Storage and manipulation of vague spatial objects using existing GIS functionality

Arta Dilo[1], Pieter Bos[2], Pawalai Kraipeerapun[3], and Rolf A. de By[1]

[1] International Institute for Geo-Information Science and Earth Observation (ITC), PO Box 6, 7500 AA Enschede, The Netherlands. Email: [dilo,deby]@itc.nl,
[2] Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE, Enschede, The Netherlands. Email: p.bos@student.utwente.nl,
[3] School of Information Technology, Murdoch University, South Street, Murdoch Western Australia 6150, Australia. Email: P.Kraipeerapun@murdoch.edu.au

1 Introduction

We collect and store data to derive information and make judgments about a world of our interest. Ideally, they should indicate in a unique and certain way which possible world corresponds to the actual world [17]. Imperfection arises when this is not possible. Imprecision is a type of imperfection that is often encountered. Data are imprecise if we cannot precisely define the actual world, i.e. several worlds satisfy the data. A specific type of imprecision is vagueness [17, 22], which is the focus of this study. A concept is vague if objects exist that cannot be classified either to the concept or to its complement. Vagueness arises in the presence of borderline cases [18]. It is often present in collected spatial information, such as forest inventories, or geological, soil, and vegetation maps. Soil or vegetation classes are such that they cannot be defined sharply. The change from one class to another is gradual. This is in conflict with current geographical information systems (GIS) which assume that spatial objects are precisely defined, sharp objects, using points, lines, and polygons as representations.

Several theoretical models have been proposed to represent and handle vague objects. They can be divided into two groups. One group [2, 3, 4, 12] deals only with regions, called broad boundary regions. The boundary of such a region is not a sharp line but a zone of transition, which is considered to be homogenous. The other group [13, 14, 23] considers gradual changes in the transition zone, and models objects by employing fuzzy sets. Schneider [14] defines fuzzy points, fuzzy lines and fuzzy regions, based on a finite collection of elements of a regular grid, which form a partition of $\mathbb{R}^2$. The model is directly implementable in raster data format. The other theoretical models

have not been followed up by implementations, and (to our knowledge) there is no other implementation of vague objects.

The work presented in this paper follows our previous work [7], which provides formal definitions of vague object types and their operators. This work is an implementation of these vague types and operators in GIS software. Its objective is to store and manipulate vague spatial information, by extending existing GIS functionality. GRASS, an open source GIS software, was selected for the implementation, to allow the use of existing spatial data handling capabilities. The rest of the paper is organized as follows: Section 2 informally describes vague objects and operators we deal with, giving the intuition of the definitions provided in [7]. Section 3 is dedicated to the creation of vague objects from input data points, and to their storage. Section 4 describes the visualization techniques used to display vague objects. Section 5 provides for set operators on vague objects. Sections 6 and 7 close the paper with discussions and conclusions.

2 Vague spatial types and operators

Vagueness in spatial information could be positional, meaning the location of a certain object is vaguely described, or it could be thematic, meaning properties of an object are vaguely described, e.g. in natural language terms, which are generally vague. The vague types that we have provided in [7] deal with thematic vagueness. A spatial object described by one of these types has a known location, but its properties can only be expressed in vague terms. For example:

- 'Densely populated' residential centres are represented by points with precise location, which have different degrees of population density. Their property of being 'densely populated' is a vague term.
- A traffic congestion is described by a property on a road network (which location is precise for our purpose): the 'congestion' level, which can only be expressed in vague terms. Part of the road is completely blocked, hence certainly belongs to the traffic congestion, whereas away from the cause of the congestion, the car build-up becomes less severe, therefore it is part of the traffic congestion only to a certain degree.
- Agricultural land 'suitability' is a property of land (space) described by a vague term. Suitability of land for a given kind of agriculture is decided on the basis of a combination of precise criteria built on natural linguistic terms [16]. There exist locations that are certainly suitable for agriculture, whereas other locations are suitable only to a certain degree.

The types we provide are either simple or general ones. A simple type is used to represent a basic object, i.e. the simplest identifiable object. A general type is used to represent a set of basic objects that belong to the same vague class (of a certain classification). Partitioning of space based on a given classification is

important for many spatial applications. A classification consisting of vague classes does not lead to a crisp partition of space. We introduce a type vague partition that allows some kind of overlapping between classes, still giving a meaningful classification of space.

A vague object of simple or general type is a fuzzy set in the real plane, satisfying specific properties. It is represented by its membership function $\mu : \mathbb{R}^2 \rightarrow [0, 1]$. Simple types are *simple vague point*, *simple vague line*, and *simple vague region*. Figure 1 illustrates objects of these types.

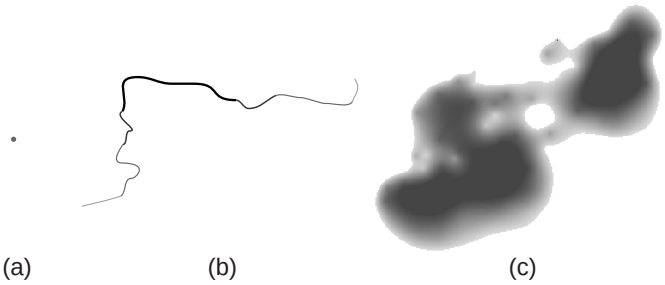

Fig. 1. (**a**) simple vague point, (**b**) simple vague line, and (**c**) simple vague region. Dark tone indicates high membership value, light tone indicates low membership value.

A *simple vague point* represents a site with a known location but with uncertain existence (to a phenomenon of interest). It has a positive membership value only at that location; the membership value represents the certainty of existence of the site. A *simple vague line* represents a linear feature with known position but with an uncertain extent, i.e. any point of the line has some degree of participating in the line. A *simple vague line* is a continuous line with mostly gradual transitions of membership values between neighbour points on the line. Membership values are positive at every point of the line, except perhaps at the end points. A *simple vague region* is a region with a broad boundary. Locations in the broad boundary typically have different positive membership values, which change mostly gradually between neighbour points in the region. It does not have cuts, punctures, isolated lines or isolated points. Its support set (i.e. the set of locations with positive membership value) is a single-component set, possibly with holes. The core (i.e. the set of locations with membership value equal to 1) however, can be composed of several components, possibly containing holes.

The general types are *vague point*, *vague line*, and *vague region*. A *vague point* is a finite set of disjoint *simple vague point*s. A *vague line* is a finite set of *simple vague line*s that intersect only at their end nodes, and have the same membership value at any common end node. A *vague region* is a finite

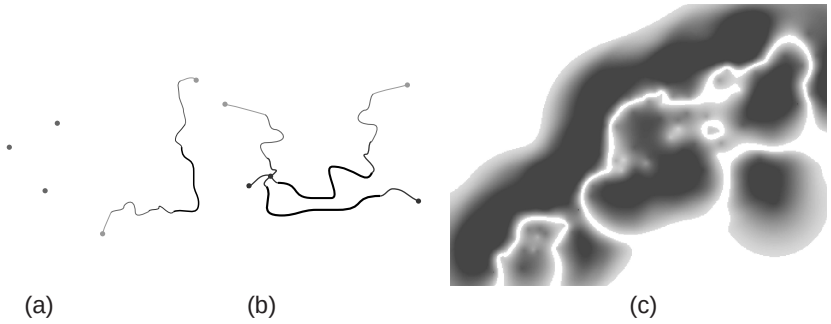

Fig. 2. (**a**) vague point, (**b**) vague line, and (**c**) vague region.

set of disjoint *simple vague region*s. All these finite sets are potentially empty. Figure 2 illustrates these objects.

A *vague partition* is a set of classes, where a class is of type *vague region*. Objects belonging to different classes could overlap only at their uncertain parts. The certain part of one object is disjoint from all objects in any other class.

The basic operators we provide are union, intersection, and difference. These are binary operators defined on general types. General types are closed under these vague spatial operators, meaning the result of an operator is of one of these types. Every operator takes as arguments (two) objects of the same type, and returns an object of that type, except for the intersection of vague lines, which returns a vague point. The union between two vague regions can be used to generalize a classification: two classes can be joined to create a new class that is more general than the previous two. The intersection of vague regions can be used to combine two classifications into a new one: two classes from different classifications can be combined to form a more refined class.

Union, intersection, and difference between vague objects are defined using fuzzy set operators – fuzzy union, fuzzy intersection, and fuzzy difference, respectively. Union of two objects μ_1, μ_2 gives a new object μ of the same type, of which the membership value at every location is taken as the maximum of membership values of the input objects: $\mu(P) = \max\{\mu_1(P), \mu_2(P)\}$. Intersection between two vague lines gives a vague point, whereas intersection between vague points and between vague regions gives, respectively, a vague point and a vague region. The intersection of two vague objects μ_1 and μ_2 results in a vague object μ of which the membership value at every location is the minimum of membership values of input objects at that location: $\mu(P) = \min\{\mu_1(P), \mu_2(P)\}$. The difference μ between two vague objects μ_1 and μ_2 is the intersection of the first object with the complement of the second. The membership value at each location is taken as $\mu(P) = \min\{\mu_1(P), 1 - \mu_2(P)\}$.

3 Storage of vague objects

In this section we discuss our storage model of vague objects. An object of a general vague type is a collection of simple vague type objects. Simple type objects are the basic objects that we need to store. An identifier is given to such an object and used to compile the complete information at any time the object is needed. In this section we show how to store simple vague points, simple vague lines, and simple vague regions. The data we can collect and store can only be finite, while a simple vague line and a simple vague region represent infinite point sets. An interpolation method is used in both cases to complete (approximate) information about objects.

The GRASS vector format[4] is used to store vague objects. Feature types supported by GRASS are *point*, *line*, *boundary*, *area* (without holes), and *centroid*. All data is stored using line representations. A line is a sequence of (x, y) coordinates forming types *point*, *line*, *boundary*, or *centroid*. A *point* or a *centroid* is constructed as a sequence of two identical elements. A *line* or a *boundary* is constructed as a sequence with at least two elements. A *line* type is used to store linear features, whereas a *boundary* type is used to store areal features. An *area* is formed by a set of lines of type *boundary*, which constitute its boundary. GRASS allows the storage of three dimensional (3D) features, i.e. a line can be a series of (x, y, z) coordinates. It can only build the complete topology for 2D features; it builds connectivity of 3D linear features, but ignores the third coordinate when creating (and building the topology of) areal features. We use that third coordinate to store membership values.

Objects within a theme, e.g. road lines, vegetation classes, form what we call a data layer. A data layer is physically stored as a directory that contains several files, `coor`, `topo`, `sidx`, etc. each containing specific information. For example, vegetation data is stored in a directory `vegetation`. Location information is stored in the `coor` file in that directory, while topology information is stored in the `topo` file. One or more attributes can be attached to objects of a data layer.

Our *simple vague point* is implemented in GRASS by an SVpoint that is a triple (x, y, mv), where $(x, y) \in \mathbb{R}^2$ provides the location and $mv \in (0, 1]$ provides the membership value. A *simple vague line* is implemented by an SVline that is a sequence of triples $\langle (x_1, y_1, mv_1), \ldots, (x_n, y_n, mv_n) \rangle$, each triple providing the x, y location of a point in the line, associated by the membership value of that point to the line. An approximation of the *simple vague line* is achieved by linear interpolation between consecutive points. A *simple vague region* is a surface embedded in $\mathbb{R}^2$. Triangulations can be used to represent it, e.g. TIN structures in GIS software. A triangulation is the division of a surface or a polygon into a set of triangles, such that each triangle edge is shared by two adjacent triangles. A triangulation method consists of two parts: creating the triangulation, and performing an interpolation within each triangle. A

[4] We use GRASS version 6.0.

simple vague region is implemented by an SVregion that is a triangulation. An approximation of the *simple vague region* is achieved by a linear interpolation within each triangle of SVregion.

A *vague point* is implemented by a Vpoint, which is a set of triples (x, y, mv) having different locations. A *vague line* is implemented by Vline, which is a collection of SVline objects intersecting only at end points. A *vague region* is implemented by Vregion, which is a set of triangulations that do not overlap each other. Objects of a simple type belonging to the same class are stored in a data layer. We call it a vague data layer, for containing vague information. It is of one of the implementation types: Vpoint, Vline, or Vregion. For example, all SVregion objects belonging to the class 'forest' of a vegetation theme, are stored in a vague (data) layer of type Vregion.

A vague layer is created by input containing membership information, or such information is derived from attributes in input data by applying membership functions. We provide a module `v.vague.membership` that applies trapezoidal membership functions to a numerical attribute of a data layer. A vague point layer, i.e. a layer of type Vpoint, may derive from any point data containing membership information, or an attribute to which we could apply membership functions. Each input point associated with the membership value creates an SVpoint object in the vague point layer. A vague line layer is derived from measurements along linear features, e.g. level of congestion at locations along roads in a road network data layer. Such measurements can be direct membership values, or functions can be applied to them to derive membership values. An SVline object is created in the vague layer for each line object in the input data.

We assume that data about a vague region layer comes from points associated with membership values, or points having an attribute to which we apply a membership function. These points may be irregularly distributed, e.g. coming from measurements. They may also be regularly distributed, e.g. coming from processed satellite images. The only information we can get from such input is a membership value to a certain vague class, which may be indeed composed of several SVregion objects. The input needs to be interpreted to create the simple objects. We cluster input points, and consider that each cluster establishes a separate object. Points of each cluster are then used to create a triangulation for each simple object identified.

The next two sections, Section 3.1 and Section 3.2, are dedicated, respectively, to clustering the input to form separate SVregion objects, and creating objects from triangulation of clusters. Up to here, we have discussed about creation and storage of a vague layer, which is the implementation of a vague class. It is convenient to bind together all classes belonging to the same theme. This is discussed in Section 3.3.

3.1 Clustering input and delineating boundaries

Several techniques exist that can be used for clustering points in space: K-means (K-median) clustering, self organizing maps, hierarchical clustering, alpha shapes (see [1] for an overview). All techniques are based on some distance measure between points. The first two techniques require the number of clusters to be determined beforehand. This is usually not known for our case. Hierarchical clustering with Euclidean distance could be used for clustering the input. The technique is used for different inputs, not just points in Euclidean space, by employing different distance measures or different group (linkage) distances. It is a quite general, but slow technique. Alpha shapes (α-shapes from here onwards) work only with points in Euclidean space. They detect separate clusters from a given point set and delineate boundaries of the detected point clusters. The technique is faster than hierarchical clustering, and its output is richer as it contains a boundary for each cluster. We use α-shapes to cluster the input data and delineate the boundary of each cluster.

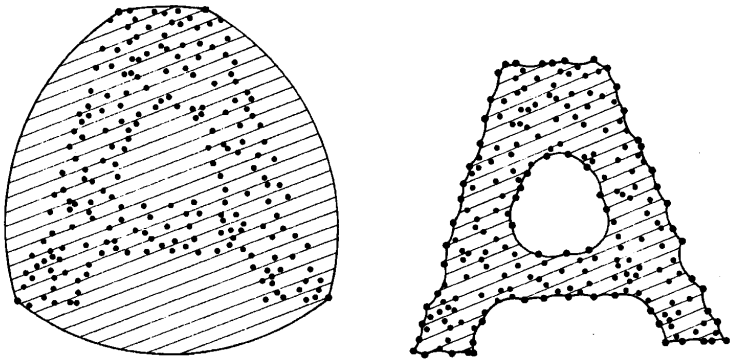

Fig. 3. Positive α-hull in the left, negative α-hull in the right (taken from [8]).

The α-shapes are a generalization of convex hulls. The *convex hull* of a point set S may be defined as the intersection of all closed half-planes that contain all points of S. This notion is generalized to *α-hulls* in [8]. For positive (yet sufficiently small) α, the α-hull of S is the intersection of all closed discs with radius $1/\alpha$ that contain all points of S. Large α produce a curved hull of which the boundary consists of parts of circles (with radius $1/\alpha$) that pass through extreme points of S. As α approaches zero, the curved hull converges to the convex hull. An α-hull is smoother than the convex hull, but its approximation of the intuitive shape of points is coarser than that of the convex hull. The shape of the hull can be refined by considering negative α's. For negative α, the α-hull of S is taken to be the intersection of all closed complements of discs with radius $-1/\alpha$ that contain all points of S. Figure 3

shows the positive α-hull (left) and the negative α-hull (right) of a point set resembling the letter 'A'.

The α-hull can be defined more concisely using the notion of a generalized disc. A generalized disc of radius $1/\alpha$ is a disc of radius $1/\alpha$ for $\alpha > 0$; it is the complement of a disc of radius $-1/\alpha$ for $\alpha < 0$, and a half-plane for $\alpha = 0$. For a point set S and a real value α, the α-hull of S is the intersection of all closed generalized discs of radius $1/\alpha$ that contain all points of S. Two other concepts, α-extreme and α-neighbour, are used in [8] to define the α-shape concept. A point p of a set S is termed an α-extreme in S if there exists a closed generalized disc of radius $1/\alpha$ such that p lies on its boundary, and it contains all points of S. Two α-extremes p and q are said to be α-neighbours if there exists a closed generalized disc of radius $1/\alpha$ with both points on its boundary, and which contains all points of S. An α-shape of S is then the straight line graph whose vertices are the α-extremes and whose edges connect the α-neighbours [8]. The assumption is that no four points of S are cocircular and no three points are collinear.

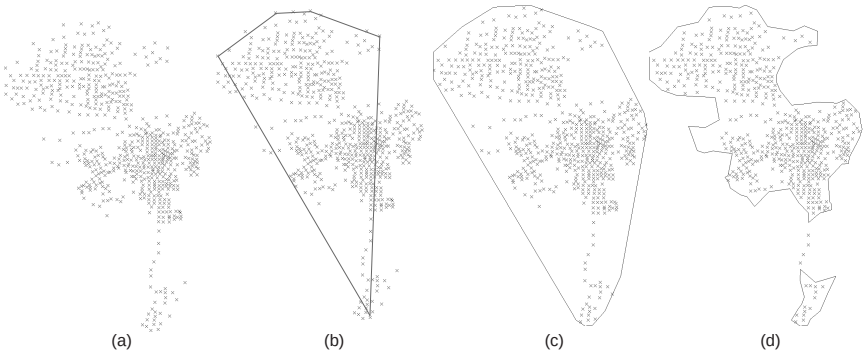

Fig. 4. (**a**) input points, (**b**) α-shape for a positive α, (**c**) α-shape for α equal to 0, (**d**) α-shape for a negative α.

Figure 4(b)–(d) illustrate α-shapes produced for decreasing α values on the same point set shown in Fig. 4(a). A decreasing α value results in a finer shape. Depending on the α value, a single point can be a cluster (and its own boundary at the same time), as is the case in Fig. 4(c) for four points. The set of α-extremes becomes larger when α decreases: the set of α_1-extreme points of a point set S is a subset of α_2-extreme points of S if $\alpha_1 > \alpha_2$ [8]. We are interested in shapes finer than the convex hull, therefore we work only with negative α values. This simplifies the checks for α-extremes and α-neighbours, and also the complete algorithm for building the α-shape.

A point is an α-extreme of a point set if and only if a circle with radius $-1/\alpha$ (α-circle hereafter) can be constructed such that the point is on the circle and no other point of the set lies inside it. Figure 5 illustrates α-extremes

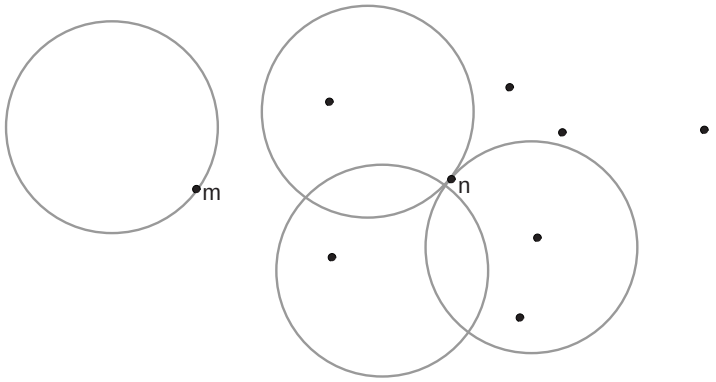

Fig. 5. Examples of α-extreme in a set A: point m is an α-extreme, point n is not.

in a point set A. Point m from A is an α-extreme. Point n is not an α-extreme; any α-circle passing through n contains at least another point of A. We use the same set A for illustrations 5–9 in this section. The set is chosen such that it covers all cases treated by the α-shape algorithm.

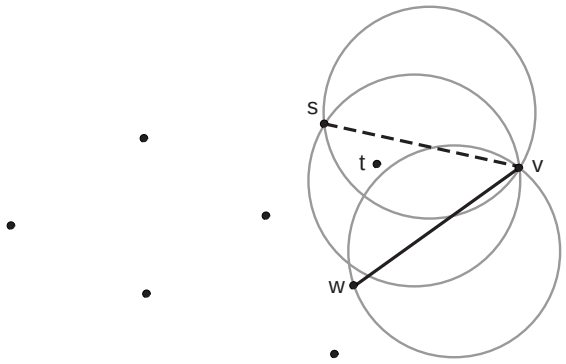

Fig. 6. Examples of α-neighbours: v and w are α-neighbours, s and v are not.

Two points are α-neighbours, if and only if an α-circle can be constructed through them, but with no other points of the set within that circle. Figure 6 illustrates α-neighbours in A. Points v and w are α-neighbours: there is a circle passing through v and w that does not contain any other point of A. Points s and v are not α-neighbours: there are only two circles with radius $-1/\alpha$ passing through both of them; both circles contain point t.

The tests for building the α-shape of a point set are based on the Delaunay triangulation of the set, its dual (in the graph theoretical sense), the Voronoi diagram, and relations between the two. A Delaunay triangulation is

a triangulation that maximizes the minimum angle [5]. A Voronoi diagram of a point set is a partition of the plane into convex polygons, one for each point of the set, such that each polygon contains only one point from the set that is its central point, and every point in a polygon is closer to its central point than to any other point of the set. Figure 7 shows the Delaunay triangulation (in light grey) and the Voronoi diagram (in thick light grey lines) of the point set A.

The α-shape of a point set S is a subset of the Delaunay triangulation of S [8]. It is built by first constructing the Delaunay triangulation of S, then testing for Delaunay edges whether they are in the α-shape. The complete algorithm is:

```
create a Delaunay triangulation DT of S
create Shape as an empty set of edges
for every edge (p, q) in DT
    if p is α-extreme and q is α-extreme and p and q are α-neighbours
        add (p, q) to Shape
    fi
rof
```

To build the Delaunay triangulation we use TRIANGLE, an open source tool created by Shewchuk [15]. Testing for α-extremes and α-neighbours of S, based on relations between the Delaunay triangulation and the Voronoi diagram of S, is explained in the next few paragraphs. Tests are translated into properties of the Delaunay triangulation and its convex hull, therefore this is the only structure needed for constructing the α-shape.

An α-extreme of a point set is a point on the boundary of the convex hull of the set, or a point which maximal circumradius[5] of triangles of which the point is a vertex, is bigger than the radius $-1/\alpha$. These properties are used to test for α-extremes. We explain them using Fig. 7. Points on the boundary of the convex hull of a point set have unbounded Voronoi polygons; any other point has a bounded Voronoi polygon. The point m is on the boundary of the convex hull of A; its Voronoi polygon $\mathrm{V_m}$ is unbounded. Any α-circle centred inside $\mathrm{V_m}$ that passes through m does not contain any other point of A. Point t is inside the convex hull of A; it has a bounded Voronoi polygon $\mathrm{V_t}$. The Voronoi polygon $\mathrm{V_t}$ of t contains all points that are closer to t than to any other point of A. This means that any circle passing through t and centred at a point inside $\mathrm{V_t}$ does not contain any other point of A. The maximal circle having this property (shown with dashed line) is the one centred at the furthest Voronoi vertex $\mathsf{c}_\mathsf{t}^{\max}$ from t. An α-circle passing through t, and lying inside the maximal circle of t, does not contain other points of A. Vertices of the Voronoi polygon $\mathrm{V_t}$ are the circumcentres of Delaunay triangles of which

[5] The circumradius is the radius the triangle's circumscribed circle, i.e., the unique circle that passes through each of the triangles vertices. The circumscribed circle is called circumcircle of the triangle.

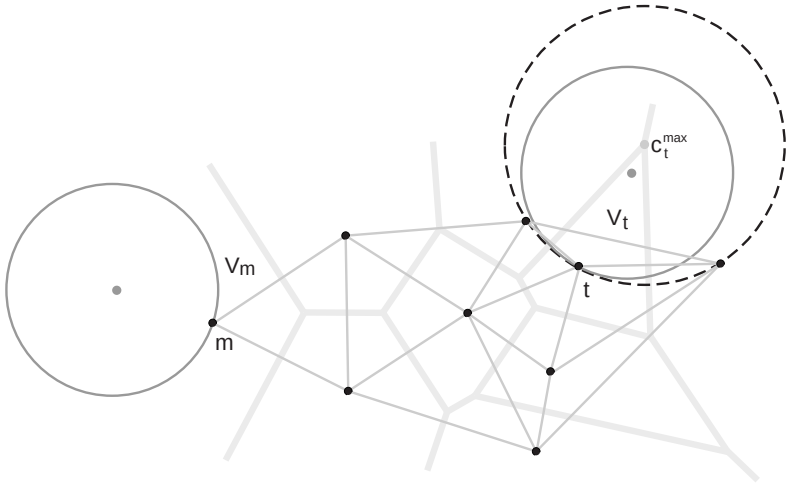

Fig. 7. Test for α-extremes: unbounded Voronoi polygon V_m and an α-circle (in grey) passing through m; bounded Voronoi polygon V_t, an α-circle passing through t, and the maximal circle of t (with dashed line).

t is a vertex. The maximal circle of t is one of the circumcircles of triangles t is a vertex of; it is the circumcircle that has the maximal radius.

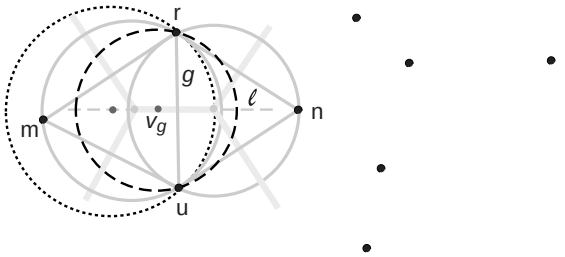

Fig. 8. Delaunay triangles sharing edge g and their circumcircles (in light grey); the corresponding Voronoi edges; and circles centred inside and outside edge v_g passing through end points of g (with dashed and dotted line, respectively).

The test for α-neighbours is based on the relation between Delaunay edges (D-edge) and Voronoi edges (V-edge). Let us first explain this relation using Fig. 8 that shows a partial Delaunay triangulation and Voronoi diagram of A. D-edges are shown in light grey, V-edges are shown with thick light grey lines. Every D-edge has a corresponding V-edge, e.g. D-edge g has its corresponding V-edge v_g. V-edge v_g joins the centres of the circumcircles of the two Delaunay triangles of which g is an edge. Circumcircles of Delaunay triangles are shown

in light grey. Any circle passing through the end points r and u of g has the centre in l, the perpendicular line to g at its midpoint (the dashed grey line). The V-edge v_g lies on this line. A Delaunay triangle has the property that its circumcircle does not contain any other point of the set. The two circumcircles of triangles sharing edge g have their centres at the end points of edge v_g. Any circle passing through end points of g and centred inside v_g has no other point from A, e.g. the circle in dashed line. Moving the centre of the circle along line l, outside v_g, produces a circle that encloses m or n, and possibly other points from A, e.g. the circle with dotted line in Fig. 8 encloses point m.

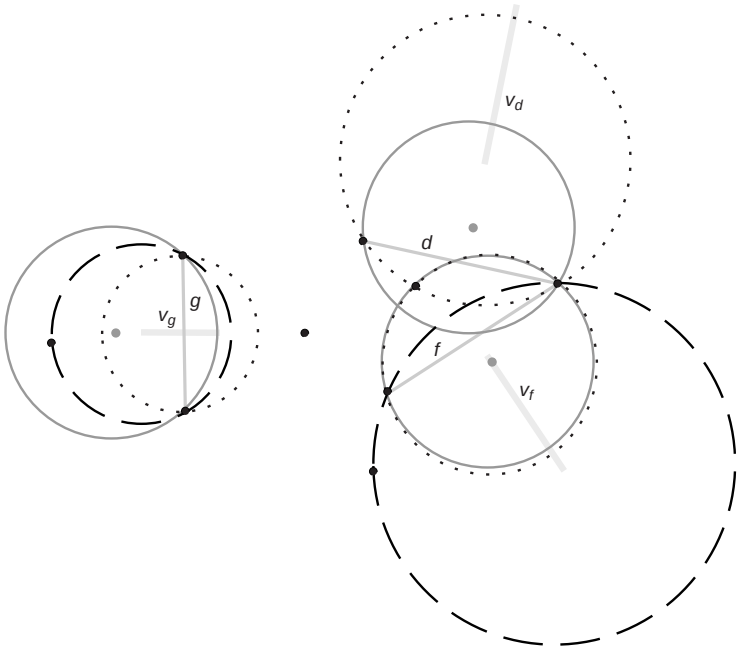

Fig. 9. Testing Delaunay edges for being in the α-shape by using min and max circles (with dotted and dashed lines, respectively), and α-circles passing through end points of D-edges (in grey).

Two α-neighbours are connected through a D-edge, and the centre of one of the α-circles passing through them is on the corresponding V-edge; no other point of the set lies inside the α-circle that has its centre in the V-edge. This property is used to test a D-edge whether its end points are α-neighbours, that is the D-edge is in the α-shape. The centre of an α-circle passing through end points of a D-edge is on the corresponding V-edge, if the radius $-1/\alpha$ is between the minimum and maximum distance from any of these points to the

V-edge. If a D-edge is in the boundary of the convex hull, its corresponding V-edge is a half line, in which case the maximum distance is infinite. The test reduces to checking whether the radius $-1/\alpha$ is bigger than the minimum distance. Figure 9 shows three different cases for calculating the minimum and maximum distance for a D-edge. Edge d is in the boundary of the convex hull A; its corresponding V-edge v_d is a half line. The minimum distance of an endpoint of d to the V-edge v_d is the circumradius of the triangle of which d is an edge. The circumcircle of this triangle is shown with dotted line. D-edge g intersects with its V-edge v_g. The minimum distance of a g endpoint to v_g is the half length of g (v_g is perpendicular with g at its midpoint). The minimum circle is shown with dotted line. The maximum distance is the distance from an endpoint of g to one of v_g endpoints. Endpoints of v_g are circumcentres of triangles of which g is an edge. The maximum distance is the maximum circumradius of the two triangles. The maximum (circum)circle is shown with dashed line. D-edge f does not intersect with its V-edge v_f. The minimum and maximum distance for f are respectively, the minimum and maximum circumradius of the two triangles of which f is an edge. The minimum and maximum (circum)circles of triangles sharing edge f are shown in dotted and dashed line, respectively. The α-circles and their centres are shown in grey. Only D-edge f is in the α-shape.

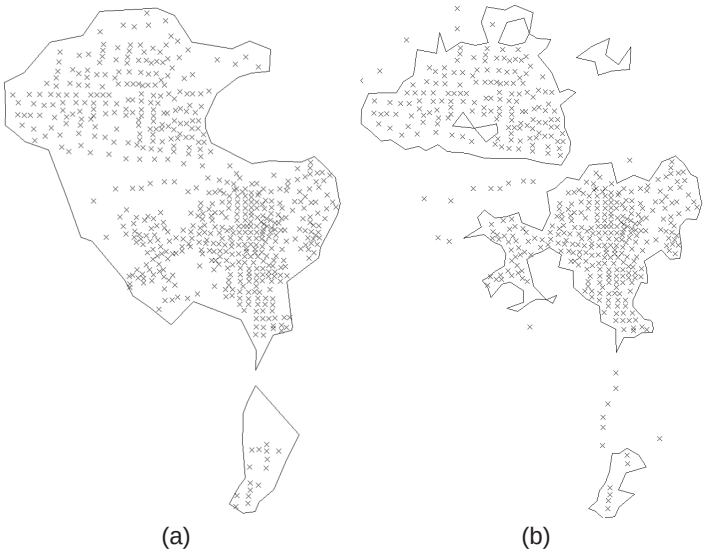

Fig. 10. (**a**) α-shape for the initial α-value, (**b**) α-shape for decreased α-value.

The value of α defines the level of detail of the α-shape. The criterion we use to set an initial α value is that every point is at least vertex of one triangle

in the α-shape interior. Figure 10(a) shows the α-shape taken from such α value, and 10(b) shows the shape after decreasing the α value. In Fig. 10(b) we see that there are loose points, not part of any shaped object. A triangle is part of the α-shape if and only if the radius of its circumscribed circle is smaller than or equal to $-1/\alpha$ [8]. To set an initial α value for a point set S, we calculate for every point $s \in S$ the minimum radius r_s of circumscribed circles of all Delaunay triangles that have s as a vertex. The radius $-1/\alpha$ is set to the maximum of r_s values for all points of S. The α value calculated from that is used to estimate the shape. This initial value can be further adjusted by the user, if needed.

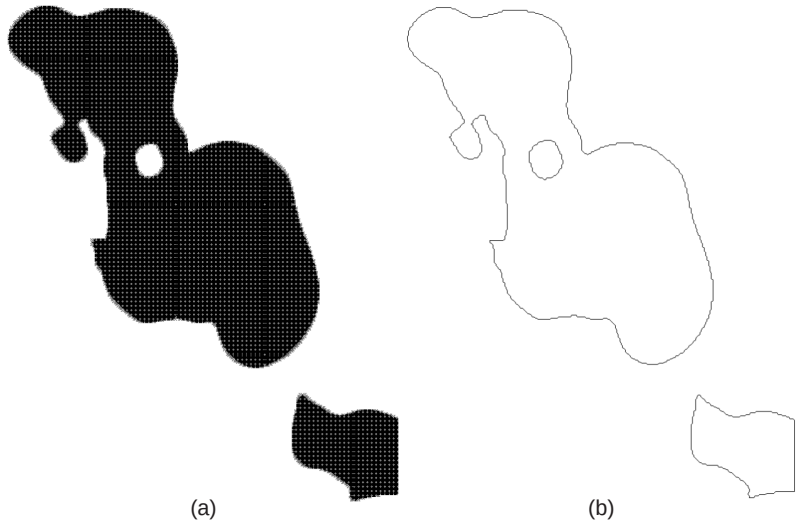

Fig. 11. (**a**) input from a regularly distributed point set, (**b**) object boundaries detected with the initial α-value.

The α-shape of regularly distributed point data (grid data) resulting from the use of the initial α-value is shown in Fig. 11. Grid points with membership value equal to zero have been excluded from the input.

3.2 Creating and storing triangulations

The output of α-shapes consists of boundary lines together with the input data points. The lines constitute boundaries of the support sets of simple vague regions. They are processed to identify objects by their boundary. An identifier is assigned to every simple region object detected. Each simple region is then built from the constrained Delaunay triangulation performed on boundaries of the region and points inside the boundary. Its triangulation

data is stored together with the identifier. Module `v.vague.triangle` performs the whole procedure. Each step of the procedure is explained in more detail in the following paragraphs.

Simple vague regions are identified from cluster boundaries. Each region may contain holes. Because GRASS does not support areas with holes, we have to check for them. A hole inside a simple vague region may contain other regions. Therefore, from cluster boundaries we have to detect which boundary is an outer boundary of a region and which is a hole boundary. We check for every area boundary if it lies inside other areas. It is the outer boundary of a simple vague region if it lies inside an even number of areas. It is the boundary of a hole if it lies inside an odd number of areas. Figure 12 shows two different configurations of outer boundaries and holes: area A is inside area B and it is a hole. Area C is inside area D which in turn is inside area E. The boundary of E and the boundary of D form (the boundaries of) a simple vague region. The boundary of C gives another simple vague region.

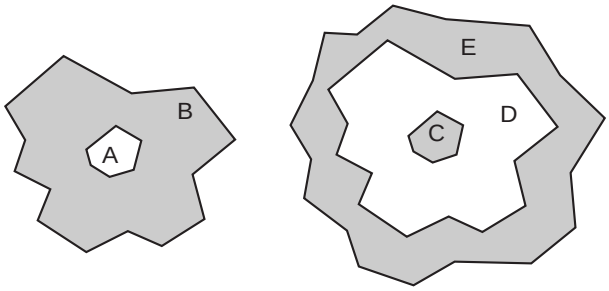

Fig. 12. Identifying simple vague regions (shown in grey) from areal boundaries: one simple vague region in the left; two simple vague regions in the right, one enclosing the other.

After identifying all simple vague regions, we perform a constrained Delaunay triangulation for every region. A constrained Delaunay triangulation is a triangulation of vertices with predefined edges. It consists of four steps [15]:

1. creating a Delaunay triangulation from the input vertices;
2. inserting missing line segments from the boundary and deleting the Delaunay edges that overlap with them;
3. removing triangles at concavities and holes;
4. adding more points in order to improve the quality of the triangulation.

Figure 13 illustrates the first three steps of the constrained Delaunay triangulation. Figure 13(a) shows boundaries of a simple vague region. For simplicity of illustration we consider a constrained Delaunay triangulation with only line boundaries as input. Figures 13(b)–(d) show the results from the first, second

and third step, respectively. To visualize change in different steps, the boundary edges are always drawn in black. In the two intermediate steps, the other edges are shown in dark grey, and the edges to be removed are shown with dashed grey lines.

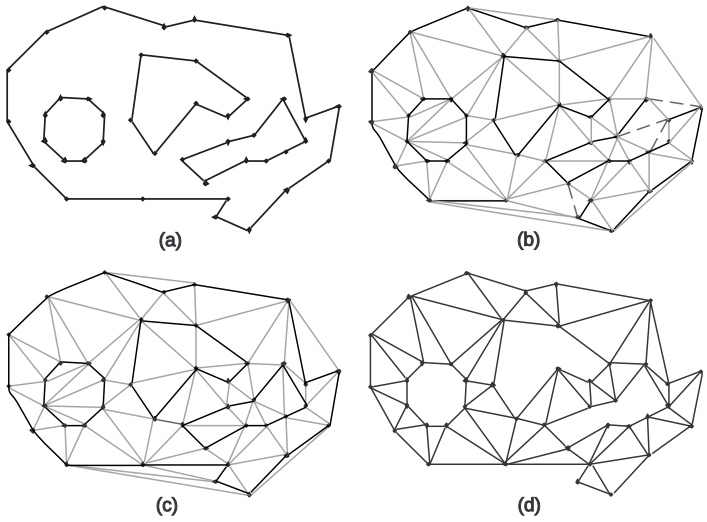

Fig. 13. Results of three steps constrained Delaunay triangulation in a simple vague region. (**a**) region boundaries, (**b**) Delaunay triangulation, (**c**) insertion of missing boundary lines, (**d**) removal of triangles outside the boundary and inside holes.

We use TRIANGLE by Shewchuk [15] to perform the constrained Delaunay triangulation. GRASS data is transformed to the TRIANGLE data format, the program is run on the transformed data, and its output is transformed back to GRASS vector format. The program cannot handle holes when the input is big (i.e. more than 50,000 points). We use TRIANGLE to perform the triangulation constrained only on the outer boundaries. Then we remove all edges inside holes. We expect the core of a region to have many input points, therefore many flat triangles will be created. We remove the redundant core triangles, by first constructing the boundary of the core from all flat triangles of value 1, then performing the constrained Delaunay on the boundary. Figure 14 shows triangulation of a simple vague region after the simplification of its core triangulation. Points are very dense outside the core, which makes the triangulation very dense. Core triangles are visible after the simplification phase.

For each simple vague region, triangulation edges are stored as *boundary* lines with (x, y, z) coordinates in the `coor` file. An attribute is used to store the identifier of the simple vague region to which the edge belongs. Topology

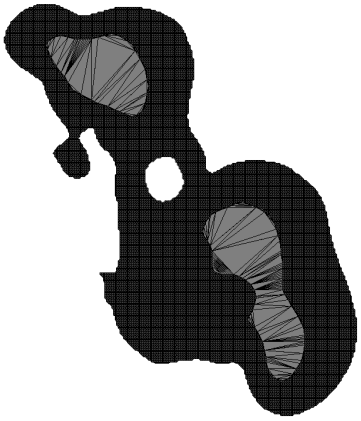

Fig. 14. Triangulation of a simple vague region after reducing core triangles. Triangulation is very dense outside the core; only core triangles are distinguishable.

data of triangles and holes, e.g. indices of line boundaries, is stored in the `topo` file.

The output of module `v.vague.triangle` is a data layer that contains information about a vague class. The complete algorithm of the module is presented below:

create a list SVR of simple vague regions from cluster boundaries
for every region r in SVR
    find all points that are inside its boundary
    build constrained Delaunay triangulation from r's boundary and points
    if r has holes
        remove edges inside every hole
    store triangulation edges together with r index in SVR
    store topology for triangles and holes
rof

Every time a layer of simple vague regions is used, its information is compiled from stored data, and put in memory in a list Vague_region of simple vague regions. Every element of Vague_region contains a list of triangles and a list of holes. Information about triangles and holes is built using several GRASS data structures that are inside a main structure, Map_info [11]. Every time a vector layer is used, data from `coor` and `topo` files is read and put into these structures. The indexed lists Area and Line inside a Map_info structure contain topology information of areas and lines, respectively. Each element of Area contains a list of indices of lines that constitute its boundary. The index of the Line list is used to connect every element with the corresponding element in another list that contains attribute values. Figure 15 shows relations between these data structures.

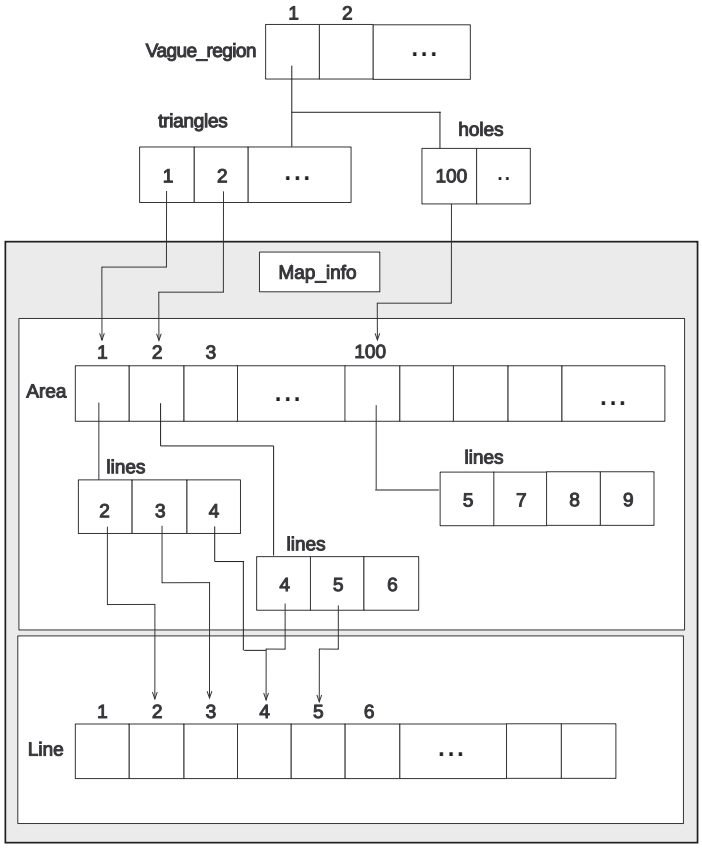

Fig. 15. GRASS structures (inside Map_info) used to store information about simple vague regions.

3.3 Creating themes from several layers

A data layer keeps information about one class. For example, a layer `forest` keeps data about simple vague regions that belong to a class 'forest' of a 'vegetation' theme. The vegetation theme consists of several classes: forest, grassland, shrubs, etc. Vague classes could overlap with each other, e.g. forest and shrubs, in which case triangulations representing them will overlap. Overlapping lines or areas are not allowed (handled) by the topology, which we need for compiling object information. Therefore we cannot store all classes together. However, we often need to have all classes of a theme together, to operate with all or a selected subset of them.

We create two tables to store the relation class and theme it belongs to. Figure 16 shows their schemas. Tables are stored in DBF format, which is a format integrated in GRASS, meaning that no connection to an external data-

base is needed. Table `Themes.dbf` stores theme name and description, respec-

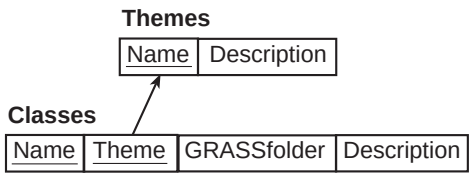

Fig. 16. Tables that keep information about themes and classes belonging to each theme.

tively in Name and Description. Name is the identifier. The table `Classes.dbf` stores class name, the name of the theme it belongs to, the name of the GRASS folder where (layer) data is stored, and class description, respectively in Name, Theme, GRASSfolder, Description. The combination Name, Theme is the identifier of the table. Theme refers to `Themes.dbf` table.

Module `v.vague.combine` binds several layers in a theme. The created theme is added to the `Themes.dbf` table and its list of layers is added to the `Classes.dbf` table (each layer as a new record in the table). The module allows users to add or remove layers from an existing theme, or to delete a theme. Changes are then reflected in `Classes.dbf` and `Themes.dbf` table.

A theme of vague region layers forms a vague partition. Objects in a layer do not overlap with each other. This is assured by the triangulation process. When adding layers to a theme with the `v.vague.combine` we check if objects from different layers overlap only in their uncertain part. We give a warning when this criterion is violated, and store a report for the violating cases (object identifiers, layers they belong to, and the theme).

4 Visualizing objects and displaying information

Visualization of objects in a layer is done using colour brightness to display levels of membership values, e.g. using grey scales. Different layers are displayed using different colour hue for each layer, and brightness for membership value on each layer. Objects can be selected by clicking. Selected objects are shown in a separate colour, not used for displaying layers. Figure 17 shows a theme of vague regions having two classes. Different colour hue is used for each class. Darker colour shows higher membership value. An object is selected and shown in yellow colour.

Module `v.vague.what` performs visualization of layers. It allows to select objects from a layer, and displays information for a given location in a layer. A theme (created by `v.vague.combine` module) is the input for this module. Vague regions can be displayed by drawing only triangle edges of their triangulations. The set of triangles is drawn with a different colour for each

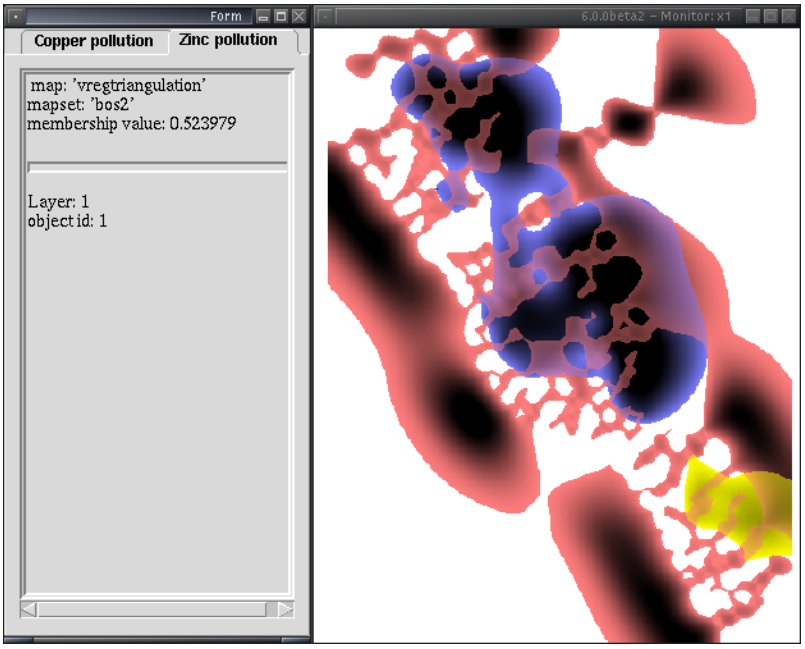

Fig. 17. Visualization of several layers, each drawn in separate colour hue. A selected object is shown in yellow.

layer. The interpolated values inside triangles can be used to colour the full extent of objects in a layer, as it is shown in Fig. 17. Section 4.1 explains more thoroughly the techniques used for the visualization.

The interface of `v.vague.what` uses two windows: the 'control window' that contains the list of layers, one layer in a separate tab, and the 'display window' for drawing the layers. Layers are first drawn in the order in which they are stored in the theme. The selected layer (tab) in the control window becomes the top layer in the display window. An object of the top layer can be selected by clicking at a location inside the object. The triangle containing the given location is found first, then the object the triangle belongs to. The object is highlighted. The information of the given location is displayed in the control window. This information contains the membership value of the location, the directory storing the layer data, and the identifier of the object (it falls in). The membership value is calculated by using linear interpolation inside the triangle containing the location. The user can select several objects and output them to a new layer.

4.1 Visualization techniques

A GRASS module, `d.vect`, is used to draw vague region layers with triangle edges. Each layer is drawn in a different colour. A new module, `d.vague`, is

created to fill triangles with interpolated values and draw them. The module uses a scanline rendering algorithm [20]. The rest of the section describes how this module works.

GRASS functions are used to map the coordinates of triangle vertices to screen coordinates. A screen is a raster map consisting of pixels. Every pixel has a red, green, and blue value, specifying its colour. Scanline rendering fills a triangle by drawing a line at a time, starting at the top of the triangle. All pixels of a line are drawn, which are part of this triangle. The line is then moved down and the drawing is repeated until the bottom of the triangle is reached. Figure 18 illustrates how the algorithm draws a triangle. Linear

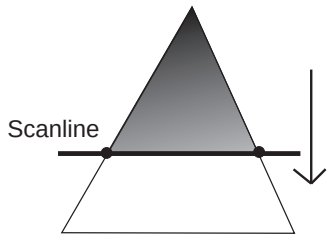

Fig. 18. Scan-line visualization technique.

interpolation is used to calculate the membership value at every location. Every triangle lies on a plane defined by the three triangle vertices. Any point (x, y, z) of the plane satisfies the equation $z = ax + by + c$. The a, b, and c values are calculated from the coordinates of the three triangle vertices. The membership value for any point in the triangle is calculated by replacing its (x, y) location in the above equation.

A different colour is used for each layer, selecting only colours that have the same saturation, so no colour draws more attention than others. The brightest colour is specified for every layer. The saturation and hue are kept constant for a layer. The brightest colour is used for the lowest membership value of the layer. The colour with brightness equal to 0 is used for the membership value equal to 1. The corresponding brightness value for any other membership value is calculated by first inverting the $[0, 1]$ interval (of membership values) then stretching it linearly to the range of brightness values. Because the monitors work with RGB colours, we use a function that applies that idea in an RGB model. The function that maps memberships value to RGB colours is

$$f : [0, 1] \rightarrow \mathbb{R}^3 \text{ such that for every } \lambda \in [0, 1]$$
$$f(\lambda) = (1 - \lambda) \times (R_{max}, G_{max}, B_{max}),$$

where $(R_{max}, G_{max}, B_{max})$ is the specified brightest colour of the layer.

If multiple layers overlap, transparency is used to draw them together. This is calculated with alpha-blending [20]. The colour $(R, G, B)_{new}$ at every

location in the overlapping part is calculated as

$$(R, G, B)_{new} = 0.5 \times (R, G, B)_{S_1} + 0.5 \times (R, G, B)_{S_2},$$

where $(R, G, B)_{S_1}$ and $(R, G, B)_{S_2}$ are the colours at that location for the top and bottom layer respectively. When more than two layers are to be drawn, first the two bottom layers are calculated. Then for every layer to be added on top of them, the above formula is reused, replacing $(R, G, B)_{S_2}$ with the calculated colour for the previous layers, and $(R, G, B)_{S_1}$ the colour of the new layer.

5 Operators on vague objects

The operators we implemented are union, intersection, and difference. This section describes shortly these operators for vague points and vague lines, for being simple, and concentrates more on the operators for vague regions.

The operators for vague points take two Vpoint layers as input, and output a new Vpoint layer. The three operators check first for simple point objects in the two input layers that have identical location. Union operator selects from each pair (of identical location points) the point with the higher membership value, and puts it in the result layer. It also adds all other (unmatched) points from both layers to the result layer. The intersection operator selects the point with the lower membership from each pair of matched objects, and inserts them in the result layer. For each pair of matched points, the difference operator calculates a new membership value (the formula provided in Sect. 2), and inserts in the result layer a point with the common location and the calculated membership value. It also adds to the result layer all unmatched points from the first layer.

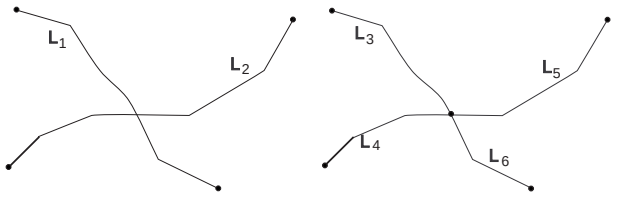

Fig. 19. Union of two simple vague lines.

The union operator for vague lines takes two Vline layers as input, and outputs a new Vline layer. It first checks for simple lines that intersect. The operator splits the intersecting lines at the intersection point, which becomes common node for the newly created lines. The membership value at the common node is the maximum value of the memberships at this location in the

two initial lines. The new lines are inserted in the result layer. Lines that do not intersect are directly added to the result layer. Figure 19 illustrates union of vague lines: lines L_1 and L_2 from input layers are intersecting; lines L_3, L_4, L_5, L_6 are created and added in the result layer. The intersect operator for vague lines takes two Vline layers as input, and outputs a Vpoint layer. It checks for intersecting lines, creates a point at the intersection location with membership value the minimum of the two lines at this location, and inserts these points in the result layer.

Operators for vague regions take two Vregion layers as input, and output a Vregion layer. A Vregion layer is a surface that consists of several non-overlapping triangulations. Triangulations belonging to different surfaces might overlap. The overlapping zone between two surfaces is important for the operators, because they treat differently triangles inside and outside the overlapping zone. The overlapping zone may consist of several separate areas. Operators start by detecting the intersection line between triangulations. The line separates the input triangulations into parts that are used to construct the output surface. Union is built by taking the higher triangulation inside an overlapping area, and adding unchanged triangulation parts that are outside the overlapping zone. Intersection is built by taking the lower triangulation inside any overlapping area. The difference operator re-calculates values at triangulations inside an overlapping area, and adds unchanged triangulations of the first surface that are outside the overlapping zone. The basic steps for the operators are:

1. Add vertical faces along boundaries of triangulations on both surfaces;
2. Detect the intersection line;
3. Re-triangulate both surfaces with this intersection line;
4. Select the right triangles from the re-triangulated surfaces to build the result.

The first three steps are the same for union and intersection operators. The forth step, selecting triangles for the output, is different. We explain the first three steps, and give the algorithm of the forth step for union. This algorithm can be used for the intersection with only few changes, which are described shortly. Difference operator requires specific treatment in most of the steps, therefore we explain it separately.

To add vertical faces along a triangulation boundary we consider every edge of the boundary. Two vertical triangles are created for each edge and added to the triangulation. Suppose the edge is defined by points $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$. The face determined by p_1, p_2 and their projections in plane, $p_3 = (x_1, y_1, 0)$ and $p_4 = (x_2, y_2, 0)$ is split into two triangles, and these are added to the triangulation.

Detection of the intersection line is performed using the GNU triangulated surface library (http://gts.sourceforge.net). It produces the set of looped lines where the two surfaces intersect. The intersection between two triangles can be a line segment, or a polygon if the triangles lie in the same plane. There-

fore, the intersection line between two surfaces consists only of straight-line segments.

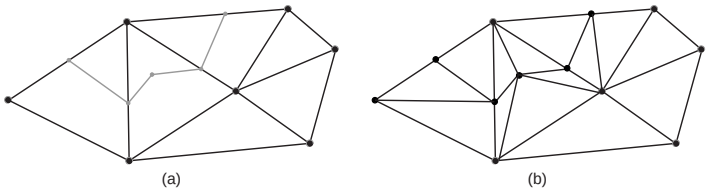

Fig. 20. Re-triangulation: (**a**) old triangles and intersection line in grey, (**b**) new triangles after re-triangulation.

The intersection line is added as constraint to the triangulations of both surfaces. On each surface, only triangulations containing a part of the intersection line are to be re-triangulated. For every triangulation (to be changed), the triangles containing a line segment from the intersection are split into several new triangles. The other triangles are added unchanged to the new triangulation. The splitting of triangles is done by greedy triangulation, as described in [21]. Figure 20 illustrates the splitting of three triangles. The algorithm below explains how splitting is done. Let V_i be the set of vertices of the triangle i, and L_i the set of line segments of the intersection line passing through the triangle i. Let us denote by E_i the set of edges of the new triangulation in triangle i, and P_i the union of V_i with the end points of L_i.

set $E_i = L_i$
for all points p from P_i
  create D_p as edges from p to any other point in P_i
    sorted in ascending order on length
  for every edge $d =$ (p,q) in D_p
    if $d \notin E_i$ **and** no edge $e \in E_i$ intersects with d
      and no point $r \in P_i - \{p, q\}$ lies on d
        add d to E_i
    fi
  rof
rof

After the re-triangulation, the relation of a triangle from one surface to any triangle in the other surface is one of the three cases:

- The three vertices of the triangle are on the other triangle: the triangle is part of an area that is contained in both surfaces.
- One or two vertices are on the other triangle: the triangle intersects the other surface only at a point or along a line (the edge joining the two vertices). The rest of the triangle is above or below the other surface.

- No vertex of the triangle is on the other surface: The triangle does not intersect with the other surface. It is completely above or below the other surface.

A point is above a surface when its z coordinate is higher than the z value of the surface at the point location. A point in the interior of a triangle from one surface determines the relation of this triangle with the other surface. When an interior point is above, on, or below the other surface, the whole triangle is above, on, or below the other surface, respectively. The centre of the incircle of a triangle[6] is always in the interior the triangle, and can therefore be used for such testing.

The intersection line consist of several looped lines. Triangles of one surface that are inside a looped line are all above the other surface, or all below the other surface. So the union of two surfaces is formed by groups of triangles bounded by these looped lines. The algorithm that performs union of two re-triangulated surfaces S_1 and S_2 and outputs a surface S is given below:

```
set S to an empty surface
for any triangle t from surface S1
    generate a point p(x,y,z) in the interior of t
    if S2 exist in location (x,y)
        if p is on or above S2
            add t to S
    else
        add t to S
    fi
rof
for any triangle t from surface S2
    generate a point p(x,y,z) in the interior of t
    if S1 exist in location (x,y)
        if p is above S1
            add t to S
    else
        add t to S
    fi
rof
```

After the output surface is created, separate triangulations are detected and given an object identifier. They are the SVregion objects of the result layer.

The algorithm for calculating the intersection is quite similar. The difference is that a triangle in one surface is discarded if the other surface does not exist at the interior location of the triangle. Also, the testing for the z value of a point and a surface is reversed: the point should be on or below the surface.

[6] The incircle is the inscribed circle of a triangle, i.e. the unique circle that is tangent to each of the triangle's edges.

Difference between surfaces S_1 and S_2 is the intersection of S_1 with the complement of S_2. Outside S_2 its complement is equal to 1. The values of S_1 are everywhere smaller or equal to 1, therefore the intersection outside S_2 is equal to S_1. Thus, we only need to build the complement of S_2 inside its boundaries. Surface S_2' is built from S_2 triangulation by inverting the values of each triangle vertex. The core of S_2 will be a hole for S_2', therefore core triangles are not put in S_2'. Holes of S_2 constitute the core of S_2'. Boundary of each hole is triangulated and included in S_2', all as flat triangles with value 1.

To build the result surface we pass through the same steps of union and intersection. First vertical faces are added along boundaries of S_1 and S_2'. For S_1 faces are added from the boundary line to its projection in the horizontal plane, that is down to membership value 0. For S_2' vertical faces are built along the boundary up to membership value 1. The intersection line between the two surfaces (extended by the vertical faces) is detected, and both surfaces are re-triangulated with this line. The forth step, selecting the right triangles, is quite similar with the intersection operator, except that every triangle of the first surface is included in the output if the second surface does not exist at its location(s). We keep to notations S_1 and S_2' to denote now the surfaces taken after re-triangulation. The algorithm for the last step, selecting the right triangles for the output surface, is

```
set S to an empty surface
for any triangle t from surface S_1
    generate a point p(x,y,z) in the interior of t
    if S_2' exist in location (x,y)
        if p is on or below S_2'
            add t to S
    else
        add t to S
    fi
rof
for any triangle t from surface S_2'
    generate a point p(x,y,z) in the interior of t
    if S_1 exist in location (x,y)
        if p is below S_1
            add t to S
rof
```

As for the other operators, after the output surface is created, separate triangulations are detected and given an object identifier. Triangle edges of every triangulation are stored together with this identifier in the new layer.

6 Discussions

The current implementation of α-shapes gives good results for data sets with more or less regular sample density. If the sample density changes too much,

the α-shapes do not work very well. Density-scaled α-shapes [19] provide a solution to this. Implementation of their algorithm would make the clustering process more robust. On the other side, density-based clustering algorithms (pointed out by a reviewer), e.g. DBSCAN, OPTICS, would be another possible solution to clustering of data points. It seems though that the technique aims at clustering of points, and does not provide for delineation of cluster boundaries.

The only input type we consider for vague regions is data points. Data about vague regions could also be (vague) lines, e.g. lines having the same membership value. A part of the procedure to create vague regions from line input would be the same as for point input. They both need the constrained Delaunay triangulation. The identification of objects would need another technique.

The operators we implemented are only part of a complete set of spatial operators, e.g. those offered by ROSE algebra [9, 10]. These are the operators returning spatial objects (types). From this group we left out the difference between vague lines, for having a certain complexity. Besides, we consider it less important than operators on vague regions, therefore we gave priority to their implementation. Other spatial operators would make use of these implemented basic operators. Several spatial relations (which definitions we have provided in [6]) are based on intersection of vague objects. Other spatial relations are based on operators like bounded difference, and absolute difference. They would therefore need an extension of these basic operators for their implementation.

The membership function of a *simple vague region* can have discontinuities along lines. These discontinuities result in vertical faces in triangulations. The work presented in this paper does not consider discontinuity lines. We build and store the topology of triangulations using GRASS topology, which ignores vertical faces and areas adjacent to them. We do not need to store vertical faces, but we do need triangles adjacent to them. To allow discontinuity lines we could build separate functions for the topology of triangulations from GRASS topology functions, modifying the last. GRASS topology builds the connectivity of 3D lines (through their nodes) correctly, but considers only their x, y coordinates when building areas from lines. We can use line connectivity to build the adjacent triangles to vertical faces, changing the existing functions to consider the special cases we need. Operators on vague regions are to be modified in order to consider discontinuities.

7 Conclusions

The paper shows how vague spatial objects can be stored and manipulated using existing GIS functionality for vector data format. Known spatial data types and structures were employed to construct vague object types. Points, lines, and triangulations were used to store simple vague points, lines, and

regions, respectively. These simple vague types represent identifiable objects. Classes of simple objects were stored in separate vague layers. Classes a certain theme can be bound together via relations saved on database tables. A theme of vague region classes forms a vague partition, which allows for a soft classification of space that is important for many spatial applications.

Few modules were offered to handle vague objects: a module that creates layers of vague objects from input data points; a module that visualizes vague layers in the screen, and allows to retrieve and display information about their objects; some modules that perform different operations on vague layers. Union, intersection, and difference operators were implemented for vague objects. These are basic operators, on which other spatial operators can be built.

References

1. P. Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, Inc., San Jose, 2002.
2. P. A. Burrough and A. U. Frank. *Geographic objects with indeterminate boundaries*. Number 2 in GISDATA. Taylor & Francis, London, 1996.
3. E. Clementini and P. di Felice. A spatial model for complex objects with a broad boundary supporting queries on uncertain data. *Data & Knowledge Engineering*, 37(3):285–305, June 2001.
4. A. G. Cohn and N. M. Gotts. Representing Spatial Vagueness: A Mereological Approach. In L. Carlucci Aiello, J. Doyle, and S. Shapiro, editors, *Principles of Knowledge Representation and Reasoning (KR'96)*, pages 230–241. Morgan Kaufmann, 1996.
5. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry-Algorithms and Applications*. Springer, Berlin, 2nd edition, 1997.
6. A. Dilo, R. A. de By, and A. Stein. A proposal for spatial relations between vague objects. In L. Wu, W. Shi, Y. Fang, and Q. Tong, editors, *Proceedings of the International Symposium on Spatial Data Quality ISSDQ'05, Beijing, China*, pages 50–59, Peking University, Beijing, China, August 2005. The Hong Kong Polytechnic University.
7. A. Dilo, R. A. de By, and A. Stein. A spatial algebra for vague objects. Submitted for publication, January 2005.
8. H. Edelsbrunner, D. G. Kirkpatrick, and S. R. On the shape of a set of points in the plane. *IEEE Transactions on Information Theory*, IT-29(4):551–559, July 1983.
9. R. H. Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–399, 1994.
10. R. H. Güting and M. Schneider. Realm-based spatial data types: The ROSE algebra. *VLDB Journal*, 4(2):243–286, 1995.
11. P. Kraipeerapun. Implementation of vague spatial objects. Master's thesis, International Institute for Geo-information Science and Earth Observation (ITC), March 2004.
12. A. J. Roy and J. G. Stell. Spatial relations between indeterminate regions. *International Journal of Approximate Reasoning*, 27(3):205–234, September 2001.

13. M. Schneider. Uncertainty management for spatial data in databases: Fuzzy spatial data types. In *SSD '99: Proceedings of the 6th International Symposium on Advances in Spatial Databases*, volume 1651 of *Lecture Notes in Computer Science*, pages 330–351. Springer-Verlag, 1999.
14. M. Schneider. Design and implementation of finite resolution crisp and fuzzy spatial objects. *Data & Knowledge Engineering*, 44(1):81–108, January 2003.
15. J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *First Workshop on Applied Computational Geometry*, pages 124–133, Philadelphia, Pennsylvania, May 1996.
16. R. S. Sicat, E. J. M. Carranza, and U. B. Nidumolu. Fuzzy modelling of farmers' knowledge for land suitability classification. *Agricultural Systems*, 83(1):49–75, January 2005.
17. P. Smets. *Handbook of Fuzzy Computation*, chapter Theories of Uncertainty. Institute of Physics Publishing, May 1999.
18. R. Sorensen. Vagueness. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab CSLI, fall 2003 edition, 2003.
19. M. Teichmann and M. Capps. Surface reconstruction with anisotropic density-scaled alpha-shapes. In *In IEEE Visualization '98 Proceedings*, pages 67–72, San Francisco, CA, October 1998. ACM/SIGGRAPH Press.
20. A. Watt and F. Policarpo. *3D Games, Real-time Rendering and Software Technology*. Pearson Education Limited, Essex, England, 1st edition, 2001.
21. M. F. Worboys. *GIS, a Computer Perspective*. Taylor & Francis, 1st edition, 1995.
22. M. F. Worboys and E. Clementini. Integration of imperfect spatial information. *Journal of Visual Languages & Computing*, 12(1):61–80, February 2001.
23. B. F. Zhan. Topological relations between fuzzy regions. In *Proceedings of the 1997 ACM Symposium on Applied Computing*, pages 192–196. ACM Press, 1997.