Map Overlay with Weighted Vertex Collections*

José Augusto Sapienza Ramos
COPPE - Programa de Engenharia de Sistemas e Computação da Universidade Federal do Rio de Janeiro
Rio de Janeiro, Rio de Janeiro, Brazil 21945-970
sapienza@cos.ufrj.br

Claudio Esperança
COPPE - Programa de Engenharia de Sistemas e Computação da Universidade Federal do Rio de Janeiro
Rio de Janeiro, Rio de Janeiro, Brazil 21945-970
esperanc@cos.ufrj.br

ABSTRACT

This work proposes a data structure called Weighted Vertex Collection (WVC) to represent region maps, i.e., polygonal subdivisions of the plane. The key idea consists of eschewing the common representation of such maps as sets of labeled polygons. Rather, region maps are viewed as discrete scalar fields, which can be represented directly by WVCs. These structures can be used to efficiently compute several important operations on region maps, including many that are usually performed with raster representations. In this paper we focus on how to compute the overlay of region maps using line sweeping algorithms. A proof-of-concept implementation was used to conduct several experiments that aim at comparing our approach with other well-established algorithms for computing map overlay operations. These tests indicate that WVCs provide a clean way to express overlays, being competitive with polygon-based approaches, especially when complex evaluation plans are required.

CCS CONCEPTS

• **Information systems** → **Geographic information systems**; *Data structures*; • **Theory of computation** → **Computational geometry**; • **Computing methodologies** → *Computer graphics*;

KEYWORDS

map overlay, geographic data, weighted vertex collections

ACM Reference format:
José Augusto Sapienza Ramos and Claudio Esperança. 2017. Map Overlay with Weighted Vertex Collections. In *Proceedings of ACM SIGSPATIAL conference, Redondo Beach California USA, November 2017 (SIGSPATIAL'17),* 12 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

In recent years, technologies that deal with digital maps have grown in importance. Geo-referenced data acquisition and sharing are now easier and more widespread than ever, helping us map and understand 500 million square kilometers of the Earth's surface. As a result, geographical datasets have grown in size and complexity. Moreover, this kind of data exceeds the capacity of today's computing technologies [35], continuously demanding more efficient algorithms. This scenario sparked the development of concepts, proposals and implementations involving complex geographic data infrastructures such as Spatial Data Infrastructures [7] and/or Spatial Big Data [1, 35].

Typical spatial database systems must implement two essential classes of operations: spatial selection queries and spatial joins. According to Güting [18], a spatial selection query returns a subset of tuples based on a spatial predicate, such as a window operation, for instance, that aims to determine what geographic features must be shown at a given zoom level on an interactive map. On the other hand, the spatial join is a particular kind of join query where the predicate being tested is spatial in nature, i.e., consists of geometrical relationships such as intersection, distance or containment. This kind of result frequently contains derived attributes involving geometric operations such as intersection, union, buffer, and others.

A *Map Overlay* is a special kind of spatial join. It combines two or more maps into a single new map [24]. The objective is to process spatial relationships between the input maps, usually involving complex conditions. Commonly, this kind of operation is performed on regions represented by polygons. In mainstream Geographic Information Systems (GIS) software, common Map Overlay operations are usually called "Geoprocessing Tools". However, other advanced GIS operations such as raster calculations – also called raster overlay – are also considered Map Overlay operations. It is worth noting that this kind of operation is also very important in Computer Aided Design (CAD), Very-Large-Scale Integration (VLSI) and other Computer Graphics applications.

Several related works propose algorithms that represent polygonal regions as polygon sets, where each polygon is encoded as a circular list of vertices, also called vertex circulations. In this work, instead, we describe the Weighted Vertex Collection (WVC), capable of representing a polygonal map as a discrete scalar field delimited by polygonal lines. In this way, each point of the plane corresponds to a numerical value that identifies the region to which it belongs. Note that this is similar to the idea used in raster GISs, except that WVCs are not dependent on the resolution of the raster representation. In the following sections we describe the WVC data structure and how it can be used to process typical map overlay operations efficiently. In particular, we show how these operations can be viewed as special cases of an operation called *scalar transformation*, which can be computed using a technique known in Computational Geometry as plane (or line) sweeping. Also, we present the results of several experiments aiming to assess how WVCs fare with respect to traditional approaches for computing map overlays.

*Produces the permission block, and copyright information

SIGSPATIAL'17, November 2017, Redondo Beach California USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

Algorithm	Characteristics (*)	O
Vatti [38]	$\uparrow \subset$	n^2
Greiner–Hormann [17]	$\subset$	n^2
Lui et al. [25]	$\bowtie \subset$	n^2
Foster–Overfelt [12]	$\bowtie \subset$	–
Margalit–Knott [26]	$\subset$	n^2
Martinez et al. [27]	$\bowtie \uparrow \subset$	$n \log n$
Wang et al. [39]	$\bowtie \uparrow T$	$n \log n$
Feito et al. [10]	$\uparrow \triangle$	n^2
Simonson [37]	$\bowtie \uparrow \partial \sqsubset$	$n \log n$
WVC	$\bowtie \uparrow \angle \sqsubset$	$n \log n$

(*)	Symbol meaning
$\bowtie$	Handles polygon sets explicitly (not only two polygons).
$\sqsubset$	Handles unlimited polygons.
$\uparrow$	Uses line sweeping.
$\subset$	Classifies edges for containment in the other polygon.
T / $\triangle$	Decomposes polygon into trapezoids / triangles.
∂	Uses the vertex-derivative concept.
$\angle$	Uses weighted vertex concept.

Table 1: Some algorithms to compute polygon overlays, and their characteristics. Column O records the asymptotic time complexity claimed in the respective work – except in Vatti's row, where the complexity was estimated by Greiner and Hormann [17].

2 RELATED WORK

Nowadays, the most widely adopted strategy in GIS applications for dealing with geometric operations of spatial joins (thus, also map overlays) is known as the *filter-and-refine* strategy – see a discussion in [4, 23, 40]. It encompasses at least two steps: (1) the filtering step quickly discards pairs of the cartesian product that do not satisfy the spatial predicate, usually implemented either by a nested loop or with the use of spatial indices such as Quadtrees, R-Trees and their variants – see Samet [33] for a survey of these structures; and (2) the refinement step, that processes remaining candidate pairs to construct the final result set.

Table 1 contains a summary of algorithms that have been proposed in the past to compute set-theoretic operations on polygons. These works were selected because they are able to process generic polygons, i.e, not necessarily convex and possibly containing holes or self-intersections. From these, the polygon overlay algorithms proposed by Vatti [38] and Greiner-Hormann [17] are among the most well-known. Vatti's algorithm applies plane sweep to compute intersections followed by a traversal analysis of double linked lists of vertices. However, as observed in Wang et al. [39], this algorithm requires complex and specific data structures, and hence is relatively complicated to be implemented.

On the other hand, Greiner-Hormann's algorithm is easier to implement, but faster only for small datasets as shown in Martinez et al. [27] and Lui et al. [25]. Martinez et al. [27] argue that this algorithm loses performance for large datasets, since it uses a brute force approach. Nevertheless, this algorithm must employ a perturbation scheme to handle degenerations when a vertex of a polygon lies on an edge of another. Martinez et al. [27] and Kim and Kim [22] demonstrate that different perturbations may change the result.

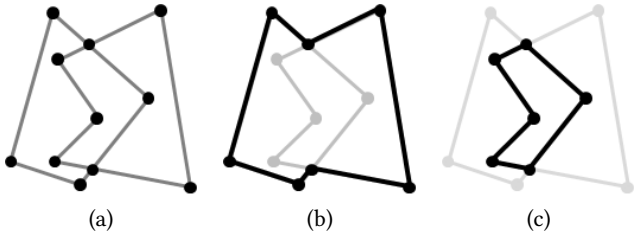

Figure 1: (a) Two overlapping polygons already with their edges split at intersections; (b) selecting edges (with dark color) that are not contained in the other polygon to build the union result; (c) similarly, the intersection result can be computed by selecting polygon edges which are contained in the other polygon. Adapted from Greiner–Hormann [17].

Lui et al. [25] adapt Greiner-Hormann's algorithm to compute polygons with holes and polygon sets, also proposing some optimizations. However, they do not contribute to the degeneration problem. In addition to Greiner-Hormann's algorithm, Foster–Overfelt [12] combines the method proposed by Kim and Kim [22] and the method mentioned by Liu et al. [25] to address degenerate cases more appropriately.

Another classical reference is the work of Margalit-Knott [26], where mathematical proofs and practical implementation issues are widely discussed. Their algorithm uses two linked lists and one hash table to classify polygon faces and to construct the final geometries. In order to handle all combinations between edge classifications and types of operations on polygons, the authors propose a table-driven algorithm.

In their turn, Martinez et al. [27] propose an algorithm in three steps: (1) subdivide the edges of the polygons at their intersection points using plane sweeping; (2) select subdivided edges that lie inside the other polygon – or for some operations, also those outside; and (3) join the edges selected in step 2 to build the resulting polygon. Although this work presents relatively few tests, the authors claim that it is more scalable than Vatti's and Greiner–Hormann's algorithms for large datasets.

We notice that all algorithms cited above apply a common general strategy: compute the intersection points in vertex circulations, classify each polygon edge as contained or not contained in the other polygon (this strategy is denoted with symbol $\subset$ in Table 1), and compute the final geometries by filtering and merging these edges. Figure 1 illustrates this concept. Most implementations use a combination of linked lists and plane sweeping to process these steps in an efficient way. On the other hand, Simonson et al. [37] present a different proposal consisting of the concept of partial vertex derivatives processed with plane sweeping. In a nutshell, this concept maps each polygon interior to +1 and adds them at intersections, where, for instance, a plane region mapped to +3 indicates an intersection between three polygons. The previous work of Simonson and Suto [36] also describes this proposal, but

focuses the implementation included in the Boost C++ package [2]. It is worth noting that this approach bears some resemblance to the scheme used in WVCs.

Wang et al. [39] and Feito et al. [10] both propose yet another alternative strategy: dividing polygons into geometric primitives using line sweeping in order to simplify the overlay operation – a kind of divide-and-conquer strategy. Wang et al. [39] applies trapezoidal meshes to represent polygons, while Feito et al. [10] adopts a special CSG-like representation, in which the primitives are triangles.

3 THE WVC DATA STRUCTURE

In this section we discuss a data structure named *Weighted Vertex Collection* (WVC) which will be used to solve the Map Overlay problem.

3.1 Weighted Vertex

The proposed approach represents bidimensional discrete scalar fields bounded by polygonal lines, i.e., functions that map $\mathbb{R}^2$ onto $\mathbb{Z}$ where regions that are mapped to the same value are delimited by straight line segments– Figure 2 contains an illustration of this idea. In essence, the structure is a collection of elements called *weighted vertices*. Each weighted vertex is defined by a point (x, y), a weight w and an angle θ. A weighted vertex adds w to the scalar field for points inside its area of influence – also termed *cone* – which is an infinite sector spanned by two rays intersecting at the vertex position: the first is a horizontal ray and the second makes an angle of θ with the x axis (see Figure 3.a). In order to illustrate the

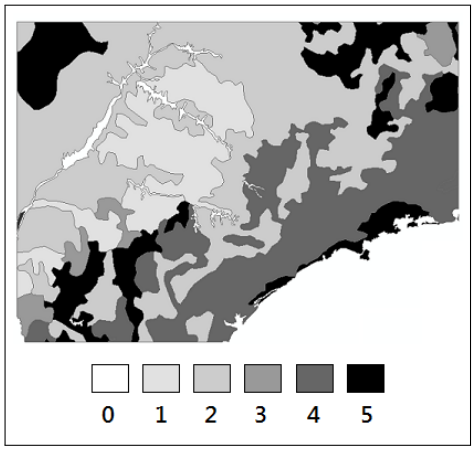

Figure 2: Map regions represented as a discrete scalar field. The points of plane has same color if they has same scalar value.

concept, let us model a scalar field called S by means of a collection of weighted vertices C. Initially, the scalar field maps all points of the domain to zero. When a weighted vertex v_1 is added to C, the region corresponding to its cone is altered by adding $w(v_1) = w$ to the scalar field, as shown in Figure 3.a. By adding other vertices it is possible to define a limited region mapped to w in S (Figure 3.b).

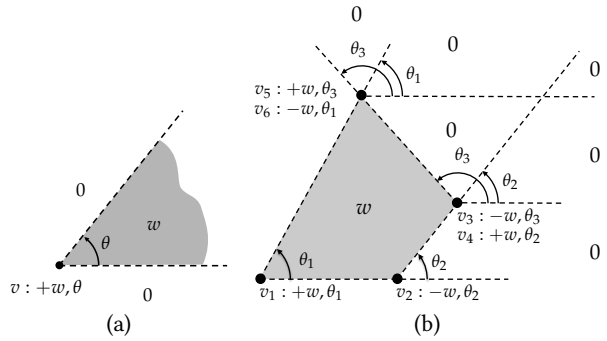

Figure 3: (a) A weighted vertex v with weight $w(v) = +w$ and angle $\theta(v) = \theta$ adds w to the scalar field within its cone (gray area). (b) five more vertices are added to the representation in order to delimit a quadrangular region mapped to w. Notice that v_3 and v_4 lie on the same point but with distinct weights and angles. The same happens with v_5 and v_6

The WVC is related to previously proposed approach for representing discrete scalar fields, namely, the *Vertex Representation* (VR) in Esperança and Samet [9]. In particular, the VC is a simplified version of the present scheme where all vertices have $\theta = 90^o$ for representing and manipulating orthogonal objects with n dimensions. The difference between the WVC and the VR lies chiefly on the former's larger modeling space with respect to the latter, whereas the VR extends easily to more than 2 dimensions.

In a WVC in *canonical form*, the following constraints are imposed: (1) the weight can not be zero; (2) two vertices v_1 and v_2 with the same position may be defined only if $\theta(v_1) \neq \theta(v_2)$. Otherwise, v_1 and v_2 are replaced by v_3 with the same angle and position, where $w(v_3) = w(v_1) + w(v_2)$ if $w(v_3) \neq 0$, and, if not, then v_1 and v_2 cancel each other and are removed from the representation.

3.2 Operations on WVCs

Let S_C denote the scalar field represented by a WVC C, i.e., $S_C(p)$ is the value of the field at point p. Also, for a weighted vertex v, let $p(v)$, $w(v)$ and $\theta(v)$ denote the position, weight and angle of v, respectively. Then we define the following operations on WVCs.

Add: the sum of two scalar fields, denoted $S_C = S_A + S_B$, is performed by placing the vertices of both collections in the same vertex collection, i.e., $C = A \cup B$. In other words, this operation puts two vertex collections $A = \{v_1, v_2, ..., v_n\}$ and $B = \{v_{n+1}, v_{n+2}, ..., v_m\}$ in same collection $C = \{v_1, v_2, ..., v_m\}$.

Scalar Transformation: computes a scalar field $S' = f(S)$, where f is a scalar function $\mathbb{R} \rightarrow \mathbb{R}$. In other words, this function computes a new scalar field where, if $S(p) = \alpha$, then $S'(p) = f(\alpha)$ for all points p of the plane. Therefore, a new vertex collection is created to represent the transformed scalar field. This is the key operation to compute Map Overlay operations.

Scalar Multiplication: the multiplication of field S by a scalar $\alpha \neq 0$, denoted αS, means keeping the v in the same position, and multiplying $w(v)$ by α, $\forall v \in C$. This is a special case of *Scalar Transformation*, but is important be highlighed here due to its role in Map Overlay operations.

In order to simplify algorithms and other further explanations, we will consider that the scalar function is such that $f(0) = 0$. Otherwise, if $f(0) = \alpha$, we just add a special vertex v to C such that $w(v) = \alpha$, $\theta(v) = 90^o$ and $p(v)$ is to the left and below of the analysis window. In other words, we have to create a vertex that adds α to the scalar field in the whole extent of the area of interest.

3.3 Properties of WVCs

Although we do not present a formal proof here, we claim that WVCs enjoy two important properties:

Boundary Property: weighted vertices of C are found only on points where the scalar field changes. This is reasonable since vertices always change the field in their neighborhood, unless they have zero weight, which is not permited for collections in canonical form.

Uniqueness of Representation: if a scalar field S can be represented by a canonical WVC C, then no other WVC can represent S. This makes sense since the scalar field in the neighborhood of each vertex $v \in C$ cannot be generated without v and C cannot be augmented by the addition of any other vertex $v' \notin C$ which would change S in the neighborhood of v'.

4 SCANNING WVCS

Since a WVC represents the changes in a scalar field, but not the field values directly, all operations that must evaluate the field are computed using a process based on the *plane sweep* [8] or, more specifically, the *line sweep* paradigm, since the problems at hand are two-dimensional. The line sweep paradigm can be viewed as a form of divide-and-conquer approach, where a two-dimensional problem is solved by breaking it up into several one-dimensional subproblems.

In order to process WVCs, a horizontal line is swept upwards along the y axis and recording the changes in the field along the way. This explains the constraint $0 < \theta < \pi$ for the vertex angles so that cones span upwards, i.e., towards the portion of the plane not yet scanned. A data structure called a "scanline" records the variation of the field along horizontal line $y = c$, for a given constant c. In the discussion that follows we will use the symbol L to refer to the state of the scanline. L records which x coordinates correspond to changes of the scalar field, i.e., the points where the scanline crosses vertex rays – Figure 4 ilustrates this concept. During a sweep, the scanline is analyzed from left to right, starting at $x = -\infty$, where the scalar field is null, adding weights associated with intersected rays until reaching $x = +\infty$. In other words, the algorithm determines the scalar value α at a point p on the scanline (i.e. $S(p) = \alpha, p \in L$) by adding up the changes generated by rays to the left of p.

Since most important algorithms on WVCs rely on line sweeping, it stands to reason to organize WVCs as lists sorted in the same order the sweeping line finds the vertices, i.e., using y as a primary key and x as a secondary key. Thus, it is not necessary to perform the initial step required by most sweep algorithms, namely, sorting the input.

In addition to L, another data structure, E, is necessary to hold so-called *stop events*. E is a priority queue that maintains stop events sorted in the same order where the sweeping line finds the vertices. Each event is associated to one of the two rays of a given vertex.

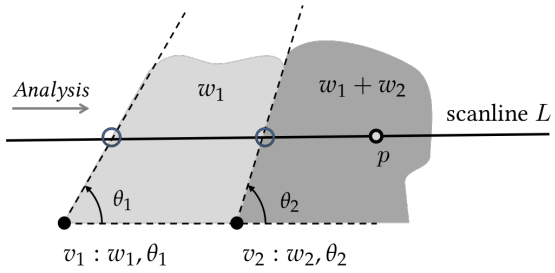

Figure 4: Each vertex changes the field to the right of the corresponding ray. Thus, one-dimensional scanline analysis must add up all rays to the left of a given point p to compute the field at that position.

Stop events occur: (a) when a vertex is swept by the scanline, and (b) when the scanline sweeps an intersection point between two rays. Events of the first type are put in E at the beginning of scan(see Algorithm 1), whereas those of the second type are discovered and added to E during the scan.

Since two or more stop events can occur at the same point, an additional key is added to the scan order rule of E and L. Namely, when two events in E have same coordinates or when two rays intersect L at the same coordinates, then the ray angle θ is used as a tie-breaker, because the ray with higher angle will cross the scan line at a lower x when the scan line moves upwards. More precisely, a scanline at $y = c$ must maintain the relative positions of rays with respect to $y \geq c$ until new events are processed.

It is important to note that the WVC properties described in Section 3.3 ensure that stop events are unique for a given vertex collection.

In the algorithm descriptions that follow, we postulate that line sweeping proceeds by repeatedly invoking a procedure called *Scan* that incorporates into the scanline all the changes related to the events that occur at the same (next) point. This loop terminates when priority queue E becomes empty ($|E| = 0$). This process is similar to what occurs in many other line sweep algorithms, and is not described in more detail here. Suffice it to say that it has the same time complexity as the sweep algorithm for finding intersections for a set of line segments [8], namely, $O(i \log k + k \log m)$ where i stands for the total number of intersections, k the number of rays and m the average number of rays crossing L.

Some operations on WVCs (e.g. *Scalar Multiplication* and *Add*) may be implemented in linear time by a simple examination of all vertices. Nonetheless, most operations on WVCs require a line sweep procedure to rebuild the scalar field and compute values at points or regions. For example, a convenient way to implement the *Draw* operation is to generate a trapezoidal decomposition of the region using line sweep, followed by a render of each trapezoid.

5 WVC ALGORITHMS

In this section, we present all relevant WVC algorithms without delving in all their intricacies. A more detailed presentation using pseudo-code is included as an appendix at the end of this paper for the interested reader.

5.1 Add

The aim of this operation is to obtain $C_3 = C_1 \cup C_2$ for two input WVCs C_1 and C_2. Since they are already sorted in scan order, this operation can be implemented by a simple merge(see Algorithm 2). Collection C_3 is output as a list sorted in scan order and respecting the canonical representation restrictions as described above.

5.2 Scalar Transformation

Given a collection C representing the field S, the goal is to compute a collection C' representing a new field $S' = f(S)$. Initially, C' is an empty collection and a sweep is executed in tandem on C and C' maintaining respectively L and L'. Notice, however, that priority queue E holds only events pertaining to the sweep of C. This is justified, since no event may occur during the sweep of C' which has not occurred also in the sweep of C.

Then, when events $e \in E$ are processed to update L, a balanced line algorithm in *scan order* is performed between L and L' to check whether $S'(p(e)) = f(S(p(e)))$, for all event points $p(e)$.See Algorithm 3.

If this condition is not met at a given event position $p(e)$, a new weighted vertex v' is added to C' to alter the field accordingly, where $w(v') = S'(p(e)) - f(S(p(e)))$ and $\theta(v') = \theta(r(e))$, assuming that $r(e)$ is the ray associated to the stop event and $\theta(r(e))$ is the ray angle.

5.3 Scalar Multiplication

The *Scalar Multiplication* is a special case of the Scalar Transformation, since a new collection C' is created with each input vertex having its weight multiplied by α to represent a new scalar field $S_t = f(S) = \alpha S$. That is, $C' = \{\alpha.v | v \in C\}$ – see Algorithm 6.

5.4 Convert To

The conversion from a polygon set represented by vertex circulations to a WVC can be performed in two steps. First, a collection C_i is created for each polygon P_i of the map, where each edge generates two weighted vertices at its ends– see Algorithm 5. Then, the Add procedure is used to generate the resulting collection $C = \bigcup_{i=1}^{n} C_i$. The weight values may be, for instance, an incremental number given to each class of the map or to each feature. An alternative algorithm consists of putting all weighted vertices generated by all vertex circulations into a single WVC, and then sorting it in scan order.

Conversion from a polygon to a WVC assumes that the input circulation is in counterclockwise order so as to map the polygon interior to $+w$. Similarly, circulations representing holes are assumed to be in clockwise order, as established by the ISO 19125-1:2004 standard [11].

It is important to note that if a vertex circulation contains a self-intersection then an inversion in direction will occur. This way, necessarily some vertices will be oriented in counterclockwise order and thus mapped to w, but others will be clockwise oriented and thus be mapped to $-w$. This issue can be addressed by performing a simple Scalar Transformation using function $f(\alpha) = |\alpha|$.

5.5 Convert From

The conversion of a WVC into vertex circulations is performed by a line sweep procedure. The scan searches boundaries of regions with the same scalar field value, and is similar to algorithm *Draw*. These boundaries are given by the cone rays generated by the weighted vertices.

5.6 Draw

A convenient way to paint a polygonal region represented by a WVC is to generate a trapezoidal decomposition of the region using line sweep [8]. Algorithms that realize such a decomposition are straightforward and are described, for instance, in Wang et al.'s [39] and Vatti's [38] proposals, and are not further explained here.

5.7 Algorithm Complexities

Some of these algorithms can be performed in $O(n)$ by simply visiting all n vertices, namely: *Add* and *Scalar Multiplication*. In fact *Convert To* is also linear, provided that the input does not contain self-intersections.

However, the scan procedure used in all other algorithms – namely, *Scalar Transformation*, *Draw*, and *Convert From* – requires at least $O((i + k) \log m)$ time [8], where i is the number of ray intersections, k is the collection size and m is the average number of rays crossing the scanline. Given $n = i + k$ and $i \ll k$, we can rewrite this bound as $O(n \log m)$. We note that, although in the worst case $i \approx k^2$, real map data will unlikely exhibit this behavior. For instance, the aforementioned related works which also employ sweeping, state their complexities considering $i \ll k$.

Additionally, we point out that in our implementation the algorithm used to analyze the scanline Algorithm 4, in the form it is presented in the Appendix,computes the scalar value in $O(m)$ time. Therefore, the combined time complexity of these algorithms is $O(m\, n \log m)$. Considering the worst case when $m \approx n$, the complexity can be written simply as $O(n^2 \log n)$. However, it is noted that m is related to the number of y-monotone curves of the polygon set, since the scanline is swept upwards. Again, we claim that real data unlikely will present $m \approx n$. In any case, an auxiliary data structure such as a *skip list* [29, 31] can be used in an implementation to ensure $O(\log m)$ for Algorithm 4, thus resulting in a final time complexity $O(n\, 2 \log m)$ – or simply $O(n \log m)$.

As for the space complexity, a WVC clearly requires $O(n)$ space, where n is the number of vertices in an equivalent polygon-based representation. In practice, the space required for each vertex must accomodate more than just its position. Also, a WVC may have more than one weighted vertex at the same position, but we note that the same is true also for polygon-based respresentations. The space complexity of all sweeping line algorithms is bounded by the space required for (1) the input and output WVCs, (2) the event queue E and (3) the scanline L. As reported by [8] for similar algorithms, these are all $O(n)$.

6 MAP OVERLAY USING WVCS

Typical Map Overlay operations such as union, intersection and difference can be performed with the aid of a *Scalar Multiplication*, an *Add*, and a *Scalar Transformation*. To illustrate the concept with a simple example, consider two collections C_1 and C_2 representing the

scalar fields S_1 and S_2 respectively. Let us assume that the non-zero mapped regions of each of these two fields can be described by two polygons, say, P_1 and P_2, respectively as shown in Figure 5. Thus, computing the *Add* of both collections, i.e. $C_3 = C_1 \cup C_2$, followed by a *Scalar Transformation* of C_3 with an appropriate function it is possible to obtain all set-theoretic operations on polygons. For instance, $P_1 \cup P_2$, $P_1 \cap P_2$ and $P_1 - P_2$ are obtained respectively by applying function $f_\cup$, $f_\cap$ and f_-.

$$f_\cup(x) = \begin{cases} 1 & \text{if } (x > 0) \\ 0 & \text{otherwise} \end{cases} \qquad f_\cap(x) = \begin{cases} 1 & \text{if } (x > 0) \\ 0 & \text{otherwise} \end{cases}$$

$$f_-(x) = \begin{cases} 1 & \text{if } (x = 1) \\ 0 & \text{otherwise} \end{cases}$$

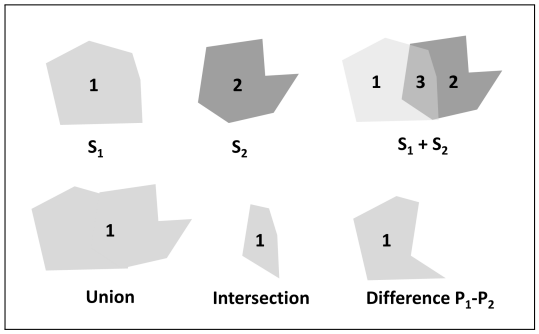

Figure 5: Set-theoretic operations expressed as Scalar Transformations.

Let us describe a more general example. Consider two polygonal maps M_1 and M_2 represented by scalar fields S_1 and S_2, which, in turn, are encoded by WVCs C_1 and C_2, respectively. Let w_1 and w_2 denote the scalar value of S_1 and S_2 at a given point. Further, let us assume that maps M_1 and M_2 consist of a subdivision of the plane into n_1 and n_2 map classes, i.e., S_1 is a scalar field $\mathbb{R}^2 \rightarrow \{1...n_1\}$ and similarly for S_2.

In order to identify all desired overlays in an arbitrary operation, it is necessary to compute a field S_3 that represents the *cartesian product* of the input maps, i.e., it encodes as scalar values all possible $n_1 \times n_2$ combinations. To this end, we can define and compute $S_3 = S_1 + \alpha S_2$, for an integer $\alpha > n_1$. This way, S_3 holds all information of both S_1 and S_2 since a point mapped to w in S_3 is mapped to $w \mod \alpha$ in S_1 and to $w \div \alpha$ in S_2. For a numerical example, let $n_1 = 12$, $n_2 = 9$ and $\alpha = 100$; then a region mapped to 211 in S_3 corresponds to the intersection of a region mapped to 11 in S_1 and to 2 in S_2. Similarly, a scalar value 800 identifies regions where $w_2 = 8$ and $w_1 = 0$, that is, a region of M_2 does not intersect in any region of M_1.

Finally, a Scalar Transformation can be applied to remove the regions that represent undesired overlaps by mapping them to zero, and mapping the meaningful overlaps to the desired map classes.

7 IMPLEMENTATION, TESTS AND RESULTS

A straightforward implementation of the *Weighted Vertex Collection* data structure and its algorithms was written in the *C++* language and compiled using the Gnu Compiler Collection 4.2.6 (g++). It is capable of computing the basic operations, properties and the *line sweep* process described above, namely, the *Scan*, *Add*, *Convert*, *Draw* and *Scalar Transformation*, whereas the ShapeLib 1.2.9 [34] library provided support for reading and writing cartographic data in the ubiquitous *Shapefile* format.

Our implementation was then used to demonstrate that (1) WVCs can be applied to the processing of *Map Overlay* operations; (2) the processing time of WVC-based operations can be competitive with algorithms implemented in mainstream Spatial Database Management Systems and Geographic Information Systems; and (3) to assess the differences between the WVCs and typical approaches to compute Map Overlays.

In this implementation the weighted vertices were adjusted to a grid with 2^n positions for ensuring robust floating-point operations, where the n is large enough to preserve the positional accuracy of spatial data. The weighted vertex angle θ is replaced by a direction vector to facilitate geometric operations such as ray intersect and ray orientation.

It should also be mentioned that this implementation uses a simple array to implement the scanline data structure L and not a more specialized structure such as a skip list to ensure the optimal time complexity $O(n \log m)$, as discussed in Section 5.7.

7.1 Experiments

In order to compare processing times, we chose public domain implementations for computing operations on polygons, namely: GEOS, CGAL, Clipper, and Boost. See Table 2.

GEOS (Geometry Engine - Open Source) [14] is a project of Open Source Geospatial Foundation [16] and applied in several of mainstream GIS solutions such as in the Spatial Database Management System called PostgreSQL/PostGIS [30], and in GIS applications such as QGIS [32], gvSIG [19] and MapServer [15] – see a list with free and proprietary packages that use this library in the GEOS homepage [14]. Moreover, the GEOS is a C++ port of the Java Topology Suite (JTS) [20] written in Java – several solutions such as GeoServer [13] use JTS in their geometry operations.

Unfortunately, as far as we know, the documentations for GEOS and JTS do not provide details about the algorithm used in polygon overlay operations. What can be said is that they do not support self-intersection and other situations described in GEOS homepage [14] – demanding a previous geometry cleaning operation.

The Computational Geometry Algorithms Library (CGAL) brings together geometric algorithms in the form of a C++ library. According to its home page [6], this library offers data structures and algorithms covering several topics of Computer Geometry. CGAL contains a package called *2D Regularized Boolean Set-Operations* to compute polygon overlay, where polygons can have holes but not self-intersections. As with GEOS and JTS, the CGAL documentation does not provide details about its overlay algorithm. However, inspecting its source code and the available documentation, we can conclude that CGAL models polygons as a set of x-monotone curves using double linked lists, then applies the plane sweep paradigm to find intersections. Lastly, the algorithm classifies the edges of a polygon as *is contained* or *is not contained* in the other polygon

in order to build the final geometry – this approach is very similar to algorithm strategies presented in Section 2.

In its turn, Clipper [21] is an open source library based on Vatti's algorithm [38] for clipping and offsetting lines or polygons and has versions written in Delphi, C++ and C#. Johnson [21] and Mondron [28] present several results where Clipper always exhibits better performance than other well-known implementations such as CGAL and GEOS. However, these results also show that Boost and Clipper have similar performances.

Boost [2] is a package that provides free peer-reviewed portable C++ source libraries. In particular, the *Boost.Polygon* library [3] supports polygon overlays, and is an implementation of the proposal presented in Simonson and Suto [36] and Simonson [37] as described in Section 2.

Library	Version	Intersection Function
Boost	1.54.0	boost::polygon::property_merge
Clipper	6.2.1	ClipperLib::Clipper::Execute
CGAL	4.9	CGAL::intersection
GEOS	3.6	geos::operation::overlay::OverlayOp

Table 2: Libraries and its functions to perform comparison with the WVC for the first experiment.

The datasets used in the experiments cover four separate areas of Brazil. Datasets RJ1 and RJ2 map administrative subdivisions and forest cover for the region of Tijuca in the city of Rio de Janeiro (Figure 6). Datasets SP1 and SP2 contain a soil fertility classification and the municipalities of the state of São Paulo in Brazil (Figure 7). Datasets TS1 and TS2 contain classifications for land use and vegetation cover for the Teresópolis municipality. Finally, datasets BR1 and BR2 contain a soil fertility classification and the political subdivision of Brazil in municipalities. The experiments using polygon collections required a pre-processing step to obtain polygon sets from the original Shapefiles, while the processing with WVCs required an additional application of Convert To (Section 5.4). Relevant statistics for these datasets are shown in Table 3.

Dataset	Polygons	Vertices	Convert To (s)
RJ1	3	830	0.01
RJ2	12	3,830	0.03
SP1	54	4,758	0.04
SP2	678	83,778	0.31
TS1	1,395	134,579	0.45
TS2	1,473	138,932	0.47
BR1	409	49,068	0.23
BR2	5,784	1,195,768	3.91

Table 3: Statistics for the polygon sets used in the experiments. Column "Convert To" lists the additional preprocessing time to convert polygon sets to equivalent WVCs.

The first experiment consists of generating the cartesian product the dataset pairs defined for each area. This is realized with WVCs using the approach outlined in Section 6. In other words, we compute operations: Scalar Multiplication, Add, and Scalar Transformation.

Datasets	Unfiltered	GEOS	Clipper	Boost	WVC
RJ1×RJ2	3	0.067	0.01	0.0047	0.039
SP1×SP2	2,308	3.88	3.5	0.15	0.21
TS1×TS2	13,649	59.8	5.59	1.78	1.8
BR1×BR2	20,686	*	58.7	2.23	4.7

Table 4: Times in seconds (mean of ten runs) to compute all intersections (cartesian product) for the first experiment, where * denotes unsuccessful processing due to an unspecified error. Column Unfiltered records the number of polygon pairs with intersecting bounding boxes.

An equivalent procedure was also conducted for the original polygonal maps using the overlay functions listed in Table 2. These are called within a nested loop to compute the intersection of polygons pairs. In order to compare the processing times in a more fair way, a simple filtering step was implemented, where pairs with non-overlapping bounding boxes are disregarded. The number of non-filtered pairs is shown in column *Unfiltered* of Table 4. In other words, the time spent iterating the nested loop is not counted and neither is the time spent for testing bounding boxes. It could be argued that this query execution plan might not be the most performant, especially if there are spatial indices available for one or more of the datasets, but we find that the use of filtering with bounding boxes is a reasonable compromise.

The results of this first experiment are shown in Table 4. Since the tests using CGAL yielded very high processing times when compared to the other libraries, their results were disregarded. This low performance exhibited by CGAL was also observed in Johnson [21].

It should be noted that this first batch of tests tries to reproduce a typical query execution plan that uses the *filter-and-refine* strategy for computing operations such as union, merge or clip. However, there are other operations, e.g., Map Algebra operations (see [5] for an in-depth discussion), that require a more complex query plan.

With this in mind, we conducted a second experiment that computes a specialized map overlay using datasets TS1, which contains 9 different vegetation classes, and TS2, which contains 22 land use classes. The overlay operation for the second experiment consists of grouping the 9 classes of TS1 into 3 groups of 3, the 22 classes of TS2 into 2 groups of 11 and then taking the cartesian product, yielding 6 classes as shown in Table 5. The results of both steps are shown in Figure 10.

The processing of this overlay with WVCs follows the same procedure outlined for the first experiment, except that the Scalar Transformation uses a function that implements the mapping shown in Table 5. On the other hand, when using polygons, several query execution plans could be used. Again disregarding the possibility of using specialized spatial indices, we opted for a plan consisting of (1) executing a "dissolve" operation on TS1 and TS2 so as to produce two derived datasets TS1B and TS2B with 3 and 2 classes, respectively, followed by (2) a cartesian product of TS1B and TS2B.

		TS2	
		Classes 1-11	Classes 12-22
TS1	Classes 1-3	Class 1	Class 2
	Classes 4-6	Class 3	Class 4
	Classes 7-9	Class 5	Class 6

Table 5: Result region classes for the second experiment.

Step (1) used specialized functions particular to each tested library, while step (2) was computed in a similar fashion to what was done for the first experiment. Incidentally, an alternative plan could be devised where steps (1) and (2) are swapped, i.e., the dissolve would be applied to the result of the cartesian product containing up to $9 \times 22 = 198$ classes, but this would certainly be more costly.

Step (1) resulted in 384 polygons in TS1B and 741 polygons in TS2B, while step (2) yielded a final result containing 1,011 polygons. The running times for all tested implementations is shown in Table 6.

Clipper			Boost			WVC
Step 1	Step 2	Total	Step 1	Step 2	Total	
3.72	4.4	8.12	3.69	0.98	4.67	1.48

Table 6: Times in seconds (mean of ten runs) to compute the map overlay of the second experiment.

7.2 Discussion

A cursory examination of the results of the two experiments indicates that Boost is more performant for polygon intersection operations than the other two tested libraries. In the first experiment our own approach performs better than Clipper and GEOS, but worse than Boost. For the first and the last test, in particular, our implementation ran about twice and eight times slower than Boost, respectively. In the first case, this can be attributed to the fact that only 3 out of 36 pairs actually intersect. The last test was the most expensive of the lot, and for these datasets the typical complexity of the scanline is fairly large, about 90+ rays on average, which impacts the performance of our naïve implementation that analyzes the scanline in linear time.

A clear advantage of using polygons is the possibility of using spatial filters and indexing data structures. We were careful not to misrepresent this advantage by using a simple bounding box filter and expunging box intersection times from the total times presented in Tables 4 and 6. Thus, even if in practice a spatial index such as an R-tree were used in our experiments, no additional gain would be measured.

At any rate, it seems clear that overlays where spatial filtering plays an important role will benefit polygon-based approaches, since much of the geometry might never be actually processed, whereas WVCs always process all vertices in the input datasets. For a more complex overlay, however, this advantage becomes less important, as is revealed in the second experiment.

A more detailed examination of the second experiment is in order. Firstly, the implementation of step (1) required a completely different approach for each of the three[1] polygon libraries, which provide different levels of support for the "dissolve" operation. GEOS follows the ISO 19125-1:2004 standard [11] and thus provides explicit support, although it failed for these particular datasets. Clipper only contains support for computing polygon unions, and thus each class in TS1B and TS2B had to be produced by successive applications of this operation. For instance, to obtain the polygons that represent a grouping of classes 1,2 and 3 of TS1, we start by creating a layer containing all polygons in class 1, 2 and 3. The same process is repeated for classes 3,4,5 and so forth. Finally, the Boost library supports all polygon overlay operations with the single function shown in Table 2, that can compute the required "dissolve" through a careful use of tags associated with the input polygons.

In contrast, the complete overlay operation was computed with WVCs in a relatively straightforward way. In fact, this kind of processing resembles what is typically done in raster-based GIS's. This simplified way of expressing the overlay is reflected in its superior performance relative to Clipper and Boost, namely, about three times as fast than the best competitor. Also worth noting is the fact that the processing of step (2) could not be benefitted by the availability of spatial indices, since its inputs had to be computed afresh. Furthermore, it is not hard to imagine other overlay operations which could widen this gap in performance. For instance, we could consider a more convoluted class grouping than that shown in Table 5 or even an overlay of more than three input maps.

8 CONCLUSIONS

This paper proposes the Weighted Vertex Collection (WVC) data structure as a representation for region maps. We demonstrated how several important operations on such maps can be efficiently computed with WVCs. The WVC revolves around the central concept of a "weighted vertex", a simple structure containing 3 fields. This contrasts with usual polygon-based representations which require more coarse-grained structures such as multipolygons, circulations and holes [11]. Moreover, WVCs model region maps as scalar fields bounded by polygonal lines, which arguably offers a more adequate abstraction than sets of labeled polygons, in the sense that the former guarantees that each point of the plane naturally maps to a single value, whereas the latter must enforce this property explicitly. This kind of modeling is reminiscent of raster representations, frequently used in GIS's for computing certain operations such as those defined in Map Algebra [5]. For some of these, WVCs might offer an interesting alternative since they are vector-based and thus not dependent on raster resolution.

A relatively small but significant set of experiments was conducted where sample map overlay operations were computed with a proof-of-concept WVC implementation. These operations were also computed using popular polygon processing libraries by the way of comparison. Results indicate that WVCs are eminently competitive and may even outperform GIS software in current use. Of course, this can only be conclusively ascertained after a more thorough empirical assessment.

[1] As mentioned earlier, we discarded using CGAL in our experiments.

(a) (b) (c)

Figure 6: (a) Dataset RJ1 that maps the neighborhoods of the city of Rio de Janeiro belonging to the administrative region of Tijuca; (b) dataset RJ2 with the forest cover of this region; and (c) the intersection of (a) and (b). These datasets were produced by Rio de Janeiro's city hall.

(a) (b) (c)

Figure 7: (a) Dataset SP1 with a soil fertility classification of the State of São Paulo; (b) dataset SP2 with that state's municipalities; and (c) the intersection result between (a) and (b). These datasets were produced by the Brazilian Institute of Geography and Statistics (acronym IBGE, in Portuguese). These are subsets of BR1 and BR2, respectively.

(a) (b) (c)

Figure 8: (a) Dataset TS1 with a land use classification and (b) dataset TS2 with a vegetation cover classification, both for the Municipality of Teresópolis (Rio de Janeiro/Brazil). (c) shows the intersection result between (a) and (b). These datasets were producted by Geotecnology Center of Rio de Janeiro State University.

WVCs suffer from several limitations, however. Firstly, no spatial indexing approach can be immediately applied to them, other than clipping maps to regions of interest. Other important operations on region maps have not yet been investigated, such the computation of buffer regions or overlays with point or network datasets. All of these issues are in our plans for future work.

REFERENCES

[1] Ludger Becker, André Giesen, Klaus H. Hinrichs, and Jan Vahrenhold. 1999. Algorithms for Performing Polygonal Map Overlay and Spatial Join on Massive Data Sets. In *Advances in Spatial Databases-6th International Symposium, SSD'99, Hong Kong*. Springer-Verlag.

[2] Boost n. d.. Boost C++ Libraries. (n. d.). Retrieved Jun 22, 2017 from http://www.boost.org/

(a) (b) (c)

Figure 9: (a) Dataset BR1 with a soil fertility classification and (b) municipalities of Brazil. (c) shows the intersection between (a) and (b). This dataset was produced by the Brazilian Institute of Geography and Statistics (acronym IBGE, in portuguese).

(a) (b) (c)

Figure 10: The results of the first step in the experiment 2, where (a) and (b) are respectively TS1B and TS2B, generated by dissolve operations as described in Table 5. (c) is the cartesian product of (a) and (b), that is, the final result.

[3] Boost Polygon 2010. Boost Polygon Library: Main Page. (2010). Retrieved Jun 22, 2017 from http://www.boost.org/doc/libs/1_63_0/libs/polygon/doc/index.htm

[4] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1994. Multi-Step Processing of Spatial Joins. In *Proceedings of ACM SIGMOD International Conference on Management of Data*.

[5] H Thomas Bruns and Max J Egenhofer. 1997. User interfaces for map algebra. *URISA-WASHINGTON DC*- 9 (1997), 44–55.

[6] CGAL n. d.. The Computational Geometry Algorithms Library. (n. d.). Retrieved Jun 22, 2017 from http://www.cgal.org/

[7] Joep Crompvoets, Abbas Rajabifard, Bastiaan van Loenen, and Tatiana Delgado Fernandez. 2008. *A Multi-View Framework to Assess Spatial Data Infrastructures*. Printed by Digital Print Centre, The University of Melbourne, Australia.

[8] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 2008. *Computational geometry: algorithms and applications*. Springer.

[9] Claudio Esperança and Hanan Samet. 1998. Vertex representations and their applications in computer graphics. *The Visual Computer* (1998).

[10] FR Feito, M Rivero, and A Rueda. 1999. Boolean representation of general planar polygons. *sign* 1 (1999), 2.

[11] International Organization for Standardization. 2013. ISO 19125-1:2004 - Geographic information Simple Feature Access - Part 1: Common Architecture. (2013). Retrieved Jun 22, 2017 from https://www.iso.org/standard/40114.html

[12] Erich L Foster and James R Overfelt. 2012. Clipping of Arbitrary Polygons with Degeneracies. *CoRR* abs/1211.3376 (2012).

[13] Open Source Geospatial Foundation. 2014. GeoServer is an open source server for sharing geospatial data. (2014). Retrieved Jun 22, 2017 from http://geoserver.org/

[14] Open Source Geospatial Foundation. 2017. GEOS – Geometry Engine, Open Source. (2017). Retrieved Jun 22, 2017 from https://trac.osgeo.org/geos/

[15] Open Source Geospatial Foundation. 2017. Welcome to MapServer - MapServer 7.06 documentation. (2017). Retrieved Jun 22, 2017 from http://mapserver.org/

[16] Open Source Geospatial Foundation. n. d.. OSGeo.org – Your Open Source Compass. (n. d.). Retrieved Jun 22, 2017 from http://www.osgeo.org/

[17] Günther Greiner and Kai Hormann. 1998. Efficient Clipping of Arbitrary Polygons. *ACM Trans. Graph.* 17, 2 (April 1998), 71–83. https://doi.org/10.1145/274363.274364

[18] Ralf Hartmut Güting. 1994. An Introduction to Spatial Database Systems. *VLDB J.* 3, 4 (1994).

[19] gvSIG Association. n. d.. Home - Portal gvSIG. (n. d.). Retrieved Jun 22, 2017 from http://www.gvsig.com/en

[20] Java Topology Suite 2016. The JTS Topology Suite is a Java library for creating and manipulating vector geometry. (2016). Retrieved Jun 22, 2017 from https://github.com/locationtech/jts

[21] Angus Johnson. 2014. Clipper – an open source freeware polygon clipping library. (2014). Retrieved Jun 22, 2017 from http://www.angusj.com/delphi/clipper.php

[22] Dae Hyun Kim and Myoung-Jun Kim. 2006. An extension of polygon clipping to resolve degenerate cases. *Computer-Aided Design and Applications* 3, 1-4 (2006), 447–456.
[23] H Kriegel, T Brinkhoof, and R. Schneider. 1993. Efficient Spatial Query Processing in Geographic Database System. *Data Enginnering Bulletin* 16 (1993), 10 – 15.
[24] Hans-Peter Kriegel, Thomas Brinkhoff, and Ralf Schneider. 1991. An Efficient Map Overlay Algorithm Based on Spatial Access Methods and Computational Geometry. In *Proceedings of the International Workshop on DBMS's for Geographic Applications*. Springer Verlag.
[25] Yong Kui Liu, Xiao Qiang Wang, Shu Zhe Bao, Matej Gomboši, and Borut alik. 2007. An Algorithm for Polygon Clipping, and for Determining Polygon Intersections and Unions. *Comput. Geosci.* 33, 5 (May 2007), 589–598. https://doi.org/10.1016/j.cageo.2006.08.008
[26] Avraham Margalit and Gary D Knott. 1989. An algorithm for computing the union, intersection or difference of two polygons. *Computers & Graphics* 13, 2 (1989), 167–183. https://doi.org/10.1016/0097-8493(89)90059-9
[27] Francisco Martínez, Antonio Jesús Rueda, and Francisco Ramón Feito. 2009. A New Algorithm for Computing Boolean Operations on Polygons. *Comput. Geosci.* 35, 6 (June 2009), 1177–1185. https://doi.org/10.1016/j.cageo.2008.08.009
[28] Rogue Modron. 2013. Polygon Clipping: a Wrapper, a Benchmark. (2013). Retrieved Jun 22, 2017 from http://rogue-modron.blogspot.com.br/2011/04/polygon-clipping-wrapper-benchmark.html
[29] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. 1992. Deterministic Skip Lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '92)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 367–375. http://dl.acm.org/citation.cfm?id=139404.139478
[30] PostGIS 2017. PostGIS – Spatial and Geographic objects for PostgreSQL. (2017). Retrieved Jun 22, 2017 from http://www.postgis.net/
[31] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.
[32] QGIS 2017. Welcome to the QGIS project. (2017). Retrieved Jun 22, 2017 from http://www.qgis.org/en/site/
[33] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann.
[34] ShapeLib n. d.. Shapefile C Library. (n. d.). Retrieved Jun 22, 2017 from http://shapelib.maptools.org/
[35] Shashi Shekhar, Viswanath Gunturi, Michael R. Evans, and KwangSoo Yang. 2012. Spatial big-data challenges intersecting mobility and cloud computing. In *Proceedings of the Eleventh ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE '12)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2258056.2258058
[36] Lucanus Simonson and Gyuszi Suto. 2009. Geometry Template Library for STL–like 2D Operations. *Colorado: GTL Boostcon* (2009).
[37] Lucanus J. Simonson. 2010. Industrial strength polygon clipping: A novel algorithm with applications in VLSI CAD. *Computer-Aided Design* 42, 12 (2010), 1189 – 1196. https://doi.org/10.1016/j.cad.2010.06.008
[38] Bala R. Vatti. 1992. A Generic Solution to Polygon Clipping. *Commun. ACM* 35, 7 (July 1992), 56–63. https://doi.org/10.1145/129902.129906
[39] Jiechen Wang, Can Cui, and Jay Gao. 2012. An Efficient Algorithm for Clipping Operation Based on Trapezoidal Meshes and Sweep-line Technique. *Adv. Eng. Softw.* 47, 1 (May 2012), 72–79. https://doi.org/10.1016/j.advengsoft.2011.12.003
[40] Hongjun Zhu, Jianwen Su, and Oscar H. Ibarra. 2000. Toward Spatial Joins for Polygons. In *Proceedings of the 12th International Conference on Scientific and Statistical Database Management (SSDBM '00)*. IEEE Computer Society, Washington, DC, USA.

A APPENDIX - ALGORITHM PSEUDOCODES

Algorithm 1 *prepareEventList* - creates the initial priority queue to Scan process

Input: C - a WVC collection (already in scan order)
Output: E - returns the initial event list

```
E ← new EventList
for each v ∈ C do
    r1 ← getRayWithHigherAngle(v)
    r2 ← getRayWithLowerAngle(v)
    e1 ← new InsertEvent(r1,+w(v))
    e2 ← new InsertEvent(r2,−w(v))
    push(E,e1)
    push(E,e2)
end for
return E
```

Algorithm 2 *Add* – adds two scalar fields represented by two WVC

Input: C_1, C_2
Output: New collection C_3

```
i ← 0
j ← 0
C3 ← new WVC
while i < |C1| ∧ j < |C2| do          ▷ |C| returns the size of C
    if C1[i] < C2[j] then             ▷ Comparing by scan order.
        push(C3,C1[i])                ▷ C[i] gets (i+1)-est vertex of C
        i ← i + 1
    else
        push(C3,C2[j])
        j ← j + 1
    end if
end while
while i < |C1| do
    push(C3,C1[i])
    i ← i + 1
end while
while j < |C2| do
    push(C3,C2[j])
    j ← j + 1
end while
return C3
```

Algorithm 3 *transform* – apply a scalar transformation, generating a new collection

Input: C, f – a WVC and a scalar function
Output: C' – transformed WVC

```
C' ← new WVC
L' ← new ScanLine
L ← new ScanLine
E ← prepareEventList(C)                          ▷ Algorithm 1
while |E| > 0 do                                 ▷ Until end of scan
    p_curr ← scan(L,E)                           ▷ Scanning next point
    for each r ⊗ p_curr do                       ▷ For each ray crossing p_curr
        s ← sumLeftChangesOfRay(L,r)             ▷ Alg. 4
        s' ← sumLeftChangesOfRay(L',r)
        if f(s) ≠ s' then                        ▷ If true, force the constraint
            w' ← f(s) − s'
            v' ← new WeightedVertex(p_curr,θ(r),w')
            push(C',v')
            insertRays(L',v')                    ▷ Update L' with new v'
        end if
    end for
end while
return C'                                        ▷ Already in scan order
```

Algorithm 4 *sumLeftChangesOfRay* – sums the left changes of r on scanline using the scan order.

Input: L, r_1 – a scanline and a ray
Output: s – scalar value at r_1

```
s ← 0
p ← r ⊗ L
for each r2 ∈ L do                               ▷ For each ray that is crossing L
    q ← r2 ⊗ L                                   ▷ Computes the intersection point
    if x(p) < x(q) ∨ (x(p) = x(q) ∧ θ(p) > θ(q)) then
        return s                                 ▷ No more rays on the left
    end if
    s ← s + w(r2)
end for
return s
```

Algorithm 5 *convertVCtoWVC* – convert one vertex circulation to WVC

Input: VC, w – the vertex circulation and the weight of region
Output: C – converted WVC

```
C ← new WVC
for each s ∈ VC do                               ▷ For each segment in VC
    p, q ← getEnds(s)                            ▷ Get the ends of s
    θ ← slope(s)                                 ▷ Compute the slope of s
    if x(p) = x(q) then
        next s                                   ▷ Vertical segment, nothing to do.
    end if
    if x(p) < x(q) then
        if y(p) ≤ y(q) then
            v1 ← new WeightedVertex(p,θ,w)
            v2 ← new WeightedVertex(q,θ,−w)
        else
            v1 ← new WeightedVertex(p,θ,−w)
            v2 ← new WeightedVertex(q,θ,w)
        end if
    else if x(p) > x(q) then
        if y(p) ≤ y(q) then
            v1 ← new WeightedVertex(p,θ,−w)
            v2 ← new WeightedVertex(q,θ,w)
        else
            v1 ← new WeightedVertex(p,θ,w)
            v2 ← new WeightedVertex(q,θ,−w)
        end if
    end if
    push(C,v1)
    push(C,v2)
end for
sort(C)                                          ▷ Sorting in scan order
return C
```

Algorithm 6 *multiplication* – calculates the scalar multiplication by a scalar value α

Input: C, α – a collection and a scalar value
Output: C' – a list of returned values

```
C' ← new WVC
for each v ∈ C do
    p' ← p(v)                                    ▷ Get position of v
    w' ← α.w(v)
    θ' ← θ(v)
    v' ← new WeightedVertex(p',θ',w')
    push(C',v')
end for
return C'
```