Vertex Representations and their Applications in Computer Graphics

Claudio Esperança [*]
COPPE, Prog. Eng. Sistemas
Universidade Federal do Rio de Janeiro
Cidade Universitária, C.T., Sala H319
Rio de Janeiro, RJ, 21945-970, Brazil
E-mail: esperanc@lcg.ufrj.br

Hanan Samet [†]
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Science
University of Maryland at College Park
College Park, Maryland 20742
E-mail: hjs@umiacs.umd.edu

March 30, 1998

Abstract

The vertex representation, a new data structure for representing and manipulating orthogonal objects, is presented. Both interiors and boundaries of regions are represented implicitly through the aid of a single vertex which is the tip of an infinite cone. The cones are similar to halfspaces in a CSG representation; however, unlike CSG, the representation of an object with the vertices is unique. Additionally, vertex representations deal with scalar fields and not solids. Algorithms are given for generating vertex representation models for primitive solids, performing affine transformations, set-theoretic operations, and displaying vertex representation models. The algorithms assume that the vertices are stored in a list although they could also be stored using other representations (e.g., a point variant of a quadtree). The contribution of the work lies, in part, in the generality of the solutions obtained 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 which was the original formulation of the representation. The vertex representation models the same objects that are obtained through use of methods such as an array of voxels or octrees (i.e., objects with orthogonal faces). Its advantage is that it requires less space and unlike octrees its space requirements are not sensitive to translation. Moreover, it enables increasing the quality of the approximation of objects by increasing the number of vertices that make up the model. In contrast, increasing the quality of the approximation in a array of voxels requires doubling the resolution which means a significant increase in the space requirements. The algorithms are illustrated via screen shots.

Keywords: Shape representation, orthogonal polygons, vertex lists, constructive solid geometry (CSG), solid modeling.

[*]The support of Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Projeto GEOTEC 481007/96-2 is gratefully acknowledged.

[†]The support of the National Science Foundation under Grant IRI-9712715, the Department of Energy under Contract DEFG0295ER25237 is gratefully acknowledged.

1 Introduction

There are many different techniques for representing objects in computer graphics applications (e.g., [6, 10, 15, 14]). The objects can be represented either by their interiors or by their boundaries. Each of these representations has its advantages and disadvantages. At times, a more compact representation is desired in which case techniques such as runlength encoding are applied.

In this paper we describe the vertex representation which represents both interiors and boundaries of regions implicitly through the aid of a single vertex which is the tip of an infinite cone. The cones are similar to halfspaces in a CSG representation; however, unlike CSG, the representation of an object with the vertices is unique. Also, the vertex representation aims at representing scalar fields, not solids. The modeling of solids is achieved by regarding a solid as as a scalar field which maps the interior of the solid to 1 and the exterior to 0. The vertex representation was originally presented as an alternative model for two-dimensional data represented as lists of rectangles for VLSI applications [18]. Algorithms have been devised for a number of vertex representation operations in two dimensions [18].

Our contribution here is in showing how the vertex representation can be extended to represent orthogonal objects in arbitrary dimensions, and presenting a number of algorithms for operations that are useful in computer graphics. In particular, we describe algorithms for generating vertex representation models for primitive solids, performing affine transformations, set operations, and displaying vertex representation models. Our algorithms assume that the vertices are stored in a list although they could also be stored using other representations (e.g., a point variant of a quadtree). We do not address these other representations in this paper.

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 [18].

The vertex representation models the same objects that are obtained through use of spatial enumeration methods such as an array of voxels or octrees [7] (i.e., objects with orthogonal faces). The vertex representation models the same objects that are obtained through use of methods such as an array of voxels or octrees (i.e., objects with orthogonal faces). Its advantage is that it requires less space and unlike octrees its space requirements are not sensitive to translation. Moreover, it enables increasing the quality of the approximation of objects by increasing the number of vertices that make up the model. In contrast, increasing the quality of the approximation in a array of voxels requires doubling the resolution which means a significant increase in the space requirements. i

The rest of this paper is organized as follows. Section 2 motivates the paper by reviewing a number of object representations, shows their strengths and weaknesses, and concludes with a brief explanation of how the latter can be overcome by the vertex representation. This section can be skipped by readers who do not need such a motivation. Section 3 reviews the vertex representation, while Section 4 presents some of its key properties. Section 5 describes the organization of the elements of the vertex representation by means of lists (termed *vertex lists*). This section also includes descriptions of a number of elementary operations that are used in the implementation of a number of solid modeling operations in Section 6. Concluding remarks are drawn in Section 7.

2 Object Representations

A common representation of the objects and their environment, and the one we focus on in this paper, is as a collection of cells (termed *pixels* and *voxels* in two and three dimensions, respectively) all of whose boundaries (with dimensionality one less than that of the cells) are of unit size. In the applications that we consider, the boundaries (i.e., edges and faces in two and three dimensions and higher, respectively) of the cells are orthogonal to each other and parallel to the coordinate axes. The elements of each collection (i.e., the cells) are labeled with values or attributes corresponding to the different objects and are frequently aggregated into subcollections of identically-valued cells. The shape of an object o can be represented either by the interiors of the cells comprising o, or by the subset of the boundaries of those cells comprising o that are adjacent to the boundary of o.

Interior-based representations aggregate identically-valued cells by recording their locations. The drawback is the amount of space needed. This is alleviated by aggregating similarly-valued unit-sized cells into *blocks* which are usually rectangular with sides that are parallel to the coordinate axes. Region quadtrees [4] and region octrees [7] are examples of such representations for two and three-dimensional data (variants for data of higher dimension also exist), respectively, that recursively decompose the environment containing the objects into four or eight, respectively, rectangular congruent blocks until each block is either completely occupied by an object or is empty. The drawback of region quadtrees and octrees is that their space requirements depend on their position and hence they are sensitive to translation. Interior-based representations are good for computing volume as well as determining whether or not a point is inside or outside the object.

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 location of the different boundary elements associated with each cell of each object and their nature (i.e., their orientation and the locations of the cells to which they are adjacent). For example, in two dimensions, the boundary elements are just the sides of the cells (i.e., unit vectors), while in three dimensions, the boundary elements are the faces of the cells (i.e., squares of unit area with a direction equal to the normal to the object). Boundary-based representations aggregate identically-valued cells whose boundary elements have the same direction, rather than just identically-valued cells as done by interior-based representations. In two dimensions, the aggregation yields boundary elements which are vectors whose length can be more than one.

Whichever boundary-based representation is used and regardless of whether any aggregation takes place, the representation must also enable the determination of the connectivity between individual boundary elements. The connectivity may be implicit or explicit (e.g., by specifying which boundary elements are connected). As an example of a boundary-based representation, let us consider two-dimensional objects in which case the boundary elements are vectors. The location of the vector is given by its start and end vertices. An object o has one more boundary than it has holes. Connectivity may be determined implicitly by ordering the boundary elements $e_{i,j}$ of boundary b_i of o so that the end vertex of the vector v_j corresponding to $e_{i,j}$ is the start vertex of the vector v_{j+1} corresponding to $e_{i,j+1}$. In dimensions higher than two, the relationship between the boundary elements associated with a particular object is more complex, as is its expression. Whereas in two dimensions we just have one type of a boundary element (i.e., an edge or a vector consisting of two vertices), in $d > 2$ dimensions, we have $d - 1$ different boundary elements (e.g., faces and edges in three dimensions). As we saw, in two dimensions, the ordered sequence of vectors is equivalent to an implicit specification of the boundary by virtue of the fact that each boundary element of an object can only be adjacent to two other elements. Thus consecutive boundary elements in the representation are implicitly connected. Implicit connectivity is not possible in $d > 2$ dimensions as there are 2^{d-1} different adjacencies (i.e., types of connectivities) per boundary element.

Nevertheless, in higher dimensions we do have a choice between an explicit and an implicit boundary-based representation. The boundary model (also known as *BRep* [1, 15]) is an example of an explicit boundary-based representation. Observe that in three dimensions, the boundary of an object is decomposed into a set of faces, edges, and vertices. The result is an explicit model based on a combined geometric and topological description of the object. The topology is captured by a set of relations that indicate explicitly how the faces, edges, and vertices are connected to each other. The drawback is that we need to update the connectivity information after a series of operations.

Constructive Solid Geometry (CSG) [11] is an example of an interior-based representation that is applicable to objects of arbitrary dimensionality. In this case, primitive instances of objects are combined to form more complex objects by use of geometric transformations and regularized Boolean set operations (e.g., union, intersection). When the primitive instances are halfspaces, the result is an boundary-based representation. In this case, the boundary of a d-dimensional object o consists of a collection of hyperplanes in d-dimensional space (i.e., the infinite boundaries of regions defined by the inequality $\sum_{i=0}^{d} a_i x_i \geq 0$ where $x_0 = 1$). We have one halfspace for each $(d-1)$-dimensional boundary element of o.

Both of these representations (i.e., primitive instances of objects and halfspaces) are implicit because the object is determined by associating a set of regular Boolean set operations with the collection of primitive instances (which may be halfspaces) the result of which is the object o. Although both BRep and CSG are quite general, it is easy to constrain them to handle orthogonal objects. The drawback of these representations is that they are not unique. At times, some operations are more easily computed by conversion to interior-based representations (e.g., [16]).

Some interior-based and boundary-based representations can be made more compact by making use of an ordering on the data (i.e., the cells, blocks, boundary elements, etc.) that it records (if one exists). In particular, instead of just recording the actual values of the cells and/or the actual locations of the cells, it records the *change* in the values of the cells and/or the locations of the cells where the value changes. The ordering is used to reconstruct the original association of values with their locations whenever we wish to determine what value is associated with a particular location. Although the reconstruction process is a costly one as it requires traversing the entire representation (i.e., all the objects), this is not a necessarily a problem when execution of the operations inherently requires that the entire representation be traversed (e.g., Boolean set operations, data structure conversion, erosion, dilation, etc.). On the other hand, operations which require the ability to randomly access a cell in the representation (e.g., neighbor finding) are inefficient in this case unless an additional access structure is imposed on the compact representation.

Runlength encoding [13] is an example of a method that can be applied to make an interior-based representation such as the array more compact. The actual locations of the cells are not recorded. Instead, it aggregates contiguous identically-valued cells into one-dimensional rectangles for which only the value associated with the cells comprising the rectangle and the length of the rectangle are recorded. The one-dimensional rectangles are ordered in raster-scan order which means that row 1 is followed by row 2, etc. The same technique is applicable to higher dimensional data in which case the one-dimensional rectangles are ordered in terms of the rows, planes, etc. Given the location of any cell c in one-dimensional rectangle r, the value associated with c can be computed by referring to the location of the starting position of the entire representation and accumulating the lengths of the one-dimensional rectangles encountered before r. Interestingly, quadtrees and octrees can also be viewed as applications of runlength encoding to form aggregates in higher dimensions.

The same principles used in adapting runlength encoding to interior-based representations can also be used to a limited extent with some boundary-based representations. For example, in two dimensions, a common boundary-based representation is the chain code [3] which indicates the relative locations of the cells that comprise the boundary of object o. In particular, it specifies the direction of each boundary element and the number of adjacent unit-sized cells that make up the boundary element. Given the location of the first cell in the sequence corresponding to the boundary of o, we can determine the boundary of o by just following the direction of the associated boundary element. Thus there is no need for the actual cells or their locations, as now o is uniquely identified. Therefore, in the chain code, the locations of the boundary elements (i.e., the cells that are adjacent to them) are given implicitly while the nature of the boundary elements is given explicitly (i.e., the direction of the boundary element). Each object has a separate chain code.

Unfortunately, as we mentioned earlier, the chain code cannot be used for data in higher dimensions than two for the same reasons that the polygon representation could not be used. The problem is that the boundary elements are hyperplanes whose dimension is one less than the space in which they are embedded, and no natural order for traversing them exists. Thus we must turn to other solutions such as those based on the vertex representation that are described in greater detail in the remaining sections. For the moment, we complete our discussion of the various object representations by giving a brief overview of the salient properties of the vertex representation.

The vertex representation is based on the *vertex algebra* which was proposed by Shechtman [18] for storing and processing VLSI masks. The vertex representation applies techniques similar to that of runlength encoding to a orthogonal objects that are represented using blocks of arbitrary size and at arbitrary positions. Space is decomposed into multidimensional blocks (not necessarily disjoint) which are aggregated using techniques such as CSG by attaching weights that serve an analogous role to that of regularized union and set-difference. Instead of using combining elements that are blocks of finite area or halfspaces, it makes use of vertices which serve as tips of infinite cones.

Each cone is equivalent to an unbounded object formed by the intersection of d orthogonal halfspaces that are parallel to the d coordinate axes passing through the vertex. In a sense, our representation resembles CSG where the primitive elements are halfspaces. In particular, the vertex plays the role of a CSG primitive and the region that it represents is equivalent to the intersection of the halfspaces that pass through it. However, unlike CSG, the representation of an object with the vertices is unique. The vertices have signs associated with them indicating whether the space spanned by their cones are included or excluded from the object being modeled. The cones corresponding to the vertices are like infinite blocks thereby slightly resembling a region octree. However, unlike the region octree, since they need not be disjoint, they are invariant under translation.

3 Vertex Representations

A *vertex* is merely a point in space to which a scalar value (termed *weight*) is attached. A *vertex representation* is a set of vertices on which the following restrictions are imposed:

1. No two vertices may lie at the same point in space.
2. No vertices may have zero weight.

The modeling space[1] of vertex representations are scalar fields, that is, functions that map each point of a given d-dimensional space to a scalar value.

A *null* vertex representation (an empty set) represents a null scalar field by definition, i.e., a scalar field where all points are mapped to zero. Now consider a set containing only one vertex v placed at point $p(v)$ with weight $w(v)$. The effect of v on the otherwise null field is to add $w(v)$ to all points q such that $p(v) \leq q$. The relationship denoted by '$\leq$' is defined in the following manner: Let $p_1(v)$, $p_2(v)$ ... $p_d(v)$ be the d coordinate values of a point $p(v)$, and let q_1, q_2 ... q_d be the d coordinate values of a point q. Then, $p(v) \leq q$ if and only if for all $i = 1...d$, $p_i(v) \leq q_i$. We may visualize the region in space that a vertex v placed at $p(v)$ influences by imagining a flat-faced cone having its tip at $p(v)$. Figure 1 depicts such a cone in three-dimensional space.

As more vertices are added to the representation, the scalar field is modified accordingly. If we consider a representation V consisting of n vertices $v_1, v_2 ... v_n$, where the position of vertex v_i is denoted by $p(v_i)$

[1]Here we use a term defined by Requicha[10] in the context of solid modeling, despite the fact that the objects modeled via vertex representations are not rigid solids.

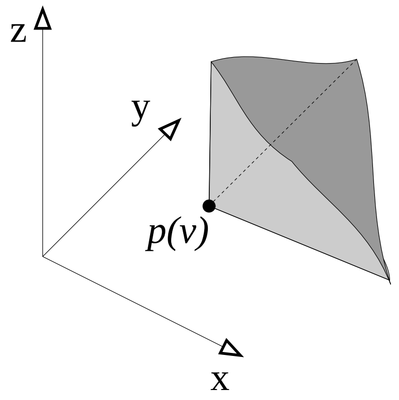


Figure 1: Cone of a vertex v placed at point $p(v)$ in $\mathcal{R}^3$.

and its weight by $w(v_i)$, then the value of the corresponding scalar field Q_V at a point q is given by

$$Q_V(q) = \sum_{p(v_i)\leq q} w(v_i)$$

Although the definition of vertex representations is complete at this point, it is rather difficult to visualize the process of building a representation which approximates any given scalar field. Perhaps the most useful insight in this matter is to observe how vertex representations behave in spaces of increasing dimensions. Consider the task of building a vertex representation for a field that maps all points to 0, except those *inside* a given hyperrectangle which are mapped to 1.

We start by analyzing the one-dimensional case, where the rectangle is actually an interval defined by two endpoints a and b. Formally, given an interval $R_1 = (a, b)$, we wish to build a scalar field Q_{R_1} defined as

$$Q_{R_1}(q) = \begin{cases} 1 & \text{if } a < q < b \\ 0 & \text{otherwise.} \end{cases}$$

The solution is trivial enough and consists of a representation with two vertices: one at position a and weight $+1$, and another at position b and weight -1 (see Figure 2). Observe that this is only an approximation of the desired field, since the field induced by the representation maps point a to 1 and not to 0.

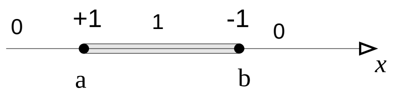


Figure 2: Vertex representation of a rectangle in one dimension.

Let us now step into the analogous two-dimensional problem. A two-dimensional axes-aligned rectangle (R_2)can be defined by two extreme points a and b. In other words, we wish to build a representation for a scalar field defined by

$$Q_{R_2}(q) = \begin{cases} 1 & \text{if } a_1 < q_1 < b_1 \wedge a_2 < q_2 < b_2 \\ 0 & \text{otherwise.} \end{cases}$$

The vertex representation obtained for the one-dimensional problem provides one half of the answer to the problem in two dimensions. By adding a second coordinate value equal to a_2 to each of those two vertices we obtain a rectangle that extends from a_2 to infinity in the positive direction of the y axis (see Figure 3(a)). All that remains is to add two more vertices at (a_1, b_2) and (b_1, b_2) with weights -1 and $+1$, respectively (see Figure 3(b)).

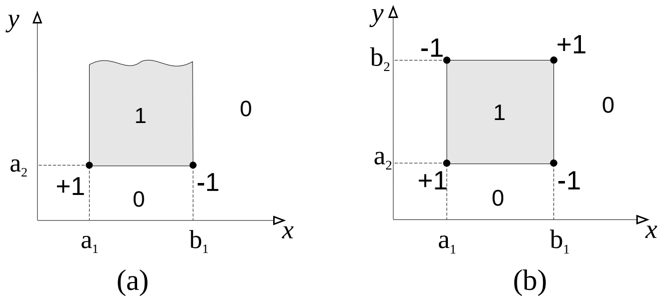


Figure 3: A rectangle in 2 dimensions is produced by embedding a one-dimensional rectangle in two-dimensional space (a) and "closing" it with a second one-dimensional rectangle (b).

The solution to the general problem of building a vertex representation for a hyperrectangle in d dimensions can then be seen as a recursive process. If d is 1, the solution is as described earlier. Otherwise:

1. Build a vertex representation for a $d-1$-dimensional rectangle obtained by discarding the last coordinate values of the extreme points, say, a_d and b_d.
2. Make two copies of the vertex representation obtained in (1). Set the d^{th} coordinate value of all vertices of the first copy to a_d and those of the second copy to b_d. Switch the weights of the vertices of the second copy from -1 to $+1$ and vice-versa.

Although simple-minded, the examples above illustrate a general way of processing vertex representations. In a nutshell, vertex representations defined in d-dimensional space can be processed by combining vertex representations in $d-1$-dimensional space. This general method of processing geometric structures is known as the *plane-sweep* paradigm [9]. We shall return to this subject in Section 5.

4 Properties of Vertex Representations

Vertices, as defined in Section 3, define a vector space. We shall not give here a mathematical treatment of this matter, but it is useful to enumerate a few properties of vertex representations. In the discussion that follows, α and β are scalar values, u and v are vertices, Q_v is the field induced by v, $p(v)$ is the position of v and $w(v)$ is the weight of v.

Scalar multiplication: Multiplication of a vertex v by a scalar α, written $\alpha \cdot v$ means a vertex with the same position as v and weight equal to $\alpha \cdot w(v)$. The effect of this operation is a field which maps all points in the cone of v to $\alpha \cdot w(v)$. Formally, $Q_{\alpha \cdot v} = \alpha \cdot Q_v$.

Vertex addition: Adding two vertices u and v, written $u + v$ is tantamount to putting them in the same representation. The field produced by $u + v$ is the sum of the fields produced by u and v individually. Formally, $Q_{u+v} = Q_u + Q_v$.

Uniqueness of vertex representations: Vertex representations are unique in the sense that either a given field Q can be represented by exactly one set of vertices or it cannot be represented by vertices at all. Uniqueness is ensured by the restrictions outlined in Section 3.

Boundary property: Loosely speaking, this property means that the vertex representation of a field Q will be constituted solely of vertices placed at points where there is a change in the value of the field, that is, at points on the boundary of the field. To put it another way, if all points in the neighborhood of a given point p are mapped to the same scalar α, then no vertex exists at p. The rectangle example of Section 3 demonstrate this property, as we can see that vertices needed to be placed only at the corners of the rectangle.

5 Vertex Lists

A convenient way of organizing the vertices of a representation is by means of lists. A list data structure requires that its elements be linked together in some order. Since vertex representations naturally lend themselves to be processed one dimension at a time, it is a good idea to maintain vertices sorted according to the order in which a sweeping plane would encounter them. This means that a vertex list in one dimension has vertices sorted by their x coordinate values. Similarly, the vertices of a two-dimensional vertex list are sorted primarily with respect to their y coordinate values with ties broken by taking their x coordinate values into account. In general, the vertices of a list in d dimensions are sorted primarily by their d^{th} coordinate values, while coordinates $d-1$, $d-2$ and so on act as secondary keys, tertiary keys etc. The fact that two vertices in the same representation cannot lie at the same position in space ensures that this scheme imposes a total ordering among the vertices of a list.

Let us return to the problem of building a vertex representation for a hyperrectangle in three dimensions. As seen in Section 3, an algorithm to perform this task with vertex lists would start by creating a one-dimensional vertex list, say $L1$, with vertices placed along the x axis. A two-dimensional vertex list $L2$ would then be created by concatenating two slightly modified copies of $L1$ placed at two different positions along the y axis. Finally, a three-dimensional vertex list $L3$ would be created by concatenating two modified copies of $L2$ placed at two different positions along the z axis. This process is illustrated in Figure 4.

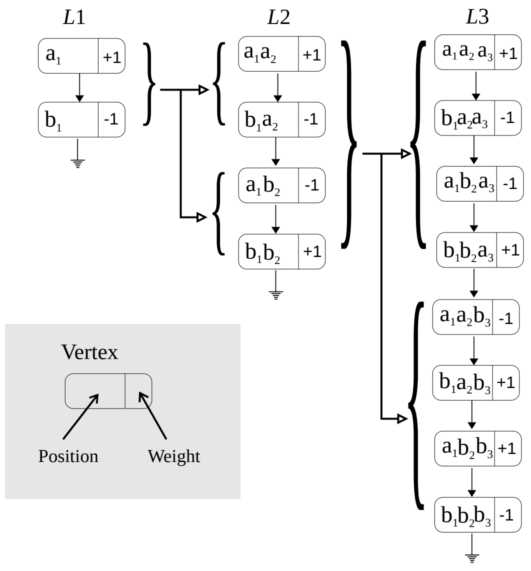


Figure 4: Building a vertex list for a three-dimensional rectangle by copying and concatenating vertex lists of lower dimensions

5.1 Elementary vertex list operations

The processing of vertex list models usually follows the same approach employed in the hyperrectangle problem. We will use the term *recursive plane sweep* to refer to this approach. At this point, however, we should familiarize ourselves with a few elementary operations employed in most – if not all – algorithms dealing with vertex lists.

Common list operations: We assume familiarity with a few operations generally defined on lists:

- ϵ denotes an empty (null) list.
- $head(L)$ is the first element of list L.
- $tail(L)$ is L minus its first element.
- $append(L1, L2)$ is a list formed by all elements of $L1$ followed by all elements of $L2$.

Prefix and Remainder: The prefix of a d-dimensional vertex list L is a sublist $\mathit{prefix}(L)$ extracted from the beginning of L such that all of its vertices have the same d^{th} coordinate value. The geometric significance of this operation lies in the fact that all vertices of $\mathit{prefix}(L)$ are guaranteed to lie on the same hyperplane. Similarly, $remainder(L)$ is the sublist composed of all remaining vertices of L after its prefix is extracted. Example: given $L3$ as depicted in Figure 4, $\mathit{prefix}(L3)$ is composed of the first 4 vertices of $L3$, whereas $\mathit{prefix}(remainder(L3))$ is composed of the 4 last vertices of $L3$.

Scalar multiplication: We denote by $\alpha \cdot L$ a list composed of all vertices of L with weights multiplied by α, where α is a non-zero scalar value.

Project and Unproject: The *project* operation takes a d-dimensional vertex list L and produces a $d-1$-dimensional vertex list by discarding the d^{th} coordinate value of every vertex in L. If all vertices of L lie on the same hyperplane $x_d = h$, then *unproject* is the inverse (dual) operation of *project* — i.e., $unproject(project(L), h) = L$. For example, in Figure 4, $L2 = project(\mathit{prefix}(L3))$ and $L3 = append(unproject(L2, a_3), -1 \cdot unproject(L2, b_3))$.

Sum and Difference: The sum of two vertex lists L and P, written $L + P$, is a vertex list that represents the scalar field $Q_L + Q_P$. This list is composed of all vertices of both L and P reordered lexicographically as discussed in Section 5. The difference between L and P (i.e. $L - P$) is equivalent to the sum of L and $-1 \cdot P$. A simple "balanced-line" algorithm can be used to compute the sum of L and P in linear time: Traverse both lists simultaneously and choose either $head(L)$ or $head(P)$ as the next vertex to be appended to the result. If both vertices are at the same position and the sum of their weights is non-zero, then append to the result a vertex at that position and weight equal to $w(head(L)) + w(head(P))$. Figure 5 shows an example.

5.2 Recursive plane sweep

Most operations on fields represented by vertex lists may be implemented using the plane sweep approach. Essentially, there are two problems to be considered:

1. Given a list L, how to evaluate field Q_L at each point.
2. Given a field Q, how to build a list L such that Q_L is a good approximation of Q.

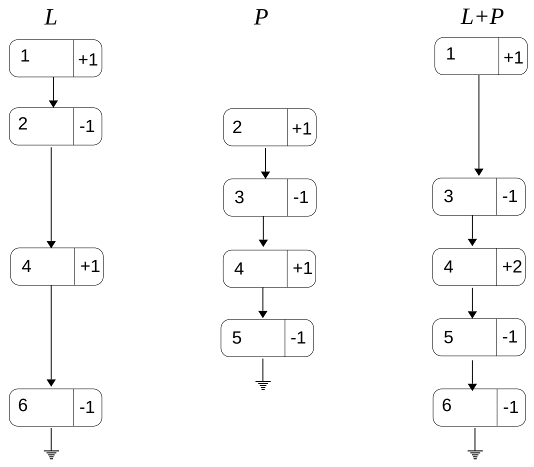


Figure 5: Example of the sum of two one-dimensional vertex lists.

These are dual problems, i.e., once one of them is solved, the solution to the other should follow the same lines.

Consider the first problem. Suppose that $L1$ is a one-dimensional list composed of vertices $v_1, v_2 ... v_n$. If a point q at q_x such that $p(v_i) \leq q_x < p(v_{i+1})$, then field Q_{L1} can be evaluated at q by *adding together* the weights of all vertices from v_1 to v_i. Thus, by traversing the list sequentially and adding vertex weights at each step we can evaluate the field for all points along the x axis. We may view this as a plane sweep in one dimension.

The same problem can be solved for a two-dimensional list $L2$ by sweeping a line perpendicular to the y axis (see Figure 6).

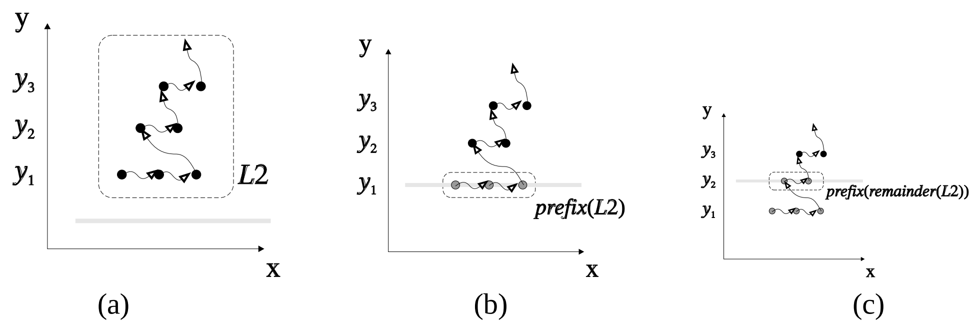


Figure 6: Plane sweep of vertex list $L2$. Stops correspond to successive prefixes of $L2$.

Since we wish to keep a data structure which records the state of the field at the sweeping line, we create a one-dimensional vertex list $L1$ to serve that purpose. At first, we may imagine the sweeping line stopped just before meeting the first vertex of $L2$ (Figure 6(a)). If the y coordinate value of this vertex is y_1, then, by definition, the field Q_{L2} is zero for all points such that $y < y_1$. Thus, the proper value for $L1$ at this point is ϵ. Moving the sweeping line so that it stops exactly at $y = y_1$ (Figure 6(b)), we notice that Q_{L2} is only influenced by vertices such that $y \leq y_1$. This is to say that Q_{L2} may be evaluated for all points of the sweeping line (at that position) by examining $\mathit{prefix}(L2)$. Thus, we update the state of the sweeping line by setting $L1$ to $project(\mathit{prefix}(L2))$. We may now move the sweeping line to the next position where the field changes, say y_2 (Figure 6(c)). The changes introduced at y_2 correspond to the vertices in $\mathit{prefix}(remainder(L2))$, or what we might call the second prefix of $L2$. Since we no longer need to keep the first prefix of $L2$ around, we may set $L2$ to its $remainder$. After this is done, we may update the state of the sweeping line by adding $project(prefix(L2))$ to $L1$. The whole list can be processed in this way until $L2$ is empty, that is, until no more prefixes can be found. The algorithm below summarizes the operations involved:

```
$L1 = \epsilon$
**while** $L2 \neq \epsilon$ **do**
    $L1 = L1 + project(prefix(L2))$
    **evaluate** $Q_{L1}$
    $L2 = remainder(L2)$
**enddo**
```

Plane sweeping can also be used to solve the dual problem. In one dimension, if Q is sampled at $x_1, x_2 ... x_n$, then a corresponding vertex list $L1$ can be built easily: First compute vertices $v_1, v_2 ... v_n$ where $p(v_i) = x_i$ and $w(v_i) = Q(x_i) - Q(x_{i-1})$ (in computing $w(v_1)$ we assume $Q(x_0) = 0$), then append to $L1$ only those vertices with non-zero weights.

Suppose that a two-dimensional field Q is sampled along planes $y = y_1, y_2 .. y_k$. For each plane, we may build a one-dimensional list $L1$ as above. We want to build a two-dimensional list $L2$ where each prefix of $L2$ corresponds to the difference between two successive values of $L1$. In other words, if $L1_1, L1_2 .. L1_k$ are the lists obtained at each stop of the sweeping plane, the first prefix of $L2$ will be $unproject(L1_1 - \epsilon, y_1)$, the second prefix of $L2$ will be $unproject(L1_2 - L1_1, y_2)$ and so on. The procedure can be summarized as follows:

```
$L2 = \epsilon$
$P1 = \epsilon$
**for** $y = y_1, y_2 ... y_k$ **do**
    $L1 =$ **sample for plane** $y$
    $L2 = append(L2, unproject(L1 - P1, y))$
    $P1 = L1$
**enddo**
```

Notice that $P1$ is an auxiliary variable that is used to carry the value of $L1$ from one iteration to the next. As we can see, this algorithm is indeed the dual of the algorithm presented earlier: instead of adding lists, we subtract them; $unproject$ is used instead of $project$, and instead of using $prefix$ and $remainder$ to split lists, they are joined with $append$.

6 Solid modeling with vertex lists

The goal of solid modeling is to be able to answer arbitrary geometric questions algorithmically (i.e., without human intervention) [6]. In general, answers to such questions involve building data structures that describe the geometric locus of a solid (a model), as well as the development of a set of algorithms that implement geometric operations on these structures. Some of the capabilities most usually found in solid modeling applications are:

1. Generate a model that approximates a solid defined algebraically.
2. Perform affine transformations (e.g., rotation, translation and scale).
3. Perform set operations (i.e., union, intersection and set difference).
4. Display (render) models.

All of these tasks can be performed with relative ease by using vertex lists as a data structure for solid models.

6.1 Expressive power

Vertex representations in general have the same expressive power as other spatial enumeration schemes such as octrees [7]. In essence, building a vertex list that represents a given solid entails the modeling of a scalar field that maps points in the interior of the solid to 1 and those outside the solid to 0. Unfortunately, due to the nature of vertex representations, there is not a simple way of consistently mapping points on the boundary of the solid to either 1 or 0. In addition, object faces are constrained to lie on a plane that is orthogonal to one of the coordinate axes. This fact implies that, as is the case with octrees, vertex lists are only able to represent approximations of objects delimited by curved surfaces.

On the other hand, approximating a solid by means of vertex lists does not require that the embedding space be discretized *a priori,* since the coordinate values stored with each vertex are limited only by the precision of the particular integer or floating-point representation being used. In other words, it is not necessary to map the modeling universe to fixed number of voxels which will serve as a discretization measure for all objects. Rather, the constraints imposed on vertex list models are usually expressed as an upper bound on the total number of vertices. As a consequence, it is possible to tune the desired level of approximation on a per-model basis. For instance, one may impose a limit of no more than 20 vertices per model, in which case a sphere, regardless of its radius, would be represented by a list containing the same number of vertices. Alternatively, one could constrain the maximum number of vertices to be a function of the overall size of the object.

6.2 Space complexity

As noted in Section 4, vertices tend to cluster at the boundaries of regions which are uniformly mapped to the same value. Consequently, the vertex representation of a solid will contain vertices placed at certain points along its (approximated) faces. Informally, the space complexity of a vertex list model is proportional to the complexity of its boundaries. For instance, we may sample a sphere of radius r by casting rays along the x axis and recording the regions where each ray intersects the model as a very thin and long parallelepiped (see Section 6.3.1). If these samples are taken by spacing rays every δ units on the y and z axes, then we may expect that roughly $(\pi r)^2/\delta^2$ rays will actually hit the sphere. Since each parallelepiped (ray) can be modeled with eight vertices, the space complexity of a vertex list that models that sphere will have $O((r/\delta)^2)$ as an upper bound.

It is instructive to compare this with similar results obtained for octrees. It has been shown [8] that the octree encoding takes space that, on average, is proportional to the boundary of the object. The space complexity of octree encoding, however, is sensitive to translation of the object within the space being mapped by the octree. For instance, an axes-aligned cube occupying 8 voxels out of an octree universe of $4 \times 4 \times 4$ voxels may result in an encoding having either 1 or 8 black nodes. On the other hand, a vertex representation of any axes-aligned parallelepiped always requires exactly 8 vertices.

In order to compare the space complexity of vertex representations with respect to octrees, let us consider an octree representation of a given solid. We know that octree representations are not translation invariant, and thus the same solid will correspond to different collections of leaf nodes depending on its position inside the world space represented by the octree. Let us call N_{min} the number of leaf nodes of the smallest among all these collections. Since we know that vertex representations *are* translation invariant, then it follows that a vertex representation of this solid will contain at most $8 \cdot N_{\text{min}}$ vertices (i.e., 8 vertices for each leaf node).

6.3 Solid modeling operations

In order to better evaluate the suitability of vertex lists as a basis for solid modeling, we describe below the overall mechanics involved in implementing some of the most common operations

6.3.1 Generating primitive vertex list models

A minimum requirement of a solid modeling system is to be able to create models for some primitive solids. These usually include polyhedral solids and solids delimited by quadric surfaces such as spheres, ellipsoids, cones, cylinders, etc.

In order to generate such models in the form of vertex lists, we could adapt many approaches currently in use for other spatial enumeration schemes. The one we describe here is an adaptation of the octree generating approach proposed by Franklin and Akman [2]. Essentially, the model is built by stacking rectangular parallelepipeds approximating the object. These data can be acquired, for instance, by casting parallel rays along the x axis and perpendicular to the $y - z$ plane [12].

Ray casting provides the means to sample the solid along a line. This can be used as the base case in a recursive plane sweep algorithm — that is, each sample will generate a one-dimensional vertex list. The algorithm proceeds by combining one-dimensional samples along a plane in order to generate two-dimensional vertex lists which will represent a cross-section of the solid. Finally, the model is built by combining consecutive two-dimensional cross-sections. Figure 7 illustrates this process for a simple model sampled with 16 rays. Notice that each ray takes a sample of the solid along a single line, but the

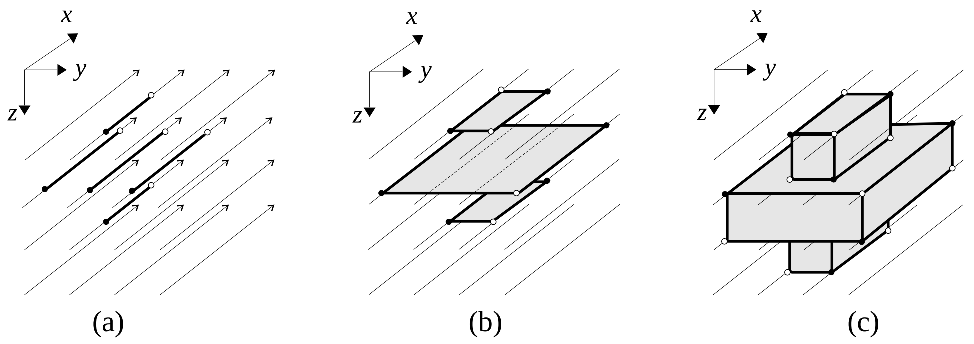


Figure 7: A solid is sampled with rays generating one-dimensional vertex lists (a). These are combined in (b) to generate two-dimensional cross-sections, which are combined to produce the final 3D vertex list model. Black/white dots represent vertices with weights +1/-1.

construction algorithm assumes that this sample holds true until the next ray is fired. For example, in the first row of rays across the top of Figure 7(a), only one ray hits the solid whereas the corresponding cross section in Figure 7(b) is depicted as a rectangle. This is due to the mechanics of sweeping plane algorithms, i.e, the state of the sweeping plane is assumed not to change between two successive stops. The same effect can be observed when a rectangular cross section is swept from one row of rays to the next generating parallelepipeds (Figure 7(c)). The algorithm is as follows:

```
$L3 = \epsilon$
$P2 = \epsilon$
**for** $z = zmin$ **to** $zmax$ **by** $zincr$ **do**
    $L2 = \epsilon$
    $P1 = \epsilon$
    **for** $y = ymin$ **to** $ymax$ **by** $yincr$ **do**
        $L1 =$ **raycast**$(y, z)$
        $L2 = append(L2, unproject(L1 - P1, y))$
        $P1 = L1$
    **enddo**
    $L3 = append(L3, unproject(L2 - P2, z))$
    $P2 = L2$
enddo
return L3
```

In the algorithm above, **raycast**(y, z) denotes the one-dimensional vertex list obtained by casting a ray parallel to the x axis and passing through point $(0, y, z)$. Such a list will have vertices with weights $+1$ or -1 at those points where the ray enters or leaves the solid, respectively (see Figure 7(a)). Two-dimensional vertex lists ($L2$) are computed incrementally by subtracting successive values of list $L1$ (innermost loop). A temporary list $P1$ is used to carry previous values of $L1$ from one iteration to the next. The outermost loop is entirely analogous to the inner loop, except that two-dimensional lists are combined to obtain the final three-dimensional list.

The analysis of this algorithm reveals that if each one-dimensional vertex list contains at most n vertices (typically, $n = 2$ for convex objects), then we may expect at most $N = n \cdot ny \cdot nz$ vertices in the final model, where $ny \cdot nz$ represents the resolution of the sampling. Each vertex v in the final model took part in at most two difference operations between two-dimensional lists: the first as an element of $L2$ and the second as an element of $P2$. Similarly, v took part in at most two difference operations between one-dimensional lists: the first as an element of $L1$ and the second as an element of $P1$. Thus, the fraction of the total cost that is dependent on the lengths of the lists involved is bounded by $O(N)$. Since the remaining operations (including $append$ and ray casting) can safely be considered to take constant time, we may estimate the total time complexity of the algorithm to be $O(N)$.

6.3.2 Affine transformations

Since the elements of a vertex list carry with themselves their own coordinate values, any transformation which involves only the modification of such values without affecting the ordering can be performed by a single sequential traversal of the list. Thus, translation and scale by positive factors can be computed in linear time.

Special cases of rotation transformations can be computed by reordering the vertices of a list. These cases correspond to transformations which can be expressed by swapping coordinate axes. Resorting a list with N elements can be performed in $O(N \log N)$ time at worst, but it is possible to improve this bound under certain circumstances by using a radix sorting algorithm [5].

For example, a rotation of 120^o around vector $(1, 1, 1)$ can be thought of as swapping axes y with x, z with y and x with z (see Figure 8). In other words, a vertex at (x, y, z) would be moved to position (y, z, x). The ordering scheme of vertex lists stipulates that the third coordinate (z in the original list) is the primary sorting key, with ties broken by considering the second and first coordinates (y and z, respectively). Notice that the relative ordering of coordinates y and z is unchanged by the proposed rotation, i.e., if two vertices have the same x coordinate values in the original list, then they will appear in the same relative order in the rotated list. This means that a list for the rotated solid may be obtained by performing an *order-preserving* sort on coordinate x of the original list.

Assuming that the model was sampled in regular intervals along the x axis, say at $x = 1, 2 \ldots n$, we could use the x coordinate values as an index. The idea is to distribute the vertices of the original list into an array of "buckets" addressed by x, say $B[x]$. After all vertices were placed in some bucket, the contents of $B[1], B[2] \ldots B[n]$ are threaded together resulting in the reordered list. This is an instance of radix sorting. To summarize:

1. Allocate an array of lists $B[]$, where each $B[i]$ will contain all vertices having $x = i$.
2. Scan the original list appending each vertex at (x, y, z) to list $B[x]$.

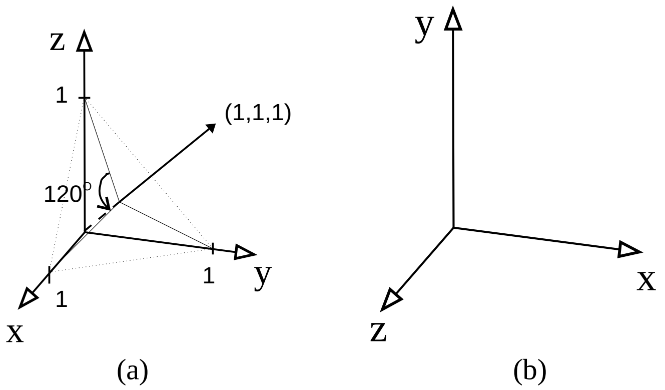


Figure 8: A 120° rotation around vector $(1, 1, 1)$ (a) has the effect of swapping axes (b).

3. Create result list R.
4. Scan array $B[]$ sequentially appending each list to R.

Other affine transformations will require a resampling of the model. One way to do this is to use the ray casting algorithm discussed in Section 6.3.1. Roth [12] discusses how a general affine transformation T can be achieved by casting a ray to which the inverse transformation T^{-1} was applied.

6.3.3 Set operations

All Boolean set operations (union, intersection, set difference) between two vertex list models L and P can be performed with the aid of an algorithm for applying scalar transformations to vertex lists. Observe that Q_{L+P} is a scalar field that maps all points of space to either 0, 1 or 2. Points mapped to 0 are outside L and outside P; points mapped to 1 are inside L or inside P and points mapped to 2 are inside both L and P[2]. Thus, the problem of computing $L \cup P$ is reduced to the problem of computing a vertex list that represents $f_\cup(Q_{L+P})$, where

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

Similarly, $L \cap P$ is the vertex list that represents $f_\cap(Q_{L+P})$, where

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

The set difference between L and P is obtained by applying $f_\cup$ to $L - P$. Figure 9 illustrates some results obtained by employing this approach.

The algorithm for computing scalar transformations on vertex lists follows the same lines as the ray casting algorithm shown in Section 6.3.1. The idea is to apply the desired transformation to one-dimensional vertex lists extracted from the original list using the prefix and $\mathit{project}$ elementary operations. These are combined first into two-dimensional lists and finally into the resulting three-dimensional list. In the following algorithm, $L3$ is the original three-dimensional vertex list, f is a scalar transformation and $R3$ is the transformed list.

$R3 = \epsilon$

[2]Note that this assertion disregards the problem of boundary points

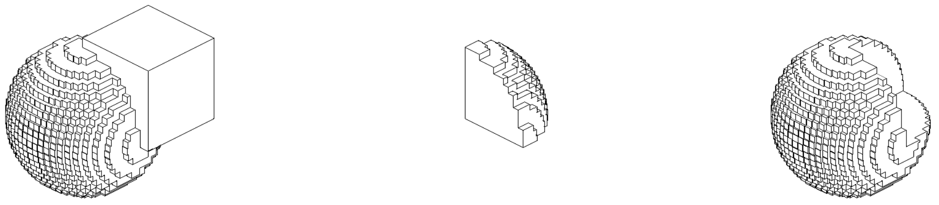

(a) (b) (c)

Figure 9: Set operations performed on vertex list models of a sphere and a cubical block: (a) union, (b) intersection and (c) difference.

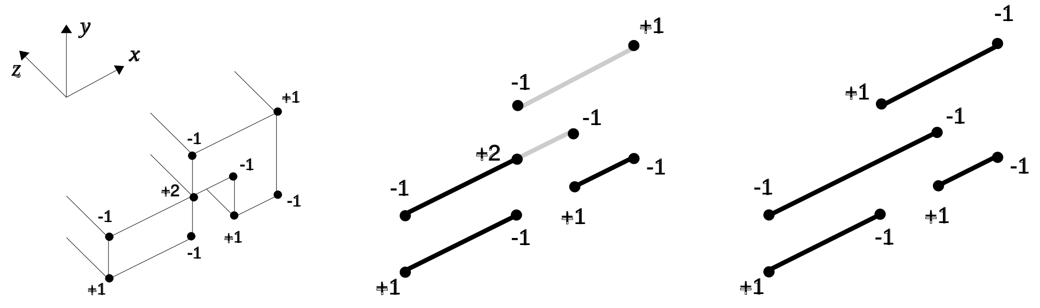
```
$P2 = S2 = \epsilon$
**while** $L3 \neq \epsilon$ **do**
    $z =$last coord. of $head(L3)$
    $L2 = project(prefix(L3)) + P2$
    $P2 = L2$
    $L3 = remainder(L3)$
    $R2 = \epsilon$
    $P1 = S1 = \epsilon$
    **while** $L2 \neq \epsilon$ **do**
        $y =$last coord. of $head(L2)$
        $L1 = project(prefix(L2)) + P1$
        $P1 = L1$
        $L2 = remainder(L2)$
        $R1 =$ **transform1D** $(L1, f)$
        $R2 = append(R2, unproject(R1 - S1, y))$
        $S1 = R1$
    **enddo**
    $R3 = append(R3, unproject(R2 - S2, z))$
    $S2 = R2$
**enddo**
**return R3**
```

In the pseudo-code above, **transform1D** $(L1, f)$ stands for a procedure that returns the one-dimensional vertex list $L1$ transformed by f. This operation is trivial and will not be described here. The algorithm works by disassembling $L3$ into pieces of lower dimension using the inverse process employed in the ray casting algorithm. For instance, while in that algorithm the prefixes of $L3$ were obtained by subtracting values of $L2$ obtained in successive iterations of the outer loop, now it is necessary to add consecutive prefixes of $L3$ to obtain $L2$. A similar process is used in the inner loop to obtain $L1$ from consecutive prefixes of $L2$. Once $L1$ is obtained, **transform1D** is invoked to compute $R1$, its transformed version. The assembly of $R2$ from successive values of $R1$ and of $R3$ from successive values of $R2$ follows the exact same procedure used in the ray casting algorithm. Once again we need temporary variables $P1$, $S1$, $P2$ and $S2$ to hold previous values of $L1$, $R1$, $L2$ and $R2$, respectively, between iterations.

6.3.4 Displaying vertex list models

Modern workstations are equipped with high-performance graphic displays, often supported by hardware that permits very fast rendering of surfaces and lines in three dimensions. Therefore, in order to make use of such capabilities, it is necessary to extract boundary geometric elements from the space occupancy information conveyed by vertex lists. Fortunately, vertex positions already provide the essential geometric clues. For instance, referring to Figure 7(c), we notice that the edges of the model have their endpoints at vertex positions.

A wireframe display of a vertex list model can be obtained by drawing edges parallel to the x, y and z axes. By traversing the list in its original order and drawing line segments between successive vertices we are guaranteed to draw all edges parallel to the x axis. However, some line segments will not correspond to edges of the model. It is clear that two successive vertices cannot form an edge parallel to the x axis if their y or z coordinate values differ. This can be worked around by applying the $prefix/remainder$ operators twice to break the list into several sublists all of which are guaranteed to contain vertices on the same x-parallel line (see Figure 10(a)). Examining one of these sublists we will notice that it represents a scalar field where



(a) (b) (c)

Figure 10: Extracting edges along the x axis.

points are mapped to -1, 0 or $+1$, where edges will correspond only to the regions mapped to -1 or $+1$ (Figure 10(b)). By using scalar transformation $f_{\neq}$ (defined below) it is possible to obtain a list where these regions are all mapped to $+1$ (Figure 10(b)).

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

The advantage of such a list is that it contains an even number of vertices and edges can be traced between two consecutive vertices v_i and v_{i+1} only if $w(v_i) = +1$ and $w(v_{i+1}) = -1$.

The pseudo-code below summarizes the algorithm to trace the edges parallel to the x-axis of a model given by list $L3$. Procedures **edge_endpoint_1** and **edge_endpoint_2** are primitives for specifying the two endpoints of a line segment.

```
while $L3 \neq \epsilon$ do
    $z$ =last coord. of $head(L3)$
    $L2 = project(prefix(L3))$
    $L3 = remainder(L3)$
    while $L2 \neq \epsilon$ do
        $y$ =last coord. of $head(L2)$
        $L1 = project(prefix(L2))$
        $L2 = remainder(L2)$
        $R1$ = transform1D $(L1, f_{\neq})$
        while $R1 \neq \epsilon$ do
            $x$ =last coord. of $head(R1)$
            $w$ =weight of $head(R1)$
            if $w == +1$ then
```

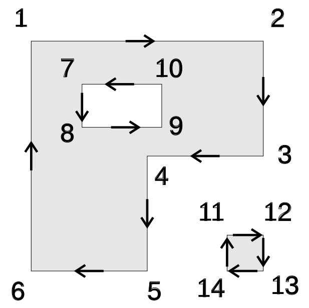
```
                **edge_endpoint_1** $(x, y, z)$
            **else**
                **edge_endpoint_2** $(x, y, z)$
            **endif**
        **enddo**
    **enddo**
**enddo**
```

In order to complete the drawing of the wireframe model it is necessary to trace the remaining edges which are parallel to the y and z axes. This can be done by reordering the list as discussed in Section 6.3.2. This is done twice and each time reordered list is given as input the the algorithm above. Figure 12(a) was produced using this approach.

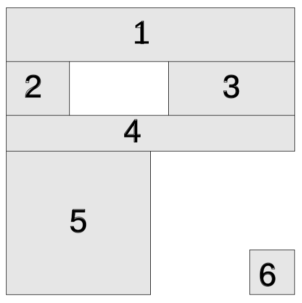
Two-dimensional boundary elements, i.e. faces, can be extracted from the model using a very similar algorithm. In fact, we can modify the algorithm above to extract vertex lists that represent faces parallel to the x-y plane as follows:

```
while $L3 \neq \epsilon$ do
    $z$ =last coord. of $head(L3)$
    $L2 = project(prefix(L3))$
    $L3 = remainder(L3)$
    $R2 =$ transform2D $(L2, f_{\neq})$
    paint_polygon $(R2, z)$
enddo
```

However, the task of drawing a polygon represented by two-dimensional lists (the intended meaning of procedure **paint_polygon** above) is not trivial. The problem is due to the fact that common graphic rendering engines are only able to render simple polygons expressed as a list of connected edges, whereas the two-dimensional vertex list may represent polygons with holes and/or delimited by disconnected contours (see Figure 11(a)). One solution would consist of converting the vertex list polygon into one or more



(a)



(b)

Figure 11: Complex polygon may be rendered by tracing the different contours (a), or by splitting it into a set of rectangles

polygons in the accepted format. This is doable, but unnecessarily messy. A cleaner solution is to split the polygon into a set of rectangles by using a plane-sweep algorithm. In his pioneer work, Shechtman [18] shows how to split a polygon given in vertex list format into a set of maximal horizontal rectangles and proves that the total number of rectangles is never greater than the number of vertices of the input list (see Figure 11(b)). Although simple enough, we will not reproduce his algorithm here.

Once a suitable implementation of procedure **paint_polygon** is achieved, a complete rendering of the model can be produced by applying the above algorithm three times, each followed by a rotation. Figure 12(c) was produced with this approach. Hidden surface elimination was accomplished by depth-buffering. A simple hidden-line rendering can also be achieved by first drawing all faces with depth-buffering enabled and then drawing the edges (Figure 12(b)).

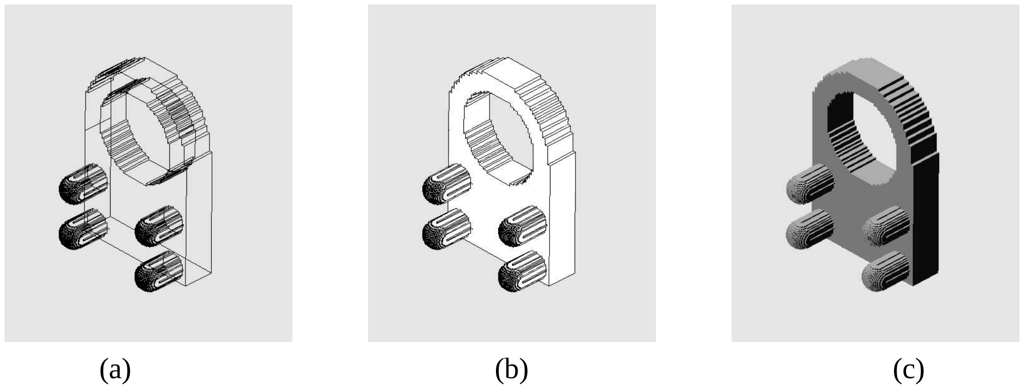


Figure 12: Rendering of a simple mechanical part modeled with vertex lists. Wireframe (a), wireframe with hidden lines (b) and shaded (c).

7 Concluding Remarks

A new approach to modeling and operating on orthogonal objects in arbitrary dimensions known as a vertex representation and implemented as a vertex list has been described. Although the basic idea has already been presented for two-dimensional data [18], our contribution has been the extension to higher dimensions. Its advantage lies in its compactness in comparison with more traditional representations such as rasters and collections of hyperrectangles (disjoint and non-disjoint). Unlike arrays of voxels, the storage required for vertex lists is proportional to the boundary of the represented objects and remains invariant under scaling. Also, unlike CSG, normalized vertex representations have the property of being unique, a property that can be exploited in many ways (e.g., Santos [17] showed how vertex lists are useful for the recognition of circuit elements in VLSI design).

Vertex lists are similar in spirit to chain codes with the advantage that they are not restricted to two dimensions. In particular, the elements of a vertex list can be ordered lexicographically. For example, 3D objects are ordered by faces perpendicular to the z axis. Algorithms have been given for generating vertex representation models for primitive solids, performing affine transformations, set-theoretic operations, and displaying vertex representation models.

The vertex representation models the same objects that are obtained through use of methods such as an array of voxels or octrees (i.e., objects with orthogonal faces). Its advantage is that it requires less space and unlike octrees its space requirements are not sensitive to translation. Moreover, it enables increasing the quality of the approximation of objects by increasing the number of vertices that make up the model. In contrast, increasing the quality of the approximation in a array of voxels requires doubling the resolution which means a significant increase in the space requirements. A drawback of the vertex list implementation of a vertex representation is that it is 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 on the average. These operations can be performed better by imposing access structures such as point variants of quadtrees and octrees (e.g. [15]) on the vertex representation. Conversion of vertex lists to these other forms of representations can be done with little difficulty.

Another issue that springs to mind when considering the power of expression of the vertex representation is whether it is possible to extend it to model non-orthogonal objects. In fact, we have preliminary results indicating that this goal is attainable by allowing vertices which represent different cone types. We expect to report these results in the near future.

8 Acknowledgments

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

References

[1] B. G. Baumgart. A polyhedron representation for computer vision. In *Proceedings of the National Computer Conference 44,* pages 589–596, Anaheim, CA, May 1975.

[2] W. R. Franklin and V. Akman. Building an octree from a set of parallelepipeds. *IEEE Computer Graphics and Applications*, 5(10):58–64, October 1985.

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

[4] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):145–153, April 1979.

[5] D. E. Knuth. *The Art of Computer Programming vol. 3, Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[6] M. Mäntylä. *An Introduction to solid Modeling*. Computer Science Press, Rockville, MD, 1987.

[7] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, June 1982.

[8] D. Meagher. Octree generation, analysis and manipulation. Electrical and Systems Engineering IPL-TR-027, Rensselaer Polytechnic Institute, Troy, NY, 1982.

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

[10] A. A. G. Requicha. Representations of rigid solids: theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, December 1980.

[11] A. A. G. Requicha and H. B. Voelcker. Solid modeling: a historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, 2(2):9–24, March 1982.

[12] S. D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144, February 1982.

[13] 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.

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

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

[16] H. Samet and M. Tamminen. Bintrees, CSG trees, and time. *Computer Graphics*, 19(3):121–130, July 1985. (also *Proceedings of the SIGGRAPH'85 Conference*, San Francisco, July 1985).

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

[18] J. Shechtman. Processamento geométrico de máscaras VLSI. Master's thesis, Eng. Elétrica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, April 1991.