\}$ ;
6   if ( $u$  is border for  $F$ ) then
7      $x \leftarrow$  the vertex in  $u$  such that  $F(x) < F(u)$ ;
8      $y \leftarrow$  the vertex in  $u$  such that  $F(y) = F(u)$ ;
9      $F(u) \leftarrow F(x)$ ;  $F(y) \leftarrow F(u)$ ;
10     $V_M \leftarrow V_M \cup \{y\}$ ;  $E_M \leftarrow E_M \cup \{u\}$ ;
11    foreach  $v = \{y', y\} \in E$  such that  $y' \notin V_M$  do  $L \leftarrow L \cup \{v\}$ ;
```

3.5 Border thinning on simplicial complexes

In this section we will introduce some notions on operators defined in [3] on the framework of simplicial complexes, and then we will be able to extend the notions introduced for border thinnings on edge weighted graphs into the case of simplicial complexes. Let us first introduce some notions for complexes, not considering weights in their faces for the moment.

Let X be a complex in \mathbb{K} and let $f \in X^+$. The face f is a *border face* for X if there exists one face $g \in \hat{f}^*$ such that f is the only face of X which contains g . Such a face g is said to be *free for X* and the pair (f, g) is said to be a *free pair for X* . We say that $f \in X^+$ is an *interior face* for X if f is not a border face.

Let X be a complex, and let (f, g) be a free pair for X . The complex $X \setminus \{f, g\}$ is an *elementary collapse* of X .

We illustrate in fig. 8(a) a complex X where g is a border face of X and j is a free face of X . Fig. 8(b) depicts the set $X \setminus \{g, j\}$ an elementary collapse of X .

Let us consider weights on the 1-faces of a complex. Let $X \preceq \mathbb{K}$, we denote by A the set of all 1-faces of X , and T the set of all 2-faces of X . Let us denote by \mathcal{F} the set of all maps from A to \mathbb{R} . Let $F \in \mathcal{F}$, we will denote also by F the map from T to \mathbb{R} such that for any $g \in T$, $F(g)$ is the minimal weight of its 1-faces, i.e $F(g) = \min\{F(u) | u \in A, u \in \hat{g}^*\}$.

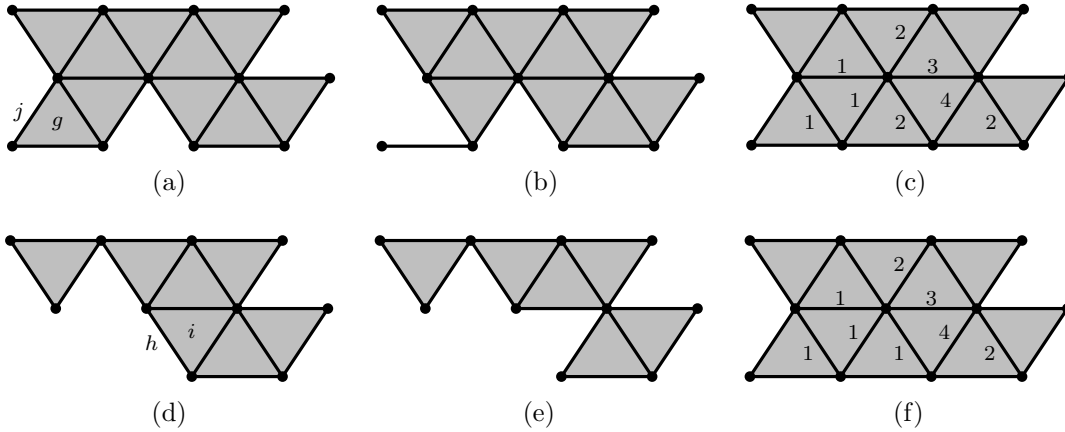


Figure 8: (a) A complex X and a free pair (j, g) for X . (b) An elementary collapse for X . (c) A weighted complex K with the map F on 1-faces (consider the 1-faces with no weight that have a weight greater than 4). (d) The section F_2 of F . (h, i) is a free pair of F_2 . (e) An elementary collapse for F_2 . (f) A border thinning of K .

We will define also for weighted complexes the lowering operation on 1-faces, as presented for edge weighted graphs in def. 7.

Let $u \in A$. The *lowering of F at u* is the map in \mathcal{F} , denoted by $[F \setminus u]$, such that:

- $[F \setminus u](u) = \min\{F(g), F(h) | g \cap h = u\}$; and
- $[F \setminus u](v) = F(v)$ for any 1-face $v \in A \setminus \{u\}$.

We denote $F_k = \{u \in A, g \in T | F(u) \geq k, F(g) \geq k\}$ with $k \in \mathbb{R}$, and we called it a *section* of F .

Consider X be a complex in \mathbb{K} and $F \in \mathcal{F}$ the weights on the 1-faces of X . Let F_k be a section of F . We define also a 1-face $u \in A$ a *border 1-face for F* if u is a free face for F_k with $k = F(u)$. It can be seen an elementary collapse of a section F_k as a lowering operation on a border 1-face for F . Thus, we can observe an equivalence with the notions explained for border thinnings on edge-weighted graphs. Furthermore, this justify our choice of the vertex per-face graph as a representation for the mesh to proceed in a proper segmentation.

We illustrate with an example these previous notions. Consider the complex K depicted in fig. 8(c) with a map F . The figure only shows values on 1-faces which are lower than 4. 2-faces have the values corresponding to the minimal weight of its 1-faces. We illustrate in fig. 8(d) the section F_2 of F , where h is a border 1-face for F , as it is a free face for F_2 and $F(h) = 2$. Fig. 8(e) depicts an elementary collapse of F_2 , and fig. 8(f) depicts the lowering on the border 1-face h . In can be observed the equivalence with the border thinning explained in def. 8 applied on the vertex per-face graph of K .

4 Component tree

The *component tree* is a tree structure used to organize the connected components of a level function. It has been used for several image processing applications, although it was first introduced in statistics for classification and clustering. Different variations have been implemented as the Max-Tree introduced by Salembier et al. [19] used as a data structure for antiextensive connected operators, which is analysed and improved by Meijster and Wilkinson [12]. The algorithm we implemented is based on the one described by Najman and Couprie [14].

We will describe briefly the building process of the tree in order to give an idea of the properties of this structure, and later it will be introduced formally. The tree can be build as a Max-Tree (focusing on regional maxima) or a Min-Tree (focusing on regional minima), and for our purposes the Min-Tree approach has been used. Consider a discrete map as a topographical relief with the level of each point corresponding to its altitude. We will start flooding by water this surface starting at the lowest points. At beginning there will appear various lakes that will form the leafs of the tree. As water level increases the lakes will grow building the branches of the tree. By the time at some levels the lakes will merge into one connected piece becoming the forks of the tree. When the water reaches the highest level the process stops and the flooded area forms a unique component that is the root of the tree.

The tree can be used for filtering the original level function to obtain, for example as in our purposes, a new map with a reduced number of regional minima. Some attributes will be computed for each leaf and branches of the tree, and they will set an order of preference. As in the analogy explained before, the smaller valleys (considering their area for example)

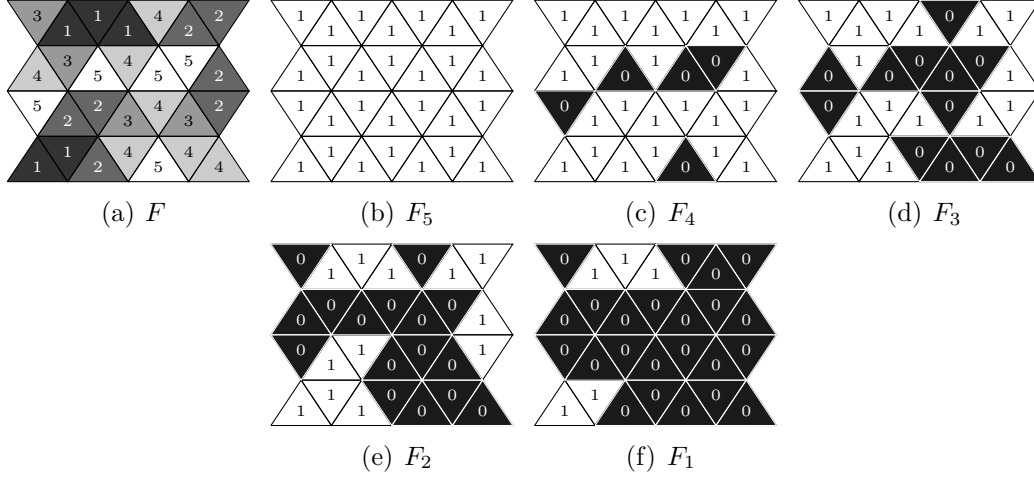


Figure 9: A weighted face set F and its cross-sections at levels 5, 4, 3, 2, 1 (in white).

of the topographical relief will be removed. Hence, removing the corresponding branches of the tree.

We implemented a quasi-linear time algorithm for computing the component tree of functions defined on edge-weighted graphs that is based on the Tarjan's union-find procedure [20]. We will define the component tree on the framework of graphs, and it will be illustrated with the case of a vertex-per-face graph on a complex as described in section 2.4. We will explain the union-find method and after introduce the algorithm to build the tree. Later we will describe the methods used for filtering the mesh curvatures values using the component tree in order to obtain the desired watershed segmentation.

4.1 Connected components notions

We introduce some notions and notations for connected components in weighted graphs in order to define properly the component tree in the following section.

We denote by \mathcal{F} the set of all maps from E to \mathbb{R} . For a map $F \in \mathcal{F}$ let us consider (V, E, F) an edge-weighted graph. We also denote by F the map from V to \mathbb{R} such that for any $x \in V$, $F(x)$ is the minimal altitude of an edge which contains x . We define $F_k = \{u \in E | F(u) \leq k\}$ with $k \in \mathbb{R}$; F_k is called a *(cross-)section* of F . It can be noticed that for any $u \in F_k$ and x, y the vertices at u , also $F(x) \leq k, F(y) \leq k$. A connected component of a section F_k is called a *(level k) component* of F . A level k component of F that does not contain the level $(k - 1)$ is called a *(regional) minimum* of F . We define $k_{min} = \min\{F(u) | u \in E\}$ and $k_{max} = \max\{F(u) | u \in E\}$, which represent respectively the minimum and the maximum level in the map F .

We illustrate this notions on the framework of triangular face sets with fig. 9 where the different cross-sections can be observed. In this case we have weights on faces, and their connectivity is determined by the vertex per-face graph explained in section 2.4.

4.2 Component tree definition

The following definition for the component tree is extracted from [14].

Let $F \in \mathcal{F}$. For any component c of F we set $h(c) = \min\{k | c \text{ is a level } k \text{ component of } F\}$. Note that $h(c) = \max\{F(x) | x \in c\}$. We define $\mathcal{C}(F)$ as the set composed of all pairs $[k, c]$, where c is a component of F and $k = h(c)$. We call *altitude* of $[k, c]$ the number k . Remark that any two distinct elements of $\mathcal{C}(F)$ correspond to distinct subgraphs.

Let $F \in \mathcal{F}$, let $[k_1, c_1], [k_2, c_2]$ be distinct elements of $\mathcal{C}(F)$. We say that $[k_1, c_1]$ is the *parent* of $[k_2, c_2]$ if $c_2 \subset c_1$ and if there is no other $[k_3, c_3]$ in $\mathcal{C}(F)$ such that $c_2 \subset c_3 \subset c_1$. In this case we also say that $[k_2, c_2]$ is a *child* of $[k_1, c_1]$. With this relation “parent”, $\mathcal{C}(F)$ forms a directed tree that we call the *component tree of F* , and that we also denote by $\mathcal{C}(F)$. Any element of $\mathcal{C}(F)$ is called a *node*. An element of $\mathcal{C}(F)$ which has no child is called a *leaf*, the node which has no parent is called the *root*.

We define the *(vertex) component mapping CM* as the map from V to $\mathcal{C}(F)$ which associates to each vertex $p \in V$ the node $CM(p)$, such that the altitude of $CM(p)$ is $F(p)$ and $p \in CM(p)$. We also define the *edge component mapping CME* as the map from E to $\mathcal{C}(F)$ which associates to each edge $u \in E$ the node $CME(u)$, such that the altitude of $CME(u)$ is $F(u)$ and $u \in CME(u)$.

Fig. 10(a) shows the component tree of the vertex per-face graph of F the face weighted set of fig. 9 and fig. 10(b) shows the associated component mapping depicted on the faces of the set F .

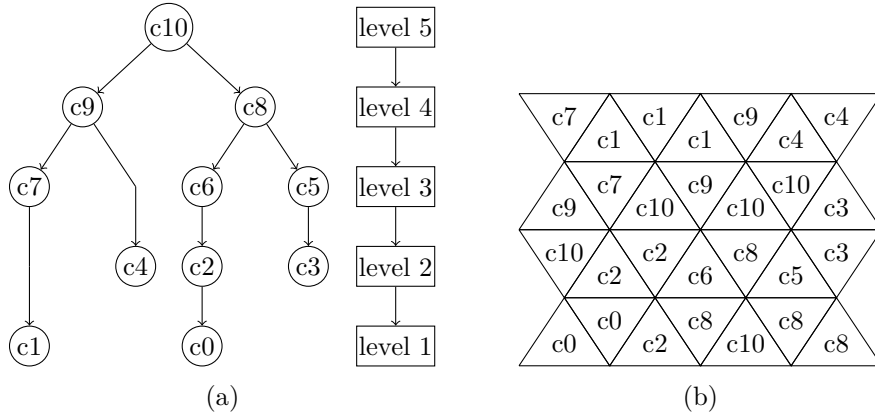


Figure 10: The component tree (a) of the vertex per-face graph of the face weighted set of fig.9 and its associated component mapping (b).

4.3 Component tree and minimum spanning tree

As our work is settled on edge-weighted graphs, some considerations will be introduced in order to understand the algorithm for the construction of the component tree implemented, and its differences with the previous algorithms [14] [19].

We present the notion of line graph that will allow us to apply the notions for vertex-weighted graph on edge-weighted graphs.

Definition 9. *The line graph of $G = (V, E)$ is the graph (E, Γ) , such that $\{u, v\}$ belongs to Γ whenever $u \in E$, $v \in E$, and u and v are adjacent, i.e., $|u \cap v| = 1$.*

We can associate to each graph G whose edges are weighted by a cost function F , a line graph G' . The vertices of G' are weighted by F and thus any transformation can be performed either in G or in G' .

We could apply the algorithm introduced by Najman [14], which focus on vertex-weighted graphs, on an edge-weighted graph G , by using the line graph of G , G' . Then, once the component tree would be built, we would obtain the component mapping of the vertices E of G' . Thus we would have the component mapping on the edges E of G . Then, by removing nodes of the tree, we would reduce the number of minima on the edge-weighted graph G .

The purpose of the filtering of a map F on edges of a graph G is to reduce the over-segmentation produced by the watershed operator. Considering this purpose our component tree will be computed only on the minimum spanning tree edges of the graph. On the following of this section we will explain this consideration.

As introduced in section 3.3 the watershed cut of F is induced by the minimum spanning forest relative of the minima of the map F . Furthermore, in section 3.3.1 we derived that the minimum spanning tree of a graph X is a MSF relative to the subgraph $(\{x\}, \emptyset)$, x being any vertex of X .

Following the construction presented in section 3.3.1 we can state also the following. Let Y be the MSF relative to $M(F)$ for F in G . For any edge $u \in Y \setminus M(F)$ it can be observed that u belongs to a minimum spanning tree for F in G .

We will give some notations concerning minimum spanning trees (MST) in order to introduce later a theorem that will help us understand the approach adopted to construct the component tree.

Let us define a *partition* of the vertices of a graph G as a division into two disjoint non-empty subsets of vertices (P, Q) . The *distance* $\rho(P, Q)$ across a partition is the smallest weight among all edges which have one vertex in P and other in Q . The *cut-set* $C(P, Q)$ is the set of edges that span a partition (P, Q) . A *link* is any edge in $C(P, Q)$ whose weight is equal to the distance $\rho(P, Q)$, while the set of all links in $C(P, Q)$ is called the *link-set* $\lambda(P, Q)$.

A main theorem concerning a MST is the following.

Theorem 4. *Any MST contains at least one edge from each $\lambda(P, Q)$.*

Let Y be a MSF relative to $M(F)$ for F in G , and X be any component of Y . Let P be the set $V(X)$ and Q the set $V(Y) \setminus V(X)$. The cut-set $C(P, Q)$ thus, belongs to the basin cut of F . By the theorem stated above it may be seen that the MST of F will contain the edge with lowest weight of $C(P, Q)$.

Thus, by this latter considerations the MST will contain all edges of the MSF relative to $M(F)$ excluding the edges of $M(F)$ that make a cycle. Also, considering S the basin cut of F , the MST will contain any edge $u \in S$ such that u is the lowest edge outgoing from a component of \overline{S} .

Consider we obtain the component tree of G taking into account only the edges of the MST. Then we filter the map F by removing a branch of the tree. Hence we give at the nodes of the branch the level value of the parent of the highest node of branch. Let us call X the subgraph corresponding to the removed component. After the filtering, the edges not considered in the component tree, *i.e* edges that not belong to a MST, will get a ∞ value. We call this filtered map F' .

Consider now S the watershed cut of F , and S' the watershed cut of F' . Consider Y a component of \overline{S} (a catchment basin). As the component X forms a branch of the tree, $X \subseteq Y$. We give to each edge of X the weight of the edge $u \in S$, such that $F(u)$ is the lowest cost from the ones outgoing from Y , and also the edges of $Y \setminus X$ get ∞ value. For this two latter reasons, there is no descending path from u in Y for F' to $M(F')$. Thus, Y is not a catchment basin for F' .

Our algorithm for the component tree is then based on a classical algorithm for the minimum spanning tree presented by Kruskal [9]. It consists in sorting first the edges of the graph by increasing order of their weight, and then selecting them, making sure to select only edges that do not form a circuit with the previously selected. In our case, during this stage we will proceed on the building of the component tree.

4.4 The Union-Find method

The Union-Find algorithm proposed by Tarjan allows keeping track of disjoint sets, performing three useful operations to manage a collection \mathcal{Q} of disjoint sets. Each set is represented by an arbitrary member called the *canonical element*. The algorithm uses rooted trees to represent sets in which the root is chosen as the canonical element. Two objects x and y are members of the same set if and only if x and y are nodes of the same tree, which is equivalent to saying that they share the same root of the tree they are stored in. The collection is managed by the following operations:

- **MakeSet(x)**: add the set $\{x\}$ to collection \mathcal{Q} . This operation assumes that x is not a member of any other set in \mathcal{Q} .
- **Find(x)**: return the canonical element of the set in \mathcal{Q} which contains x .
- **Link(x,y)**: let X and Y be two non empty set in \mathcal{Q} represented by x and y (x and y must be different). Both sets are removed from \mathcal{Q} , their union $Z = X \cup Y$ is added to \mathcal{Q} and a canonical element for Z is selected and returned.

The implementation of the algorithm is the one presented at procedure **MakeSet** and functions **Find** and **Link**. To each element x is associated a parent $\text{Par}(x)$ and a rank

$\text{Rnk}(x)$. A technique used to reduce the run cost of the function **Find** called *path compression* consists, after finding the root r of a tree which contains x , in setting the parent of each element of the parent path from x to r to be r . Another technique used is the one called *union by rank*. The rank $\text{Rnk}(x)$ is incremented by one if x becomes parent of y . The technique consists in always choosing the representing element with the greatest rank while performing the **Link** operation. If both elements have the same rank one of them is chosen arbitrarily.

Procedure MakeSet(*element* x)

1 Par(x):=x; Rnk(x) :=0;

Function element Find(*element* x)

1 **if** ($\text{Par}(x) \neq x$) **then** Par(x) := Find(Par(x));
2 **return** Par(x);

Function element Link(*element* x , *element* y)

1 **if** ($\text{Rnk}(x) > \text{Rnk}(y)$) **then** exchange(x,y);
2 **if** ($\text{Rnk}(x) == \text{Rnk}(y)$) **then** Rnk(y) := Rnk(y) + 1;
3 Par(x):=y;
4 **return** Par(x);

4.5 Component tree algorithm

In this section we explain the algorithm implemented to build the component tree $\mathcal{C}(F)$ on an edge-weighted graph in the context of a vertex per-face graph on a mesh surface. The algorithm is based on the one presented by Najman and Couprie with some modifications to better perform on edge-weighted graphs as explained in sec. 4.3. We first will describe the algorithm from a high point of view and later some details will be explained.

4.5.1 High-level view

The process will simulate a flooding as described before. This flooding is realized by scanning the edges of the graph by increasing order of their level. Two disjoint set collections \mathcal{Q}_{node} and \mathcal{Q}_{tree} will be used to manage the merging of nodes and branches of the tree. As edges are scanned by level increasing order, the vertices first time visited will get the minimum weight value of their adjacent edges. The elements needed to manage the collections \mathcal{Q}_{node} and \mathcal{Q}_{tree} are build during the process. The \mathcal{Q}_{node} collection will group vertices and edges belonging to the same connected component and having the same altitude. Simultaneously the \mathcal{Q}_{tree} collection will group the canonical nodes of each \mathcal{Q}_{node} node to form

partial trees. At the end of the execution a unique tree groups all the canonical nodes, each of the nodes represents a component of the graph, and the whole tree represents the component tree.

4.5.2 Detailed view

The algorithm for the component tree computation is presented below **BuildComponentTree**, also with the complementary functions used, **InitNode** and **MergeNodes**. To represent a node of $\mathcal{C}(F)$ we use a structure called **node** that contains its level and the list of nodes which are sons of this node. In a preprocessing step edges are sorted by increasing order of level (line 1). Then the process starts (line 3) proceeding by taking each edge of the graph starting by the lowest one. We use a label *nblabs* and the array *CM* (used for the resulting component mapping) to proceed properly on the generation of new nodes of $\mathcal{C}(F)$ during the processing step.

The function **InitNode** creates new elements of collections \mathcal{Q}_{node} and \mathcal{Q}_{tree} using the procedure **MakeSet** described in section 4.4. It also creates a new **node** structure.

The array *highestNode* will help us to know the node with highest level from a partial tree during the processing, and it is necessary due to the particularities of the union-find algorithm.

From the algorithm we also obtain the edge component mapping *CME*. We do not describe this array on the text presented in **BuildComponentTree** but we give the precise assignments below.

The **BuildComponentTree** procedure have three main possible cases, which depend on the previously visited vertices. To know if a vertex has already been visited we use the array *CM* set at an initial value for all vertices, in a preprocessing step. Considering $u \in E$ the edge chosen and $(x, y) \in u$ the vertices at this edge, the three possibilities are:

- both vertices are visited for the first time; in this case (lines 4-6), as we proceed on edges by increasing order of their level, and vertices have the minimal value of their adjacent edges, both vertices have the same level and it equals the edge level $F(u)$. Thus, a new node of level $F(u)$ is build (function **InitNode**) and the vertices x, y will belong to this node, so the component mapping of x and y is set to the new label *nblabs*. The edge component mapping *CME* is set to the same label *nblabs* for u .
- one vertex has been visited and the other is visited for the first time; in this case (lines 7-13) we use the array *CM* to know which node the vertex x belongs to (x is the vertex already visited). Using then the function **Find** on the collections \mathcal{Q}_{node} and \mathcal{Q}_{tree} we get its canonical node x_{node} (line 8).

If the edge level is the same as x_{node} level, it means that the other vertex y should have this level also, and it belongs to the same node (line 9). $CME(u)$ is set to x_{node} .

In the other case if $F(u)$ is greater than the level of x_{node} , a new node it is created, and x_{node} (the node that represents x) becomes its son (lines 11-12). The elements of \mathcal{Q}_{tree} are linked to form a partial tree. The array *highestNode* is modified to set the

new node as the highest on level of the partial tree recently build (line 13). $CME(u)$ is set to the new node label.

- both vertices have been visited before; in this case we first find the canonical nodes that represent x and y (lines 15-16). In case that they are not in the same partial tree (line 17) we proceed differently depending on their levels.

If their level it is the same, nodes are merged (function **MergeNodes**)(line 22). $CME(u)$ is set to the canonical node resulting from the merging.

If the edge level is equal than one of the nodes but greater than the other, the bigger becomes parent (line 24). $CME(u)$ is set to the parent node.

If the edge level is greater than both nodes a new node is created, that becomes parent of both (lines 27-29). $CME(u)$ is set to the parent node.

The partial trees are linked for this three preceeding cases.

The root of the component tree is found as the highest node of the resulting tree of the collection \mathcal{Q}_{tree} (line 33), while the component mapping is the resulting array CM .

Function node **InitNode**(*int label, double level*)

```

1 Allocate a new node n with an empty list of children;
2 n → level := level;
3 MakeSettree(label); MakeSetnode(label);
4 return n;
```

Function int **MergeNodes**(*int node1, int node2*)

```

1 tmpNode := Linknode(node1, node2);
2 if (tmpNode == node2) then
3   | Add the list of childrens of nodes[node1] to the list of children of nodes[node2];
4 else
5   | Add the list of childrens of nodes[node2] to the list of children of nodes[node1];
6 return tmpNode;
```

Algorithm 7: BuildComponentTree

Data: (V, E, F) - edge-weighted graph with N vertices
Result: *nodes* - array $[0 \dots N - 1]$
Result: *Root* - Root of the component tree
Result: *CM* - map from V to $[0 \dots N - 1]$ (component mapping)
Local: *highestNode* - map from $[0 \dots N - 1]$ to $[0 \dots N - 1]$

- 1 Sort the edges in increasing order of level for F ;
- 2 $nblabs := 0$;
- 3 **foreach** $u \in E$ in increasing order of level for F **do**
 - 4 // $u = (x, y) | x, y$ are vertices
 - 5 **if** (both vertices are visited for the first time) **then**
 - 6 $nodes[nblabs] := \text{InitNode}(nblabs, F(u))$; $CM(x) := nblabs$; $CM(y) := nblabs$;
 - 7 $highestNode[nblabs] := nblabs$; $nblabs++$;
 - 8 **else if** (one vertex has already been visited and the other not) **then**
 - 9 // considering x the vertex already visited
 - 10 $x_{tree} := \text{Find}_{tree}(CM(x))$; $x_{node} := \text{Find}_{node}(highestNode[x_{tree}])$;
 - 11 **if** ($F(u) == nodes[x_{node}] \rightarrow level$) **then** $CM(y) := x_{node}$;
 - 12 **else if** ($F(u) > nodes[x_{node}] \rightarrow level$) **then**
 - 13 $nodes[nblabs] := \text{InitNode}(nblabs, F(u))$; $CM(y) := nblabs$;
 - 14 $nodes[nblabs] \rightarrow \text{addChild}(nodes[x_{node}])$;
 - 15 $highestNode[\text{Link}_{tree}(x_{tree}, \text{Find}_{tree}(nblabs))] := nblabs$; $nblabs++$;
 - 16 **else if** (both vertices already been visited) **then**
 - 17 $x_{tree} := \text{Find}_{tree}(CM(x))$; $x_{node} := \text{Find}_{node}(highestNode[x_{tree}])$;
 - 18 $y_{tree} := \text{Find}_{tree}(CM(y))$; $y_{node} := \text{Find}_{node}(highestNode[y_{tree}])$;
 - 19 **if** ($x_{node} \neq y_{node}$) **then**
 - 20 **if** ($nodes[x_{node}] \rightarrow level < nodes[y_{node}] \rightarrow level$) **then**
 - 21 $temp := y_{node}$; $x_{node} := y_{node}$; $x_{node} := temp$;
 - 22 $temp := y_{tree}$; $x_{tree} := y_{tree}$; $x_{tree} := temp$;
 - 23 **if** ($F(u) == nodes[x_{node}] \rightarrow level == nodes[y_{node}] \rightarrow level$) **then**
 - 24 $highestNode[\text{Link}_{tree}(x_{tree}, y_{tree})] := \text{MergeNodes}(x_{node}, y_{node})$;
 - 25 **else if** ($(F(u) == nodes[x_{node}] \rightarrow level) \&\& (F(u) > nodes[y_{node}] \rightarrow level)$) **then**
 - 26 $nodes[x_{node}] \rightarrow \text{addChild}(nodes[y_{node}])$;
 - 27 $highestNode[\text{Link}_{tree}(x_{tree}, y_{tree})] := x_{node}$;
 - 28 **else if** ($F(u) > nodes[x_{node}] \rightarrow level$) **then**
 - 29 $nodes[nblabs] := \text{InitNode}(nblabs, F(u))$;
 - 30 $nodes[nblabs] \rightarrow \text{addChild}(nodes[x_{node}])$;
 - 31 $nodes[nblabs] \rightarrow \text{addChild}(nodes[y_{node}])$;
 - 32 $highestNode[x_{tree}] := nblabs$; $highestNode[y_{tree}] := nblabs$;
 - 33 $highestNode[\text{Link}_{tree}(\text{Link}_{tree}(x_{tree}, y_{tree}), \text{Find}_{tree}(nblabs))] := nblabs$;
 - 34 $nblabs++$;

- 35 **foreach** $x \in V$ **do** $CM(x) := \text{Find}_{node}(CM(0))$;
- 36 $Root := highestNode[\text{Find}_{tree}(\text{Find}_{node}(CM(0)))]$;

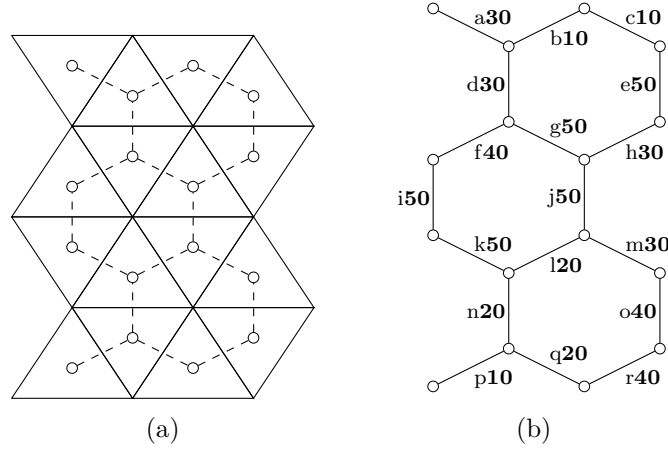


Figure 11: (a) Set of triangular faces and its vertex-per face graph G on dashed. (b) The graph G with weights on edges (in bold) and labels to identify them. Edges will be examined in the following order, $(p, b, c, l, q, n, a, d, m, h, f, o, r, e, g, i, k, j)$

4.5.3 Example

We are going to illustrate on an example how the algorithm works. The example will be settled on the framework of edge weighted graphs, and more precisely on the vertex per-face graph of fig. 11(b) that represents the set of faces of fig. 11(a). The edges of the graph are labeled by letters to refer to them easily later. We will focus on certain steps of the building process of the component tree to describe the details of the algorithm.

Edges are examined by increasing order of their level as follows, $(p, b, c, l, q, n, a, d, m, h, f, o, r, e, g, i, k, j)$, as result of the initial sorting. Suppose that we are already at the beginning of step 6, so we will proceed with edge n . At this moment the partial tree built is the one depicted at fig. 12(b). the node level and the node label are depicted for each node in the form $[k, c]$ where k is the level and c is the label of the component. We represent the arrays Fth_{tree} , Fth_{node} to take into account the situation of the collections \mathcal{Q}_{tree} , \mathcal{Q}_{node} respectively. The arrays order is from left to right and from up to down. The array CM is also represented. Each cell of the array CM represents a vertex of the graph, ordered as vertices disposition on the graph. Thus, the upper left grid cell of the array represents the vertex at left of edge a , while following on the right, the grid cell represent the other vertex of a , and then the vertex between b and c , etc..

When edge n is chosen their both vertices have already been visited. Thus, the Find_{tree} operations are applied on the nodes representing these vertices (by using the array CM), to find their canonical nodes. As their canonical nodes 3 and 2 are at same level 20, and edge n makes them connected, these nodes should be merged (**MergeNodes**). As result of this linking the canonical node of the node 3 for the collection \mathcal{Q}_{node} will change to be 2, and the son of node 3 (node 0) will be now son of 2. Even though, the canonical node of

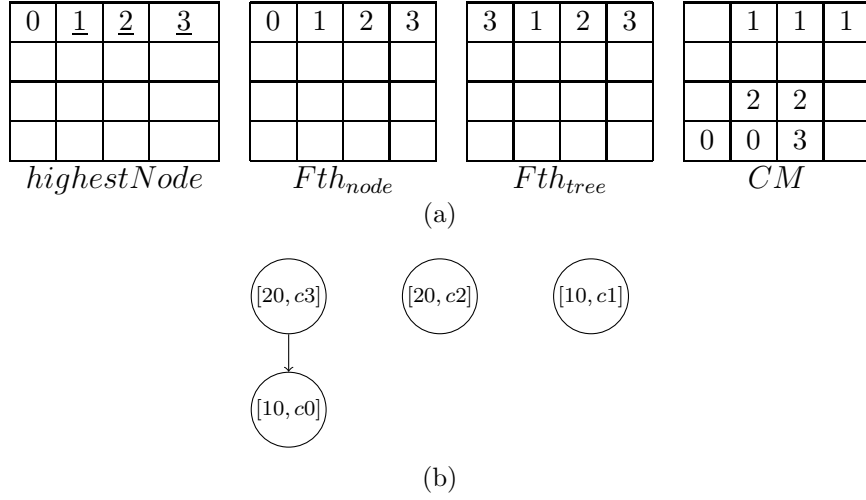


Figure 12: End of step 5. (a) The arrays $highestNode$, Fth_{tree} , Fth_{node} and the component mapping CM . (b) Partial trees already constructed.

nodes 2 and 3 in \mathcal{Q}_{tree} will still be 3 as result of the heuristics of Tarjan algorithm (union by rank, section 4.4). The $highestNode$ array at position 3 is set to 2, as it is the new father. The array $highestNode$ is only used on canonical node positions, which are underlined at the figures.

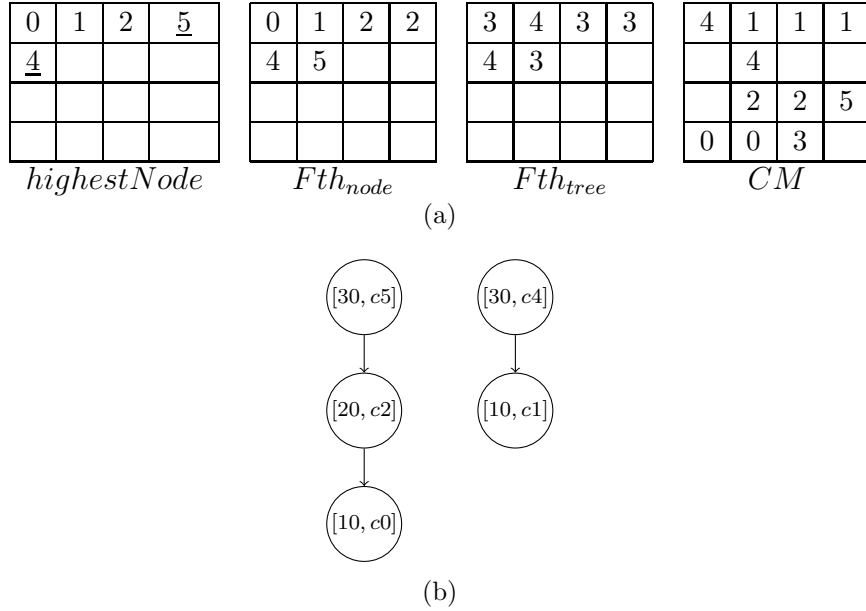


Figure 13: End of step 9. (a) The arrays $highestNode$, Fth_{tree} , Fth_{node} and the component mapping CM . (b) Partial trees already constructed.

After n , and edges a, d, m at level 30 are processed. This situation is depicted at fig. 13. Node 4 became father of node 1 as result of processing for edges a, d . The linking led node 4 being the canonical node of its partial tree. The processing of edge m illustrates the work of the arrays. The component mapping of the already visited vertex at m is 2, while the canonical node of 2 in Q_{tree} is 3 (line 8 of the algorithm). Its highest node at this moment is 2, thus will be the level of node 2 that will be checked. As the level at m is greater, a new node 5 is created, which becomes father. The canonical node remains 3 and *highestNode* at this position is changed to 5.

0	1	2	<u>8</u>
<u>9</u>		6	

highestNode

0	1	2	2
4	5	6	7
8	9		

Fth_{node}

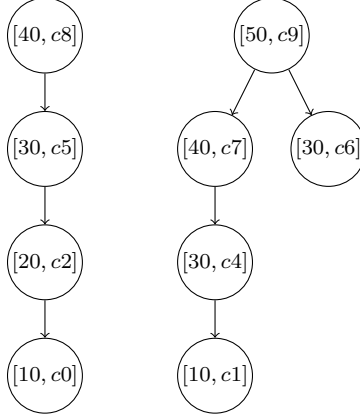
3	4	3	3
4	3	4	4
3	4		

Fth_{tree}

4	1	1	1
7	4	6	6
	2	2	5
0	0	3	8

CM

(a)



(b)

Figure 14: End of step 14. (a) The arrays *highestNode*, *Fth_{tree}*, *Fth_{node}* and the component mapping *CM*. (b) Partial trees already constructed.

It follows the processing of edge h at level 30, that creates a new node 6 that form a new partial tree. Edges f and o at level 40 create new nodes that become parents of the partial trees represented by nodes 3 and 4 respectively. Edge r is not treated as it connects nodes that belong to the same partial tree. Remark that this edge is not in the MST. The processing of edge e illustrates the case where both vertices belong to different partial trees, 6 and 4 in this case, and the edge level is greater than the highest node level of both trees. Thus, a new node 9 at level 50 is built, these trees are linked, and 9 becomes its father. This current step is depicted in fig. 14. It can be observed that this new node 9 is not representing any vertex, *i.e* is not appearing in the vertex component mapping but it would appear on an edge component mapping.

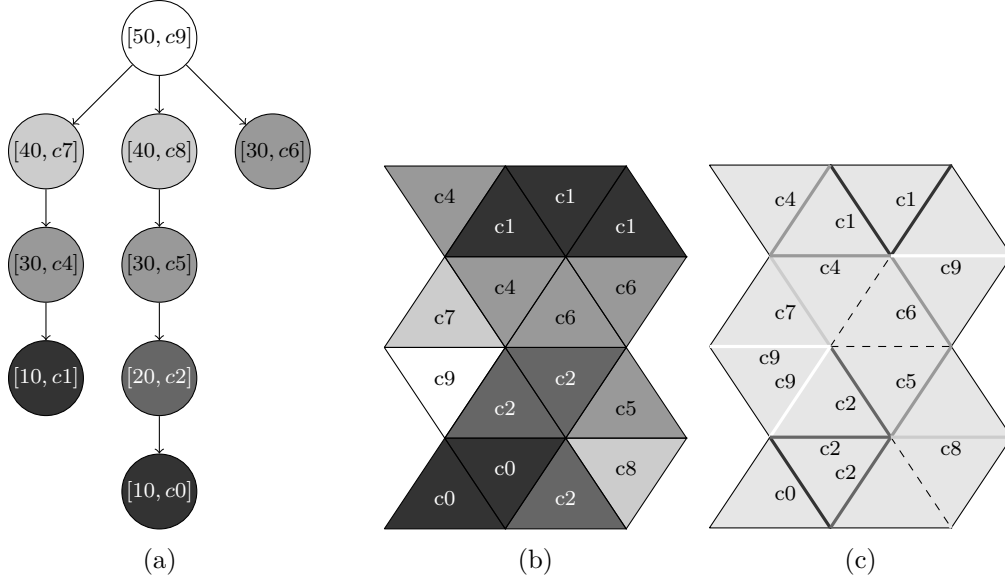


Figure 15: End of the algorithm. (a) The component tree. (b) The vertex component mapping on faces of the set of fig. 11(a). Colors on faces represent its level. (c) The edge component mapping on edges of the set of fig. 11(a), (dashed edges do not belong to the MST of the vertex per-face graph). Colors on edges represent its level.

For the rest of edges at level 50, edge g is not treated as is not part of the MST as edge r . Edge i will make the last not assigned vertex belong to node 9, while edge k will link the remaining two partial trees. Edge j is the last edge and is not considered as g . The component tree is then built. It is applied the operation $\text{Find}_{\text{node}}$ to retrieve the definitive component mapping, actually to get the correct labels that may change due to nodes merging. The component tree is depicted in 15(a) and the component mapping on the faces of fig. 11(a) in fig. 15(b). The edge component mapping is depicted in fig. 15(c). Dashed edges on the edge component mapping are the corresponding edges of g, r, j from the vertex per-face graph, which do not belong to the MST, thus they are not considered in the mapping. Remark also to consider the edge corresponding to e , that belongs to the component 9 of the tree, and that is not reflected in the face component mapping.

4.6 Node Attributes

We will use the component tree to reduce the number of minima of the initial curvature map, in order to reduce the over-segmentation of the watershed operator. As the minima of the function are represented by leaves of the tree, the idea will be to remove leaves in an order of importance until we obtain a desired filtered function. To quantify this order of importance we will compute some attributes on each node of the tree, to prune the tree by different criteria.

Several attributes can be computed, and we have computed the dynamics, the area and

the volume (fig. 16).

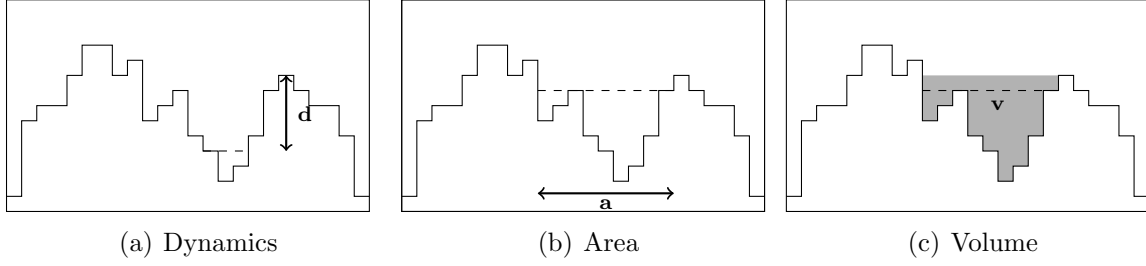


Figure 16: Illustration of the dynamics, area and volume of the component at the level depicted in a dashed line.

Let $[k, c] \in \mathcal{C}(F)$ and $[k_p, c_p] \in \mathcal{C}(F)$ the parent of $[k, c]$. We define

$$dynamic([k, c]) = \begin{cases} \infty & [k, c] = [k_{max}, V] \text{ or } dynamic(k_p, c_p) = \infty \\ k_p - k + dynamic(k_p, c_p) & \text{if } \min\{F(x)|x \in c\} = \min\{F(y)|y \in c_p\} \\ k_p - k & \text{in other case} \end{cases}$$

Intuitively, if we consider the height function as a topographical relief and we flood it by water starting at the regional minima points, the dynamic of a point will be the height between this point and the point where the water would overflow to another minima. Thus, the water flooding from a minima will not overflow to regions of lower depth, as water from these regions will overflow first.

We also define

$$area([k, c]) = card(c)$$

$$volume([k, c]) = \sum_{x \in c} (F(k_p) - F(x))$$

The area is computed while building the component tree. Each time a vertex is added to a node (lines 5, 9, 11 of the algorithm) the area is increased by the number of vertex added (*i.e* 2 in the case of line 5, and 1 in the other two cases). As in our case each vertex represent a triangular face, the area of the face may be added.

The volume and the dynamics are computed in a post-processing step. To compute the volume the area is needed. Then we apply the recursive function `ComputeVolume` on the root of the tree.

To compute the dynamics we need to sort the nodes of the tree by increasing order of their level. Then, by using the recursive function `CompTreeOrder` applied on the root we obtain the son of minimum level for each node of the tree (array *minSon*). Then the function `ComputeDynamics` also applied on the root of the tree, will compute the dynamics of each node.

Function double ComputeVolume(*int n*)

```
1 vol := nodes[n] →area;
2 fth := nodes[n] →father;
3 foreach c child of nodes[n] do
4   if (nodes[n] is the root) then
5     | vol := vol + ComputeVolume (c) + c→area*(n→level - c→level);
6   else
7     | vol := vol + ComputeVolume (c) + n→area*(fth→level - n→level);
8 nodes[n] →volume := vol;
9 return vol;
```

Function int CompTreeOrder(*int n*)

```
// N Number of nodes of the component tree
Data: nodeOrder - array [0...N - 1] node order by increasing level
Result: minSon - array [0...N - 1] son of minimum level for each node
1 m := MAX ;
2 if (nodes[n] is a leaf) then
3   | minSon[n] := n;
4   | return nodeOrder[n];
5 else
6   foreach c child of nodes[n] do
7     | r := CompTreeOrder(c);
8     | if (r < m) then x := minSon[c]; m = r;
9   minSon[n] := x;
10 return m;
```

4.7 Filtering

Once the component tree is built and the node attributes computed, our purpose is to use it to filter the initial function by reducing its number of minima. This can be done either by removing nodes until an attribute level threshold is reached, or by removing nodes until a desired number of minima is reached , *i.e* nodes are removed until a number of leafs remains in the tree.

The attributes area and volume increase from child to parent. In the algorithm **Keep_N_Minima** we perform the task of removing nodes by increasing order of area or volume until a number N of minima is reached. For the case of the dynamics, the attribute value is not increasing from child to parent, and in this case the algorithm **Keep_N_Minima** is lightly modified to remove entire branches by increasing order of leaf dynamics values. Also changing the conditions, the filtering process is achieved for a desired attribute threshold instead of the parameter number of minima. Remark that the filtering can be achieved for

Function `ComputeDynamics(int n)`

Data: *minSon* - array $[0 \dots N - 1]$ son of minimum level for each node

```

1 fth := nodes[n] →father;
2 if nodes[n] is the root then nodes[n] →dynamics := MAX;
3 else
4   if (minSon[n] == minSon[fth]) then
5     if (nodes[fth] →dynamics == MAX) then nodes[n] →dynamics := MAX;
6     else
7       nodes[n] →dynamics := nodes[fth] →level - nodes[n] →level +
       nodes[fth] →dynamics;
8   else
9     nodes[n] →dynamics := nodes[fth] →level - nodes[n] →level;
10 foreach c child of nodes[n] do ComputeDynamics(c);

```

the vertex values or by edges values.

The filtering process can be applied successively with different attributes and thresholds to obtain the desired results.

5 Region attributes

In order to obtain feature vectors which will form the object signature for a content-based search system, we compute attributes for each region resulting after the segmentation step. The attributes computed are three, all of them in the form of a histogram for each region. The cord histogram, and the Extended Gaussian Images (EGI) are based on global feature descriptors, in our case applied independently for each region, which had been used by Gorisse [7] in a previous work in the Eros3D project. Also we compute a histogram of the curvatures for each region.

5.1 Cords histogram

The cords histogram is based on the global method presented in [16], and represent a normalized histogram of the distances between the barycenter of the object and the barycenter of each triangle of the mesh. Also, the histograms of the angles with the first and second principal axis of the principal components analysis (PCA) are computed. Thus, a first step will be to calculate the principal axes.

The principal components analysis (PCA) is a statistic method consistent on finding the directions in a space which best explain the dispersion of random variables. In our case the random variables are the points of the 3D objects. The principle of the PCA is to calculate the eigen vectors and eigen values of the covariance matrix of the coordinates of

Algorithm 11: Keep_N_Minima

Data: (V, E, F) - edge-weighted graph with N vertices and CT the component tree with attribute value for each node, and the edge component mapping CME

Data: N the number of desired minima

Result: F the filtered map

```
1 Sort the nodes of  $CT$  by increasing order of the attribute value;
2  $Q := \emptyset$ ;  $L :=$  number of leaves in  $CT$ ;
3 forall  $n$  do  $nodes[n] \rightarrow mark := 0$ ;
4 while  $L > N$  do
5   Choose a (leaf) node  $c$  in  $CT$  with the smallest attribute value;
6    $fth := nodes[c] \rightarrow father$ ;
7    $nodes[c] \rightarrow NbChildren := nodes[fth] \rightarrow NbChildren - 1$ ;
8   if  $nodes[fth] \rightarrow NbChildren > 0$  then  $L := L - 1$ ;
9    $nodes[c] \rightarrow mark := 1$ ;  $Q := Q \cup \{c\}$ ;
10 while  $\exists c \in Q$  do
11    $Q := Q \setminus \{c\}$ ;  $RemoveMinima(c)$ ;
12 foreach  $u \in E$  do  $F(u) := nodes[CME[u]] \rightarrow level$ ;
```

Function `int RemoveMinima(int n)`

```
1 if  $(nodes[n] \rightarrow mark == 1)$  then
2    $nodes[n] := nodes[RemoveMinima(nodes[n] \rightarrow father)]$ ;
3 return  $n$ ;
```

each triangle barycenter. The barycenter of the object is the center of the 3D space while the three reference axis correspond to the directions of the eigen vectors of the covariance matrix. The PCA is a method widely used to obtain a normalized pose of 3D objects.

Once the PCA is computed, the distances (c_i) between the barycenter of each triangle i and the barycenter of the object are calculated. Also the angles (α_{2i}, α_{2i}) with the two principal axis. Then we proceed to calculate the histograms. A determined number of classes is established for the distances c_i of each triangle, and its angles α_{1i}, α_{2i} , and then independent histograms are computed for each region, and normalised. A histogram that relates distances and angles is calculated, and we call it *cords2D*. The histogram *cords2D* has one class for each pair distance-angle (for instance, only one of the angles is considered $t\{c_i, \alpha_{1i}\}$), so gives precise values of the position of the region by reference on the principal axis.

5.2 Extended Gaussian Images

The principle of the Extended Gaussian Images (EGI) [8] is to project a function that synthetize the information concerning the mesh of the object into a Gauss sphere divided

in faces. Each triangle adds a contribution of its area to the face of the sphere which has the same orientation. Thus, we have for each face P of the Gauss sphere with orientation \hat{n}_k ,

$$P_{\hat{n}_k} = \sum_{i=1}^{N_k} A_{i,\hat{n}_k}$$

where N_k is the number of triangles of the mesh oriented following the direction \hat{n}_k . The orientation of a triangle of the mesh is defined by its normal vector.

The discretized Gauss sphere is build to obtain size homogenic faces. To do so, we use the faces of a regular octahedron, that for instance are subdivided two times, so we obtain $8 * 4 * 4 = 128$ faces. Then we project these faces on a sphere.

Once the sphere is built the algorithm proceed as follows:

- calculate the reference directions resulting from the sphere
- calculate the directions of each mesh triangle of the object
- for each mesh triangle direction, find the nearest reference direction.

Then we calculate the histograms for each region of the mesh. Each histogram has as classes as faces of the sphere. Then for each triangle of a region we add its area to the class that previously has been computed as its nearest reference direction.

This descriptor applied on regions combines the representation of the position of the region in the object with the shape of this region.

5.3 Curvature histogram

This attribute is an histogram of the principal curvatures computed in each triangle of the mesh. Also in this case we applied the histogram type described for the cords (cords2D sec. 5.1) which considers a class for each pair $\{\kappa_1, \kappa_2\}$, where κ_1, κ_2 are the principal curvatures at each triangle of the mesh.

6 Experimental results

In this section we will present our experimental results on 3D models. The procedure we applied is as follows:

- Compute a curvature map on the mesh.
- Build the component tree of the curvature map.
- Filter the original curvature map by using the component tree.
- Compute the watershed on the filtered curvature map.

- Compute the region attributes.

We illustrate some results from the filtering step in fig. 17. Once the component tree is built, different approaches can be used to reduce the number of minima of the curvature map. We applied the function described in sec. 4.7 `Keep_N_Minima` in order to obtain exactly 30 minima on the resulting curvature map.

In fig. 17(a) it can be observed the original curvature map. In fig.17(b) we depict the resulting curvature map after reduce the minima by considering the attribute Area. In fig.17(c) the attribute considered is Volume, while in fig. 17(d) the map is filtered considering Dynamics. In fig.17(e) the filtering is achieved by applying successively the function `Keep_N_Minima` considering several different attributes. More precisely we applied first Area filtering (200 minima), then we applied Volume filtering on the resulting map (80 minima), and finally the Dynamics filtering (30 minima). It can be noticed that either by Area filtering as Volume filtering the minima corresponding to both feet disappear. The differences between the dynamics filtering and the combined filter (fig. 17(d), fig. 17(e)) are more difficult to appreciate, but the dynamics filtering is not eliminating small minima, as the small minima between both legs, while the combined filter it does.

It can be noticed small black dots in filtered images. It is due at the visualization software. For a proper visualization colors should be given at faces of the mesh. The software we used allowed to give colors on vertices of the mesh, so we proceed giving each vertex a value proportional to the mean of its surrounding edges. Thus, there are some vertices that are surrounded by edges that belong to the MST of the vertex per-face graph. We give black color at this vertices as we do not have a filtered value for this vertex. .

After the filtering process the watershed is computed. The parameters for the filtering step can vary depending on the desired definitive regions. In fig. 18 we show the evolution of the number of regions by filtering the initial curvature map. Fig. 18(a) shows the watershed segmentation of the original map. Figures 18(b) and 18(c) show watershed cuts of two different filtered maps.

For certain objects we observed that the curvature scalar function M (max of principal curvatures) performs better than the H_{inv} (sec. 2.3). Fig. 19 illustrates the watershed cut applied on the H_{inv} map (fig.19(a)) and on the M map (fig.19(a)). It can be observed that in the case of H_{inv} the minima on convex zones produce undesired watershed lines. By using M the watershed lines are either placed on convex or concave edges.

Fig. 20 show the results of the segmentation step on different sculptures. We applied the same parameters for them three, to obtain a maximum of 100 regions. As the filtering parameters that can be optimal for one object, cause a lost of regions with special meaning in other objects, the filtering parameters are set up despite they lead to an over-segmentation in some objects.

From the watershed cut we could obtain also features related to the divide lines. These lines may not have "meaning", in the sense that they may not follow features of the object. We give the weight of the curvature values of the original map to measure the importance of these lines. Fig. 21(a) shows the watershed lines in grayscale, which correspond to the curvature values of the original map. In Fig. 21(b) and 21(c) the lines in black represent

the parts of the watershed lines that are over a certain threshold of curvature value. It can be observed that in these two similar sculptures the lines extract the same features.

Once the watershed is computed we calculate the attributes for each region obtained (sec. 5.3). Figures 22, 23 and 24 show the results for three different objects. Regions are labeled with letters. For each region we show a type of histogram to show the different features that can be considered to identify a region.

For the histograms *cords2D* we set 50 classes for the cord length value, and 50 classes for the first angle. Thus the resulting histogram has $50 \times 50 = 2500$ classes, set in one dimension, so for each angle class we depict the 50 cord length classes. The histograms *curvatures2D* have the same number of classes, 50 classes for the principal curvature κ_1 and 50 classes for κ_2 . Also they are depicted in one dimension, so for each class of κ_2 the 50 classes of κ_1 . For the *EGI* histograms we set 128 classes, which correspond to 128 faces of a Gauss sphere.

7 Conclusion

We presented a method which allows to define similarities and differences between 3D objects that have the same global shape. This is for example the case of scanned sculptures. In sculptures usually we have characteristic differences between them that rely on the surface either than on its shape. In this case, methods which extract global shape descriptors are not able to describe properly the object while our method obtain regions that follow carved features on the surface.

We can obtain feature vectors by using the histograms computed for each region. These vectors are an essential part for a content-based retrieval system as they help the indexation of the database. As we obtain independent feature vectors for each region, the method can be used for partial matching. In other words, we can search objects which have one or some parts similar even if they are different in global shape.

The component tree is a structure that gives robustness to the segmentation method, as it allows noise filtering and different levels of precision for the feature extraction. The three tree node attributes presented (volume, dynamics and area) improve the filtering by applying them successively. The combination component tree and watershed can be used in other frameworks, as image, video, signal. Also the same method can be applied on other maps on the surface mesh, as color, texture, or other geometrical functions instead of curvature.

We defined some notions on simplicial complexes, which is a structure that fit perfectly with the representation of 3D meshes. We gave the links to apply the watershed transform in the framework of complexes. This can lead to optimality properties for the method presented in a further work.

Further tests on retrieval systems may define better region attributes. Also by studying the results by different sets of parameters we can define an optimal automatic filtering. The remeshing step combined with the filtering step can lead to multi-resolution results.

References

- [1] M. Attene and B. Falcidieno. ReMESH: An Interactive Environment to Edit and Repair Triangle Meshes. *Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)-Volume 00*, 2006.
- [2] G. Bertrand. On critical kernels. Technical report, IGM 2005-05, Institut Gaspard Monge, 2005.
- [3] G. Bertrand and M. Couprie. Two-dimensional parallel thinning algorithms based on critical kernels. Technical report, IGM2006-2, Institut Gaspard-Monge, Universite de Marne-la-Vallee, 2006.
- [4] M. Couprie. Pink. ESIEE. <http://www.esiee.fr/~coupriem>.
- [5] J. Cousty, G. Bertrand, L. Najman, and M. Couprie. Watersheds, minimum spanning forests, and the drop of water principle. Technical report, IGM 2007-01, Institut Gaspard Monge, 2007.
- [6] P. Frey. YAMS A fully Automatic Adaptive Isotropic Surface Remeshing Procedure. *Writing*, page 11, 2001.
- [7] D. Gorisse, M. Cord, and M. Jordan. Indexation 3d dans une base d'oeuvres d'art. Technical report, ETIS, Ecole National Supérieur d'Electronique et ses Applications, 2006.
- [8] B. K. P. Horn. Extended Gaussian Image. *Proceedings of the IEEE*, 72(12):1671–1686, 1984.
- [9] J. Kruskal Jr. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.
- [10] J. O. Lachaud. Topologically defined iso-surfaces. *DGCI'96, LNCS 1176, Springer Verlag*, pages 245–256, 1996.
- [11] A. P. Mangan and R. T. Whitaker. Partitioning 3d surface meshes using watershed segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):308–321, 1999.
- [12] A. Meijster and M. Wilkinson. A comparison of algorithms for connected set openings and closings. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4):484–494, 2002.
- [13] P. Min. binvox. <http://www.cs.princeton.edu/~min/binvox>, 2003.
- [14] L. Najman and M. Couprie. Quasi-linear algorithm for the component tree, 2004. IS&T/SPIE Symposium on Electronic Imaging 2004, Vision Geometry XII.

- [15] F. S. Nooruddin and G. Turk. Simplification and Repair of Polygonal Models Using Volumetric Techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):191–205, 2003.
- [16] E. Paquet and M. Rioux. Nefertiti: a Query by Content Software for Three-Dimensional Models Database Management. *Proceedings of the International Conference on Recent Advances. 3-D Digital Imaging and Modeling*, pages 345–352, 1997.
- [17] A. Razdan and M. Bae. A hybrid approach to feature segmentation of triangle meshes. *Computer-Aided Design*, 35(9):783–789, 2003.
- [18] S. Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes, 2004. 3D Data Processing, Visualization, and Transmission, 2nd International Symposium on (3DPVT’04), 486–493.
- [19] P. Salembier, A. Oliveras, and L. Garrido. Anti-extensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*, 7(4):555–570, 1998.
- [20] R. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.
- [21] C. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computers*, 20(1):68–86, 1971.

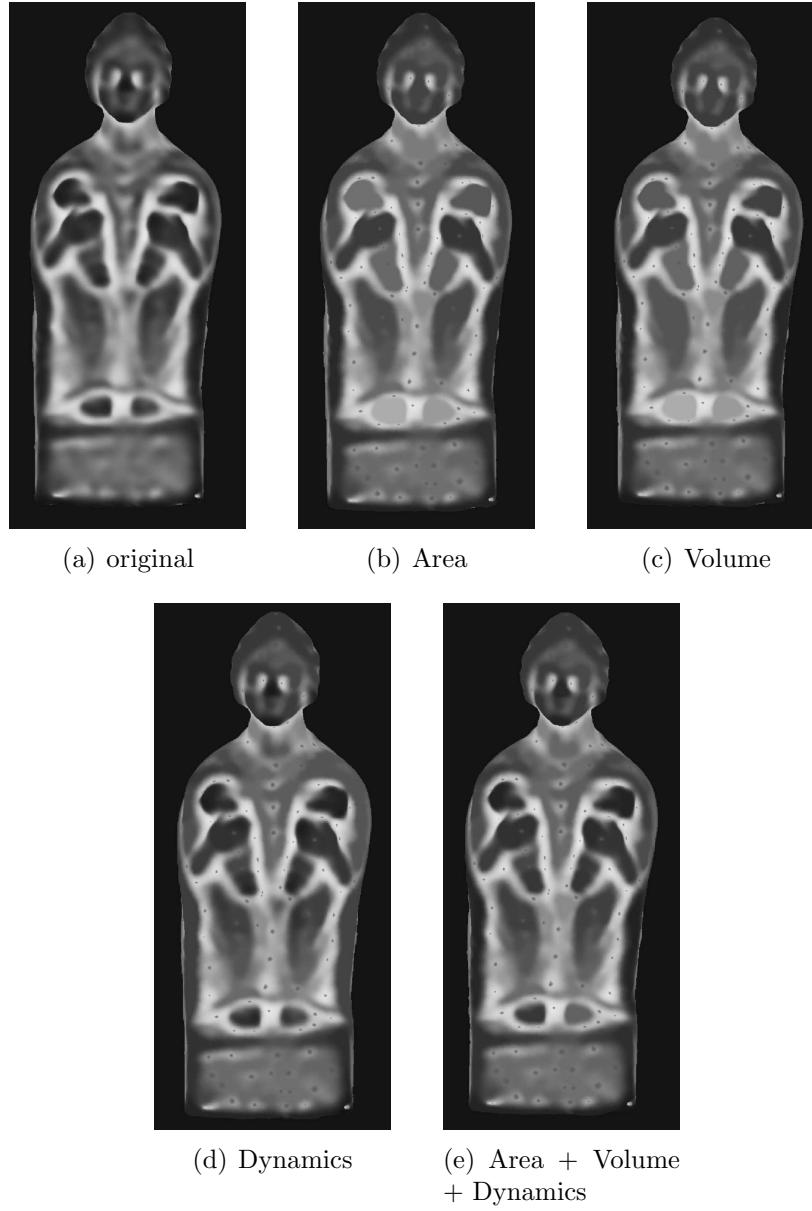


Figure 17: Filtering. Curvature maps resulting after applying the filtering on the original curvature map from fig. (a). Minimum values are in black while maximum values are in white. The number of minima of all the resulting maps is 30.

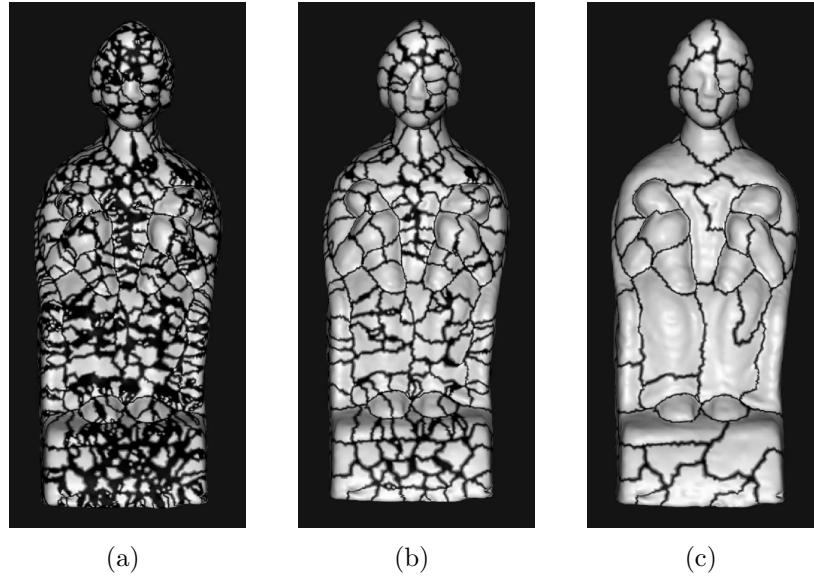


Figure 18: Watershed. (a) Watershed cut of the original curvature map (2622 regions) and (b) watersheds of two filtered curvature maps (b), (c) (800 and 200 regions respectively).

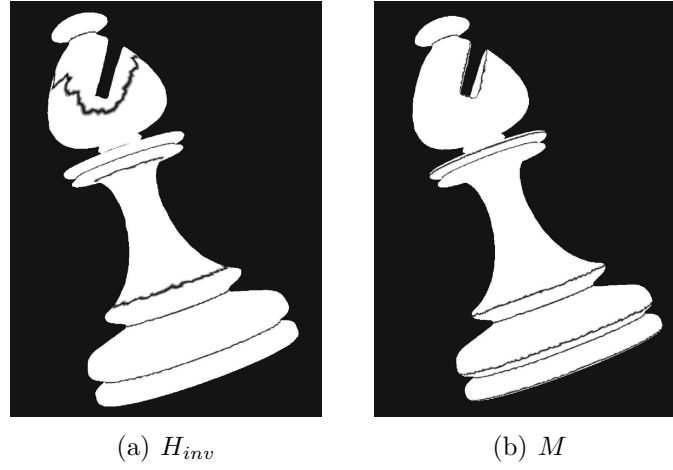


Figure 19: (a) Watershed cut on H_{inv} curvature map.(a) Watershed cut on M curvature map.

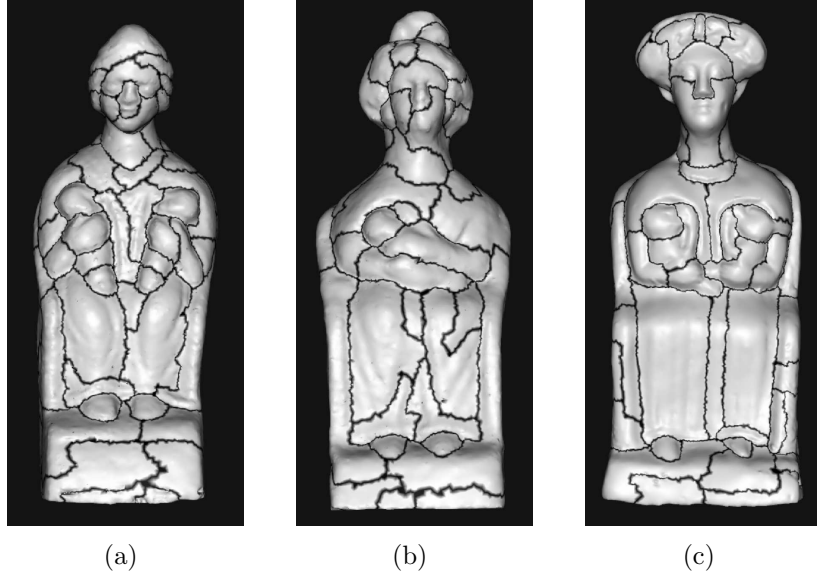


Figure 20: Segmentation on different sculptures.

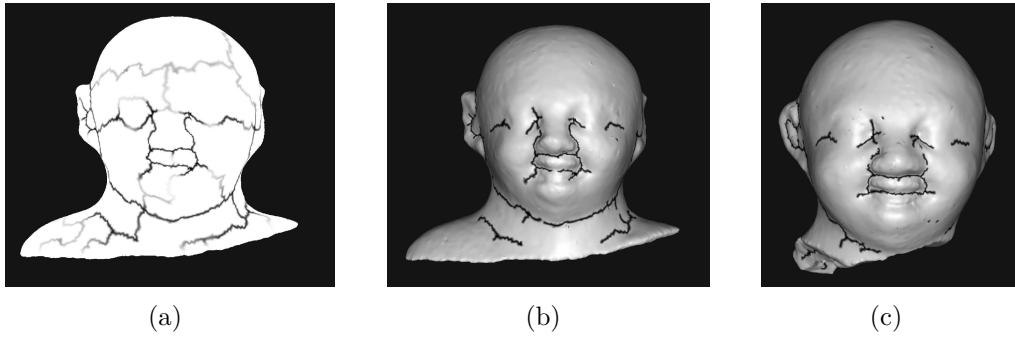
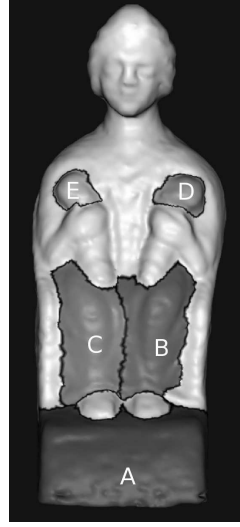
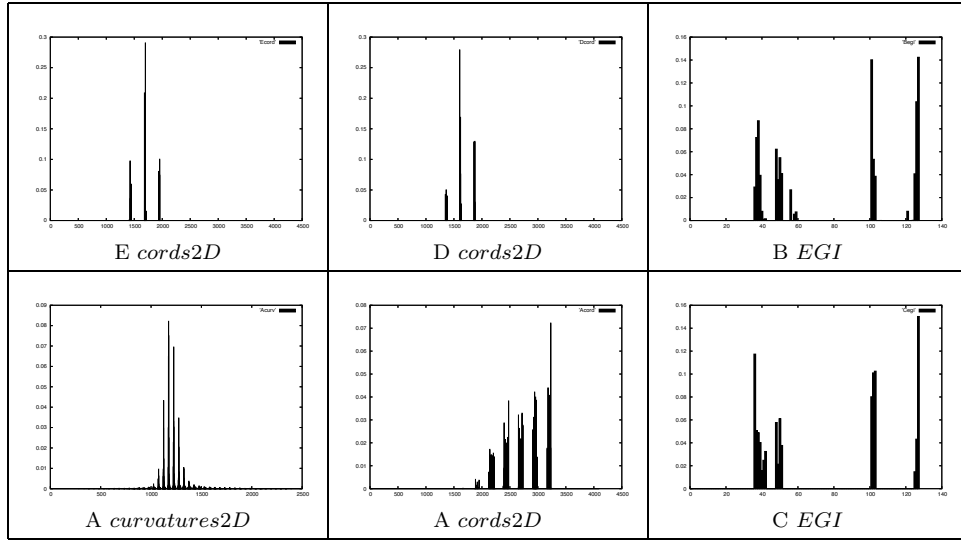


Figure 21: (a) Watershed cut in grayscale values of the original curvature map. (b) Thresholded cut of the watershed in (a) and (c) thresholded cut of a similar sculpture.

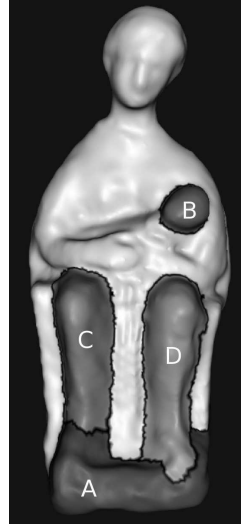


(a)

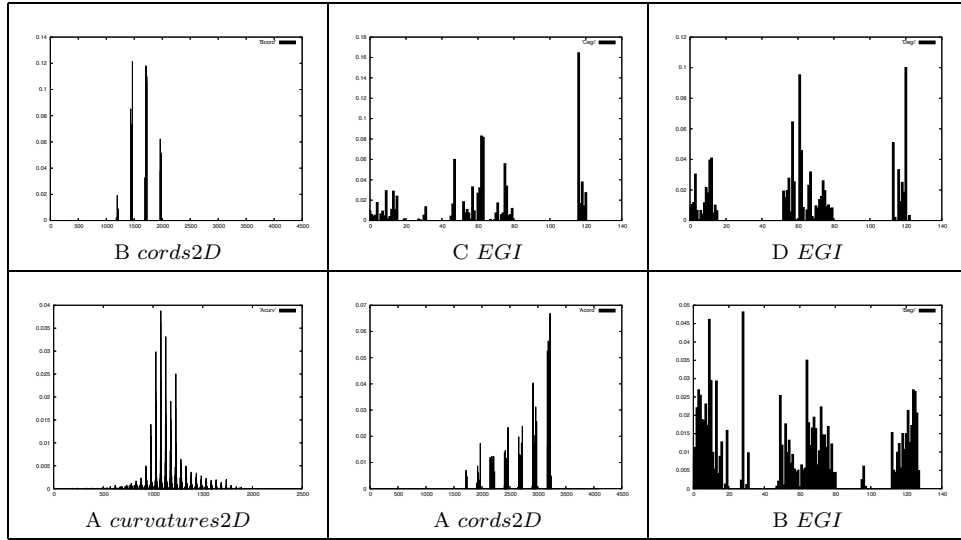


(b)

Figure 22: Region attributes. Histograms for the region depicted by a letter in (a). *cords2D* $50 \times 50 = 2500$ classes. *curvatures2D* $50 \times 50 = 2500$ classes. *EGI* 128 classes.

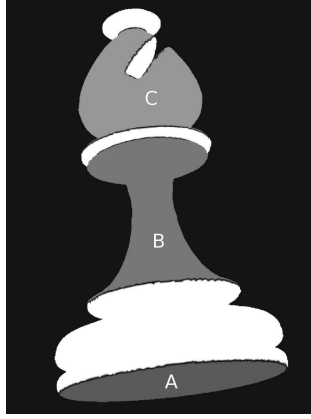


(a)

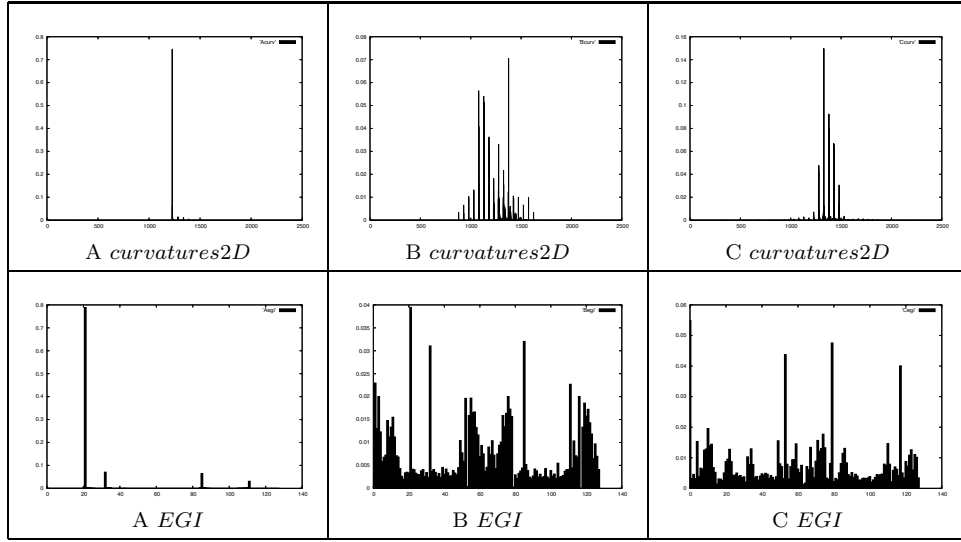


(b)

Figure 23: Region attributes. Histograms for the region depicted by a letter in (a). *cords2D* $50 \times 50 = 2500$ classes. *curvatures2D* $50 \times 50 = 2500$ classes. *EGI* 128 classes.



(a)



(b)

Figure 24: Region attributes. Histograms for the region depicted by a letter in (a). *cords2D* $50 \times 50 = 2500$ classes. *curvatures2D* $50 \times 50 = 2500$ classes. *EGI* 128 classes.