REPRESENTING ORTHOGONAL MULTIDIMENSIONAL OBJECTS BY VERTEX LISTS [1]

CLAUDIO ESPERANÇA
Computer Science Department, University of Maryland
College Park, Maryland 20742

and

HANAN SAMET
Computer Science Department and Center for Automation Research and
Institute for Advanced Computer Studies, University of Maryland
College Park, Maryland 20742

Abstract

A method is presented for representing multidimensional objects with orthogonal faces (i.e., collections of hyperrectangles or n-dimensional rasters) compactly by a vertex list. This is a list of points with associated weights (vertices). Algorithms for converting between rasters and vertex lists as well as performing set-theoretic operations on these structures are described and shown to execute in $O(N \cdot d)$ time, where d is the dimension and N is the number of vertices in the representation. Other applications of vertex lists are also discussed.

1 Introduction

The shape of objects can be represented by the interiors of the objects or by their boundaries. The most common interior-based representation is one that decomposes the object into a collection of cells all of whose sides are of unit length (termed *pixels* or *voxels* in two and three dimensions respectively). The elements of the this collection (i.e., the cells) are labeled with values or attributes and are frequently aggregated into subcollections of similarly-valued cells. The decomposition into a collection of uniform cells makes it very easy to calculate properties such as mass as well as determining the value associated with any point of the space covered by the cell.

On the other hand, boundary-based representations are more amenable to the calculation of properties pertaining to shape (e.g., perimeter, extent, etc.). In this case, we simply record the different boundary elements. We frequently make use of a similar technique to that of decomposing an object into a collection of unit-sized cells. The difference is that now we record their boundaries. Such methods are relatively

[1]The support of the National Science Foundation under Grant IRI-9017393 and of Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPQ) are gratefully acknowledged.

easy in two dimensions where the boundaries are lines. It is somewhat more difficult in three dimensions (and higher).

Both interior- and boundary-based representations can be made more compact by just recording the *change* in the values rather than the values themselves. For interior-based representations this technique is known as runlength encoding [3]. It aggregates identically valued cells into one-dimensional rectangles for which only the value and the length of the rectangle are recorded. The location of the cell can be computed by referring to the location of the starting position. This method is also applicable to data of higher dimensions. For boundary-based representations, an analogous effect can be achieved by use of chain codes [1]. This is a technique for representing the boundary of an object by a sequence of directional codes corresponding to unit steps in the four or eight principal directions. Unfortunately, it is difficult to extend this idea to dimensions higher than two as there is no natural order for traversing the boundaries of such objects.

Recently, Schechtman [7] introduced a new technique for storing VLSI masks known as the *vertex algebra*. The vertex algebra is also based on recording changes in values rather than the values themselves. It is a combination of an interior and a boundary representation in the sense that both the interiors and boundaries of the regions are represented implicitly through the aid of a single vertex which is the tip of a cone. The vertex algebra was originally presented primarily as an alternative model for two-dimensional data represented as lists of rectangles. Algorithms have been devised for a number of vertex algebra operations in two dimensions [7].

In this paper, we show how the vertex algebra can be extended to represent orthogonal objects in arbitrary dimensions, and present a number of algorithms for operations that are useful in computer vision and pattern recognition applications. The contribution of our work is the generality of the solutions that we obtain through the use of recursion to construct the representation in higher dimensions, as well as performing the operations on it. This is largely a result of the switch of the primitive unit to being a one-dimensional vertex list rather than a two-dimensional vertex list. This same principle was also used by us in developing a raster to vertex list conversion algorithm, reported here, which works in arbitrary dimensions. Of course, we could have also treated the individual cells as unit hyperrectangles and applied the methods of Schechtman. However, this is cumbersome and defeats the notion of aggregation of similarly-valued cells.

The rest of this paper is organized as follows. Section 2 reviews the vertex algebra. Section 3 shows how to use this method to represent multidimensional objects with orthogonal faces (termed *orthogonal maps*). Section 4 introduces a data structure called a *vertex list* and several associated algorithms. Section 5 summarizes what is presently known about vertex representations and gives directions for further research.

2 Vertex Algebra

Let $p = (p_1, p_2 \cdots p_d)$ be a point in $\mathcal{R}^d$. We define the *cone* of p as the scalar field $Q_p : \mathcal{R}^d \rightarrow \mathcal{Z}$ where:

$$Q_p(x) = \begin{cases} 1 & \text{if } x_1 \geq p_1 \wedge x_2 \geq p_2 \wedge \cdots \wedge x_d \geq p_d \\ 0 & \text{otherwise.} \end{cases}$$

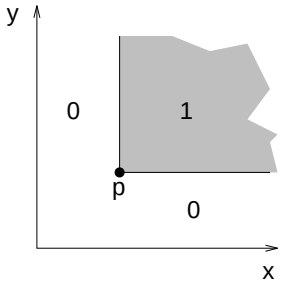


Figure 1: Cone of a point in $\mathcal{R}^2$

In other words, the cone of p maps to 1 all points inside a hyperrectangle with its minumum vertex at p and extending to infinity in the positive directions of all axes, and maps to 0 all other points. Figure 1 illustrates the scalar field represented by the cone of one single point in $\mathcal{R}^2$.

A *vertex* is defined as a weighted point, that is, a point with an associated scalar value (a point can be thought of as a vertex with weight 1). The cone of a vertex $v = (p, \alpha)$, where p is the *position* of v and α is the *weight* of v, is a scalar field given by $Q_v = \alpha \cdot Q_p$. That is, $Q_v(x)$ is α for all points such that $Q_p(x) = 1$ and 0 otherwise.

Similarly, the scalar field represented by a set of vertices $V = \{v_1, v_2 \cdots v_k\}$ is given by $Q_V = \sum_{i=1}^{k} Q_{v_i}$.

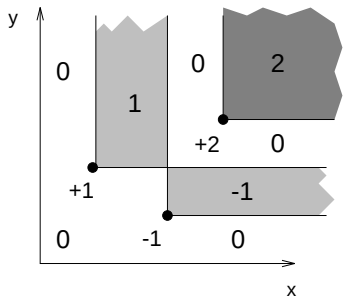


Figure 2: Scalar field represented by a set of vertices in $\mathcal{R}^2$

Figure 2 shows an example of a scalar field in $\mathcal{R}^2$ given by a set of 3 vertices. The labeled black dots correspond to the positions of the vertices and their weights.

It is useful to enumerate some properties of scalar fields expressed using vertex algebra. Let a and b be vertices, α and β be integers, and Q_v denote the scalar field represented by vertex v. Then,

1. $\alpha Q_a + \beta Q_a = (\alpha + \beta) Q_a$.
2. $0 Q_a = 0$.
3. If $\alpha Q_a + \beta Q_b + \cdots = 0$ and the positions of a, b etc are all distinct, then $\alpha = \beta = \cdots = 0$.

3 Orthogonal Maps and Polygons

In the following, we use the term *orthogonal map* to designate any scalar field that can be represented by a finite set of vertices. A set of vertices that represents an orthogonal map is termed a *vertex representation* and obeys the following rules:

1. No two vertices with the same positions are allowed. This is reasonable, since if $t = (p, \alpha)$ and $u = (p, \beta)$ are vertices in the set, then they can be replaced by vertex $v = (p, \alpha + \beta)$
2. Vertices with weight 0 are not allowed as they add no new information.

The main purpose of enforcing these rules on sets of vertices is to assure uniqueness, i.e., that every orthogonal map has exactly *one* vertex representation.

We define an *orthogonal polygon* to be a special case of an orthogonal map where all points are mapped either to 0 or to 1. Those points that are mapped to 0 are said to be *outside* the polygon and those that are mapped to 1 are *inside* the polygon. If we restrict ourselves to $\mathcal{R}^2$, the class of orthogonal polygons closely resembles the class of ordinary polygons whose edges are parallel to one of the coordinate axes. In fact, given any such polygon, we can contruct an equivalent vertex representation with the following algorithm. Since we have already used the term *vertex* to mean a point with an associated weight, we refer to the vertices of ordinary polygons as *articulation points*:

1. Select a *minimum* articulation point of the polygon and create a vertex with weight $+1$. A *minimum articulation point* is a point (x_0, y_0) such that no other point (x, y) on the boundary of the polygon has $x < x_0 \wedge y < y_0$. Note that more than one articulation point may satisfy these requirements.
2. Follow the boundary of the polygon starting from the minimum articulation point in a clockwise (or counterclockwise) direction. For each articulation point, create a vertex with weights alternating between -1 and $+1$, that is, $+1$ for point (x_0, y_0), -1 for point (x_1, y_1), $+1$ for point (x_2, y_2), and so on.

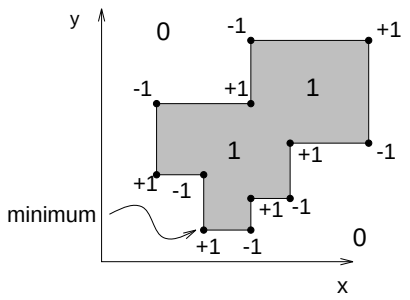


Figure 3: Vertex construction of an orthogonal polygon

3. Stop when reaching the minimum articulation point again.

The labeling process described above is illustrated in Figure 3. It must be observed that the algorithm is only well defined if in step 2 we know exactly which vertex/articulation point to visit next. In the case of self-intersecting polygons we have more than one choice at certain points and the resulting vertex representation may differ. This is illustrated in Figure 4. Notice that Figure 4(d) contains a vertex with weight 2, which is a result of visiting an articulation point twice while traversing the boundary of the polygon.

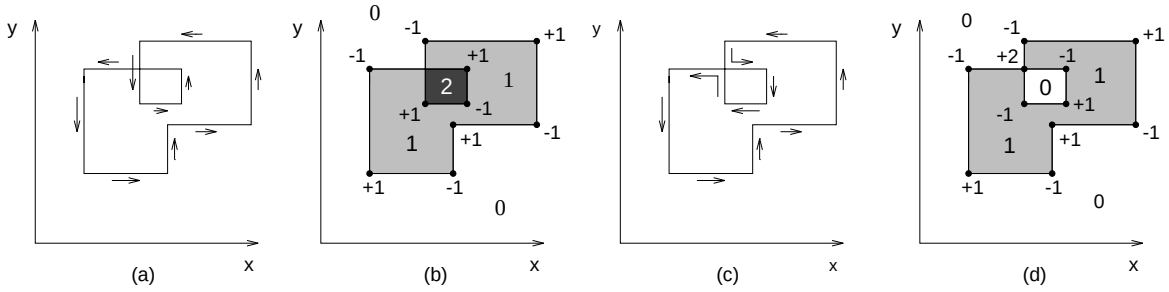


Figure 4: Two ways of buiding a vertex representation of a self-intersecting orthogonal polygon. Visiting the articulation points as in (a) yields the vertex representation given in (b), whereas visiting them as in (c) results in the vertex representation given in (d)

It is also possible to represent polygons with "holes". In this case, apply the algorithm to the boundary of each hole but label the starting minimum vertex with value -1 instead of $+1$. If we regard the example in Figure 4(a) as a polygon with one hole touching the outer boundary, then we would also get the vertex representation in Figure 4(d).

A similar construction algorithm may be devised for orthogonal polygons in any dimension. The task of selecting a minimum articulation point poses no difficulties. There is no obvious order, however, for traversing the articulation points. This doesn't

matter at all since, at least for non-self-intersecting polygons, all that is needed is to assign opposite weights to any two vertices that share an edge.

4 Vertex Lists

Most algorithms on vertex representations use the plane-sweep paradigm [2]. Moreover, plane-sweep is frequently applied recursively, that is, the solution of a problem in d dimensions is obtained by combining the partial solutions of many subproblems in $d-1$ dimensions. This suggests that vertices should be organized as a list in reverse lexicographic order by dimension. For example, letting $((u_1, u_2 \cdots u_d), \alpha)$ and $((v_1, v_2 \cdots v_d), \beta)$ be vertices u and v, respectively, we have that vertex u appears before vertex v in a vertex list if u_d (u's last coordinate value) is less than v_d. Similarly, if u_d is greater than v_d then u appears after v. If $u_d = v_d$ then u and v are on the same hyperplane and the ordering between them is determined by the $d-1$ remaining coordinate values, with coordinate value $d-1$ being the most significant. If all coordinate values of u and v are equal, then u and v are in the same position and they may be replaced by one vertex whose weight is the sum of the weights of u and v. For instance, the vertices $a = ((4,5),1)$, $b = ((3,2),2)$ and $c = ((4,2),-1)$ would appear in a list in the order $b\ c\ a$.

In the remaining sections, vertex list algorithms are given using the following conventions:

1. Uppercase letters such as L, R and S represent lists of vertices.
2. Lowercase letters from the end of the alphabet, such as u, v and w represent vertices.
3. Other lowercase letters, such as i, j and k represent integers.
4. $head(L)$ is the first vertex of L.
5. $tail(L)$ is L without its first vertex.
6. $concat(L, S)$ is list L concatenated with list S.
7. $concat(L, v)$ is list L concatenated with vertex v.
8. ϵ represents an empty list.
9. $dim(v)$ is the number of coordinates of v.
10. $weight(v)$ is the weight of v.
11. $coord(v, i)$ is the i^{th} coordinate value of v.
12. $proj(v)$ is vertex v with its last coordinate dropped.
13. $unproj(v, k)$ is vertex v with one additional coordinate with value k.
14. $precedes(u, v)$ is a Boolean function that is defined only if $dim(u) = dim(v)$. It returns $true$ if $u < v$ in reverse lexicographic order, i.e.,

$$\begin{aligned} precedes(u,v) \Leftrightarrow \quad & (coord(u, dim(u)) < coord(v, dim(v))) \vee \\ & (coord(u, dim(u)) = coord(v, dim(v)) \wedge \\ & precedes(proj(u), proj(v))) \end{aligned}$$

4.1 Sum of Vertex Lists

Given two vertex lists L and S of the same dimension, their sum is a vertex list that represents the sum of their orthogonal maps, that is $Q_{L+S} = Q_L + Q_S$. This operation is achieved by merging vertex lists L and S adding the weights of vertices in the same position. Similarly, the multiplication of a vertex list by a scalar α is equivalent to multiplying the weight of each vertex of the representation by α.

Procedure *add*, given below, implements these operations. Given vertex lists L and S and a scalar α, the algorithm returns the vertex list $L + \alpha S$ in vertex list R.

```
procedure add(L, S, α)
    R ← ε
    while L ≠ ε ∨ S ≠ ε do
        if S = ε ∨ precedes(head(L), head(S)) then
            R ← concat(R, head(L))
            L ← tail(L)
        else if L = ε ∨ precedes(head(S), head(L)) then
            v ← head(S)
            weight(v) ← α · weight(v)
            R ← concat(R, v)
            S ← tail(S)
        else
            v ← head(L)
            weight(v) ← weight(v) + α · weight(head(S))
            if weight(v) > 0 then R ← concat(R, v)
            L ← tail(L)
            S ← tail(S)
    return R
```

This algorithm is clearly linear in the total number of vertices in the two vertex lists, since at each iteration one vertex of L or S (or both) is removed and one (or none) is added to R. This claim can be contested on the grounds that comparing the positions of two d-dimensional points in space (as required by *precedes*) takes $O(d)$ time. Fortunately, though, the reverse lexicographic ordering of vertices in vertex lists enables this operation to be executed in $O(1)$ time.

4.2 Converting Rasters to Vertex Lists

A raster in d dimensions is defined by the value associated with each of its cells. Let $length(i)$ represent the length of the raster in the i^{th} dimension, and F be a list of the raster's values stored in reverse lexicographic order. For instance, if the raster is a 2×3 matrix, then $length(1)$ is 2, $length(2)$ is 3, and the values would be stored in F row-wise, i.e., first the element in row 1 column 1, then row 1 column 2, row 2 column 1, etc.

Each vertex represents the position of space where there is a change in value when the raster is viewed as an aggregation of one-dimensional rows stored in reverse lexicographic order. In one dimension, if two cells a and b with values α and β are neighbors, then the change in value is given by $\beta - \alpha$; if $\alpha = \beta$ then no vertex is necessary to indicate the change from a to b. Also, we assume that these changes are accumulated over an initial value of 0.

As an example, consider the one-dimensional raster given by values $\langle 5, 3, 3, 4 \rangle$: the corresponding vertex representation is given by $\langle ((1), +5), ((2), -2), ((4), +1) \rangle$ (see Figure 5). Notice that the interval $(-\infty, 1)$ is assumed to be mapped to 0, while interval $[4, +\infty)$ is mapped to the last value in the raster, i.e. 4. This means that changes in value accross different one-dimensional rows are not recorded. In other words, every row starts with the first non-zero location.

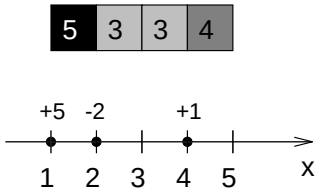


Figure 5: Conversion of a one-dimensional raster to a vertex list.

For higher dimensions the problem can be expressed as a combination of $length(d)$ subproblems of dimension $d - 1$. First, we find the vertex list for each sub-raster corresponding to the hyperplanes perpendicular to the d^{th} axis at coordinates $1, 2 \ldots length(d)$. The vertex representation for the d-dimensional raster is then computed by *subtracting* each previous vertex list from the current one. The rationale behind this procedure is that we are computing the *change* in value from one hyperplane to the next. Procedure *raster_to_vlist*, given below, implements this process and returns its result in vertex list R.

procedure *raster_to_vlist* $(d, length, F)$
    $R \leftarrow \epsilon$
    if $d = 1$ **then**
        $j \leftarrow 0$
        for i **in** $\{1 \ldots length(1)\}$ **do**
            $k \leftarrow weight(head(F))$
            $F \leftarrow tail(F)$
            if $j \neq k$ **then** $R \leftarrow concat(R, ((i), k - j))$
    else
        $S \leftarrow \epsilon$
        for i **in** $\{1 \ldots length(d)\}$ **do**
            $T \leftarrow$ *raster_to_vlist* $(d - 1, length, F)$
            $R \leftarrow concat(R, unproj(add(T, S, -1), i))$
            $S \leftarrow T$
    return R

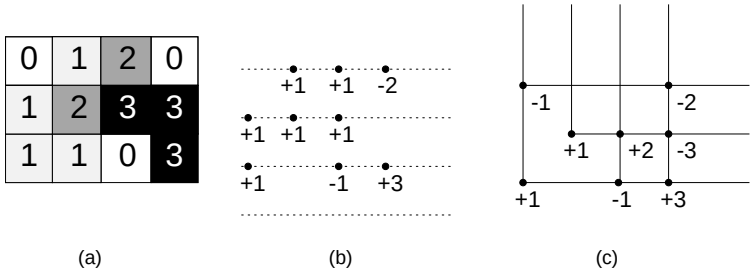


Figure 6: Converting a raster to a vertex representation. (a) A two-dimensional raster. Each raster line is converted to a one-dimensional vertex representation as given in (b). The subtraction of each pair of 1D vertex lists results in the 2D vertex representation given in (c).

Figure 6 shows an example in two dimensions. We can see that the algorithm reads each cell value exactly once and thus is linear in the number of cells. For a one-dimensional raster, the algorithm produces at most as many vertices as there are cells in the raster. This worst case corresponds to the case where each cell value is different from the previous one. For an d-dimensional raster, the algorithm outputs $length(d)$ lists of vertices of dimension $d-1$, each with potentially as many as $length(1) \times \ldots \times length(d-1)$ vertices. Thus, given a raster with N cells, the storage requirement is always $O(N)$. Also, each vertex list produced by the algorithm at dimensions $d > 1$ results from the addition of two vertex lists, a process that is linear in the number of vertices. This is done once for all but the first dimension and thus the algorithm executes in $O(N \cdot d)$ time.

4.3 Transformations and Set Operations

A common operation on orthogonal maps is to compose them with a transformation function $f : \mathcal{Z} \rightarrow \mathcal{Z}$. That is, if L is a vertex list representation of a particular orthogonal map Q_L, then to apply f on L is equivalent to evaluating $f(Q_L)$.

One major application of transformations is to perform set-theoretic operations. For instance, consider two orthogonal polygons represented by their vertex lists A and B. In the orthogonal map represented by the list $L = A + B$, those areas originally covered by either A or B are mapped to 1, while those areas covered by both A and B are mapped to 2. It is possible to obtain the *union* of A and B by applying to list L the transformation

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

Similarly, the following transformation on L may be used to obtain the *intersection* of A and B:

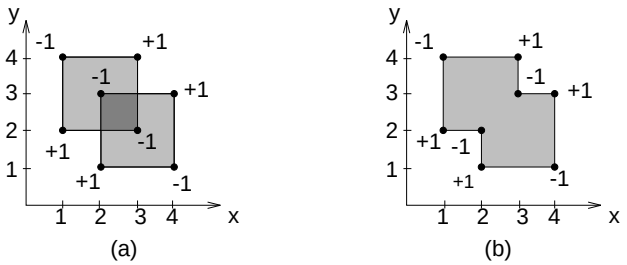
$$f_{\cap}(x) = \begin{cases} 1 & \text{if } x > 1 \\ 0 & \text{otherwise.} \end{cases}$$

The *set difference* may be obtained by first subtracting B from A and then applying transformation $f_{\cup}$.

The algorithm to apply a transformation to a vertex list is similar to the raster to vertex list conversion algorithm. Once again, we recursively sweep hyperplanes of decreasing dimension. The problem that we must address is that the values stored in the vertices correspond to differences in the values of cones in space rooted at the vertices, while the transformation involves the application of a function to the actual values of the points in space. This is easy to overcome by recalculating the values of the points in space during the sweep. In algorithm *transform* below, L is a vertex list of dimension d and f is a transformation. Variable i contains the value of the d^{th} coordinate for the current position of the sweeping hyperplane and I is the list of vertices on it. S and T are vertex lists corresponding to the value of the orthogonal map just before and just after the sweeping hyperplane. S' and T' are transformed versions of S and T, respectively.

```
procedure transform (L, f)
    R ← ε
    d ← dim(L)
    if d = 1 then
        j ← 0
        while L ≠ ε do
            v ← head(L)
            L ← tail(L)
            k ← weight(v) + j
            if f(k) ≠ f(j) then
                weight(v) ← f(k) − f(j)
                R ← concat(R, v)
            j ← k
    else
        S ← ε
        S' ← ε
        while L ≠ ε do
            i ← coord(head(L), n)
            I ← proj(ihead(L, i))
            T ← add(S, I, +1)
            L ← itail(L, i)
            T' ← transform(T, f)
            R ← concat(R, unproj(add(T', S', −1), i))
            S ← T
            S' ← T'
    return R
```

In procedure *transform*, function $ihead(L, i)$ returns the prefix of L where all vertices have their last coordinate value equal to i, while $itail(L, i)$ returns the remainder of L (i.e., L with the vertices in $ihead(L, i)$ dropped). Figure 7 shows how a two-



i	I	S	S'	T	T'
1	+1 -1			+1 -1	+1 -1
2	+1 -1	+1 -1	+1 -1	+1 +1 -1 -1	+1 -1
3	-1 +1	+1 +1 -1 -1	+1 -1	+1 -1	+1 -1
4	-1 +1	+1 -1	+1 -1		

(c)

Figure 7: Example of a transformation applied to a two-dimensional orthogonal map. A call to procedure *transform* with list L containing the vertices shown in (a) and $f = f_{\cup}$ results in the vertex representation given in (b), which is essentially $T' - S'$. Table (c) shows how I, S, T, S' and T' vary with i.

dimensional orthogonal map is transformed by $f_{\cup}$.

The complexity analysis for *transform* follows the same reasoning as in the raster to vertex list conversion algorithm. For a vertex list with N vertices of dimension d, the algorithm has a worst-case $O(N \cdot d)$ running time.

5 Concluding Remarks

A new representation known as vertex lists for orthogonal objects in arbitrary dimensions has been described. Although the basic idea has already been presented for two-dimensional data [7], our contribution has been the extension to higher dimensions. Its advantage lies in its compactness in comparison with the more traditional representations such as rasters and collections of hyperrectangles (disjoint and non-disjoint). Unlike rasters, the storage required for vertex lists is proportional to the boundary of the represented objects and remains invariant under scaling. Also, normalized vertex lists have the property of being unique, a property that can be exploited in many ways. For instance, Santos [6] showed how vertex lists are useful for the recognition of circuit elements in VLSI design.

Vertex lists are similar in spirit to the chain code with the advantage that they are

not restricted to two dimensions. Algorithms have been given for converting between vertex lists and a conventional raster representation as well as set-theoretic transformations for the construction of more complicated objects. The same techniques can also be used to perform contraction and expansion. This means that operations such as skeletonization can be implemented easily.

In general, vertex lists are not appropriate for localized querying – for instance, the extraction of the value of a particular point in space requires the traversal of the entire vertex list. These operations can be better performed using rasters or hierarchical structures such as *quadtrees* [4, 5]. Conversion of vertex lists to these other forms of represention can be done with little difficulty.

6 Acknowledgements

We have benefitted from discussions with J. Schechtman, whose work sparked our interest in vertex representations.

References

[1] H. Freeman. Computer processing of line-drawing images. *ACM Computing Surveys*, 6:57–97, March 1974.

[2] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[3] D. Rutovitz. Data structures for operations on digital images. In G. C. Cheng et al., editor, *Pictorial Pattern Recognition*, pages 105–133. Thompson Book Co., Washington DC, 1968.

[4] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.

[5] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[6] F. V. Santos. Extração de circuitos VLSI. Master's thesis, Eng. Elétrica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, April 1991.

[7] J. Shechtman. Processamento geométrico de máscaras VLSI. Master's thesis, Eng. Elétrica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, August 1992.