Data structures to support Bézier-based modelling

Hanan Samet and Robert E Webber*

Data structure issues that arise in Bézier-based modelling are surveyed. Both low and high level implementation details are discussed. The low level issues deal with the interaction between the representation of the individual patches and the representation of the collection of patches. The high level discussion consists of two parts. The first deals with how to maintain the topological integrity of the objects being modelled. This includes an outline of a technique for adapting Euler operators to Bézier primitives. The second is concerned with increasing the efficiency of certain operations by using hierarchical spatial data structures. Such representations facilitate the execution of operations that depend on spatial proximity (i.e., which patch is closest to another patch).

Bézier primitives, surfaces, computer-aided design, data structures, quadtrees, octrees, Euler operators

Many applications in computer-aided design require the manipulation of objects whose surfaces are arbitrarily curved rather than being flat (i.e., planar). Examples of such applications range from design packages in the automobile industry[1,55] to font design tools in desktop publishing[2]. In fact, it is not uncommon for the user to be unable to give a precise mathematical formula for the surface. Instead, the user would like to resort to an intuitive (e.g., interactive) specification of the surface in the sense that aesthetic feedback from the designer plays an important role in the final form of the surface. Ideally, the user would like to have a system where a surface can be varied by tweaking a few knobs rather than defining new equations.

In general, it is difficult to draw the desired surface. This has led to the development of techniques that enable the surface to be specified by the coordinates of a few points in space. These points are known as control points. By varying the locations of these control points, the surface can be defined and then visualized. One such set of control points is said to comprise a Bézier primitive[3,5,55]. The Bézier primitives present the designer with a collection of control points that define a curved object. These control points form an exaggerated outline of the curved object. By moving the control points, designers can interactively construct the desired curved object. The algebraic basis of the Bézier primitives is the Bernstein polynomials[1,5] commonly used by numerical analysts. Bézier primitives can also be derived from a purely geometric construction due to de Casteljau[6,7].

An even more complex problem is that frequently it is not possible to define the entire surface by a single concise mathematical formula. Thus the surface is often viewed as consisting of a collection of surfaces (termed patches) which may be required to satisfy continuity, differentiability, and curvature constraints at the points and curves at which they are adjacent. The efficient representation of collections of the patches based on Bézier primitives is the subject of this paper.

The mathematics of Bézier primitives has a number of useful properties. One such property is the convex hull property. It states that the object defined by a set of control points of a Bézier primitive will lie within the convex hull of those control points. Bézier primitives are usually based on a triangular or a rectangular network of control points. For Bézier primitives based on a triangular network of control points, the following property also holds. If the network of control points is a convex surface, then the curved surface generated by the control points will also be convex[8,9,10].

Another important property that holds for both a triangular and rectangular network of control points is that a Bézier primitive can be subdivided into a collection of smaller Bézier primitives that collectively define the same surface as the original Bézier primitive. Using this property, in conjunction with the convex hull property, enables us to obtain progressively tighter approximating bounds (i.e., collections of convex hulls) within which the surface must lie. Indeed, these approximations eventually converge to the original surface.

The above properties can be combined to form the basis of a generic surface intersection procedure[11,12]. Such a procedure is useful for displaying the objects being modelled, checking that a collection of Bézier primitives forms a valid boundary of an object, and locating the position of a Bézier primitive with respect to a linear volume element (i.e., whether or not it intersects the element) as is common when using some form of spatial subdivision (e.g., octrees as discussed

Computer Science Department and Institute of Advanced Computer Studies and Center for Automation Research, University of Maryland, College Park, MD 20742, USA
*Department of Computer Science, The University of Western Ontario, London, Ontario, Canada N6A 5B9
Paper received: 15 August 1990. Revised: 1 November 1990

below). For example, the following five observations serve as the control structure of a simple recursive algorithm for detecting whether or not a particular Bézier primitive, say B, intersects a given cube, say C:

- If the control points of B are inside C, then B is inside C
- If all the control points of B are outside C and on the other side of the same face (e.g., above the top face of the cube), then B is outside C
- If B can be decomposed into subprimitives that are all inside C, then B is inside C
- If B can be decomposed into subprimitives that are all outside C, then B is outside C
- If B can be decomposed into subprimitives, some of which are inside C and some of which are outside C, then B is both inside and outside C

An algorithm that has this control structure explores the possibility of intersection between cube C and Bézier primitive B by recursively subdividing those subprimitives that could not be identified as being completely inside C or completely outside C until they could be so classified.

As the number of Bézier primitives increases, we must pay more attention to data structure issues. They are influenced by the type of operations that we wish to perform. There are three major issues which are most logically considered in the following order, as it is indeed the order in which they arise. The first is the interaction between the representation of the individual patches and the representation of the collection of patches. The remaining two issues are somewhat complementary. They address the question of whether the representation is to support operations based on the topology of the patches (i.e., which patch is connected to which patch) or operations based on spatial proximity (i.e., which patch is closest to another patch).

The first issue is the representation of the primitives as a set – i.e., to support the operations of insertion, deletion, and enumeration. As the number of primitives increases, it is often the case that the operations only need to manipulate a subset of the primitives. For example, as the number of primitives increases, the proportion of the primitives that are visible from a particular viewpoint usually decreases. Thus it becomes increasingly wasteful to have to access every primitive for each display operation.

One obvious optimization of the display operation is to ignore primitives that appear on the back sides of objects (e.g.,[11]). However, this requires that we be able to determine which primitives are parts of which objects, as well as their orientation with respect to the objects. This determination requires that we construct a connectivity graph of primitives that are neighbours by virtue of sharing a vertex or edge. We must also keep track of orientation information in the connectivity graph. In addition, such a graph is useful for investigating whether or not a collection of primitives define a valid boundary of an object. Moreover, this graph is a good heuristic for grouping patches that are neighbours by virtue of spatial proximity. This heuristic is motivated by the tendency for primitives that are very close to other primitives to be directly connected to them.

Nevertheless, as the number of primitives becomes very large, it is useful to organize them directly by their spatial location. This, of course, improves the efficiency of algorithms that are spatially motivated (e.g., determining the patch at which a user is pointing). However, for a very large collection of primitives, it also improves the efficiency of algorithms that manipulate the connectivity graph of primitives. This makes sense because spatial proximity is a good heuristic for grouping patches that are connected.

Spatial proximity is important because the performance of connectivity graph algorithms depend on the accessibility of the actual memory locations at which the relevant information is stored. It is well known that most computer memories are organized in a linear manner – i.e., once location i has been accessed, it is much easier to access location $i+1$ than arbitrary location j. Moreover, it is easier to devise a technique for embedding a higher-dimensional space (e.g., a three-dimensional array) in a linear space than a general graph. Thus for the purpose of placing the data in memory, it is preferable to embed a connectivity graph in a spatial organization rather than a general graph structure.

The rest of this paper is organized as follows. The next section presents techniques for representing the collection of Bézier primitives as a set, as well as the individual primitives. The third section discusses techniques for representing the topological aspects of a collection of Bézier primitives. This discussion includes material relevant to confirming that a collection of primitives defines a topologically meaningful object. The fourth section describes some techniques for organizing the Bézier primitives by their spatial properties.

REPRESENTING SETS OF BÉZIER PRIMITIVES

In this section, we briefly discuss some alternative issues in the representation of the individual Bézier primitives, followed by a more detailed examination of the representation of the collection of the primitives. One such issue is the question of whether the Bézier primitives are curves, surfaces, or solids, and whether they exist in the plane or in 3D space. Collections of 1D Bézier primitives in the plane tend to be easier to represent than 2D Bézier primitives (i.e., surfaces) in a 3D space. In this paper, the focus is on 2D Bézier primitives in a 3D space. Another issue is whether or not all of the Bézier primitives in a collection will have the same number of control points. In this paper, we do not address the additional complications that arise when using different types of Bézier primitives, thereby requiring variable amounts of storage per primitive.

There is also the issue of whether the Bézier primitive (i.e., patch) is based on a triangular or a rectangular grid of control points. A triangular patch (e.g., Figure 1(a)) will have at most three neighbours, where two patches are said to be neighbours if they share an edge. However, a rectangular patch (e.g., Figure 1(b)) will

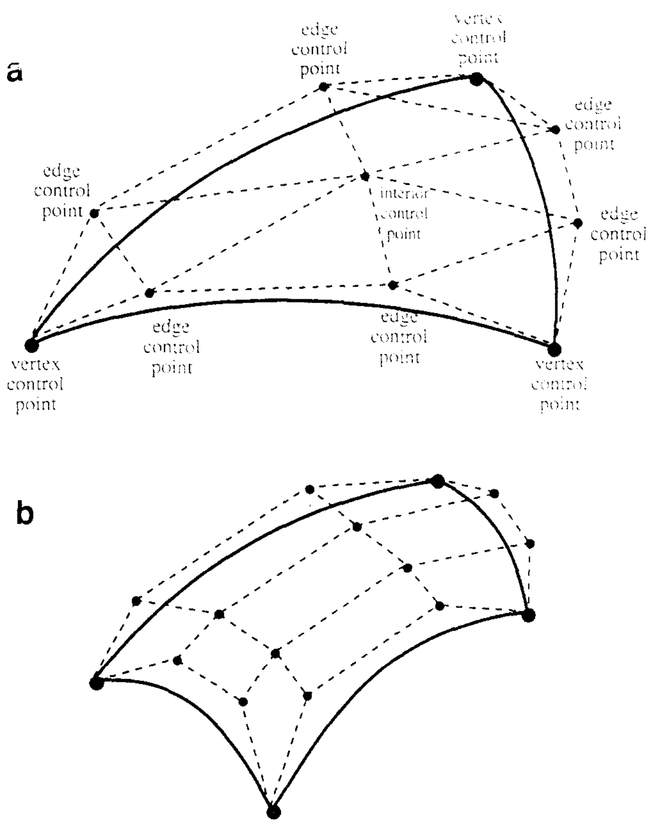


Figure 1. (a) Triangular Bézier patch; (b) rectangular Bézier patch

have at most four neighbours with which it shares an edge. It is often easier to decompose a surface into a tessellation of triangles than one of rectangles (e.g., consider a globe and the problems that arise at the poles in a Mercator projection). Another benefit of triangular patches is that they have fewer control points, which means that the evaluation of the primitive is faster. In this paper, we use triangular Bézier primitives (specifically a cubic primitive requiring 10 control points). However, the data structure issues that we discuss are equally applicable to rectangular patches.

In the following discussion, it is convenient to distinguish among three classes of control points within a triangular Bézier primitive (as labelled in Figure 1(a)). The three control points that are at the corners of the triangular control network are called vertex control points. A useful property of the Bézier primitive is that it actually passes through the vertex control points. In our example cubic triangular Bézier patch, there are two control points along the edge of the triangular network between each pair of vertex control points. These are called edge control points. The four control points along an edge, say E, of the triangular network (i.e., a vertex control point, two edge control points, and the other vertex control point on E) are the control points for a cubic Bézier curve that forms the boundary of the cubic triangular patch on E. In general, the patch does not pass through the edge control points. However, in order for two neighbouring patches to be properly aligned, they will share both the vertex control points and the edge control points on the appropriate edge. The final class of control points consists of the interior control point, say I, and called an interior control point. Usually, the patch does not pass through I, and I is not shared with any of its neighbouring patches.

An individual Bézier primitive consists of an ordered collection of control points. A typical Bézier primitive is a triangular cubic patch, determined by ten control ponts in 3D space, i.e., by 30 numbers (either fixed-point or floating-point numeric representations could be used). Assuming a 4-byte number, a single Bézier primitive requires 120 bytes of storage. Editing a Bézier primitive consists of altering the value of some subset of these 30 numbers. Whenever we need to determine the location of the actual surface defined by the patch, the control points of the patch are usually accessed in a row-by-row manner. Thus a standard embedding of a triangular array into a linear array is algorithmically efficient.

One way to represent a collection of Bézier primitives is as an array of records where each record corresponds to a single Bézier primitive. At any instance of time, not all the records of the array are in use. A simple way to organize the array would be to use a counter N to indicate how many primitives are in use. To add a new primitive P, N is incremented, and P is copied into the N^{th} element of the array. To delete the k^{th} primitive, the N^{th} primitive in the array is copied to the k^{th} entry in the array, and N is decremented.

One problem with this approach is determining the maximum size of the array. Instead of using a 120 megabyte array (thereby permitting the storage of a million cubic triangular patches), we can use a paging approach. In this case, we can make use of an array of 1000 pointers to serve as a page table (which would consume 4000 bytes of storage). Each pointer would either point at a page (i.e., another array) of 1024 Bézier primitives (120 kbytes) or be NIL (indicating that no primitives reside on that page). Finding the k^{th} entry in the set of primitives is easy. First, find the $\lfloor k/1024\rfloor^{th}$ pointer, say P, and then access the $(k - \lfloor k/1024\rfloor * 1024)^{th}$ entry in the array pointed at by P.

A second problem with the array approach is that it ignores the expected redundancy in a control network of triangular Bézier primitives. For such a primitive, the six edge control points will most likely be shared with one other primitive, the three vertex control points will most likely be shared with two or more other primitives, and the one internal control point is most likely to be unique to that primitive. Thus, for our example of one million primitives, the number of unique control points is likely to be less than five million, rather than the 10 million that are stored in the array approach.

The above approach can be implemented by storing the five million 'unique' control points as an array of five million records of three numbers each (requiring 12 bytes apiece) thereby consuming 60 megabytes of storage. We also need to store the actual primitives. They can be stored as an array of one million records of 10 indices apiece (requiring 4 bytes per index) into the array of control points, thereby consuming an additional 40 million bytes. Such a representation would store the entire set of primitives in 100 megabytes rather than the 120 megabytes required in the original array approach. This technique is similar to that used by

Baumgart[56] in the winged-edge data structure, a basis of the boundary model (BRep)[14] representation of planar-faced solids, where the basic data item is an edge rather than a control point. The paging approach outlined earlier could be used here as well for managing both the array of control points and the array of Bézier primitives.

A further refinement would be to separate the array of control points into three different types of arrays: one for the vertex control points, one for the pairs of edge control points, and one for the isolated internal control point. Thus, each record that corresponds to a cubic triangular Bézier primitive would require seven indices (three vertex indices, three edge indices, and one internal index) rather than 10 indices. This implementation means that each record requires 28 bytes instead of 40 bytes. Thus one million primitives would consume 28 megabytes. Remembering that we need 60 megabytes for just storing the control points means that the entire set of primitives requires 88 megabytes instead of 120 megabytes.

As a final refinement of the cubic traingular Bézier primitive, the index for the internal control point could be replaced by the actual value of the coordinates at that point. Since there is no expected sharing of this point, the net effect of this optimization would be the elimination of one million indices (4 megabytes). The result is that only 84 megabytes of storage are needed for the set of one million primitives. It is interesting to observe that the net gain as a result of sharing the storage for the control points in a manner analogous to the sharing of edges in the winged-edge data structure is not very large.

REPRESENTATION OF THE TOPOLOGY OF BÉZIER PRIMITIVES

One problem that arises when working with a collection of geometric primitives (regardless of whether they are Bézier primitives) is that of ensuring that the primitives define a valid object in the space being modelled. For example, when designing an aeroplane, a Möbius strip (see Figure 2) is generally not considered a meaningful part. Ensuring that such objects do not arise depends on the representation being used. For example in the case of Constructive Solid Geometry (CSG)[15] this is achieved by using 'regularized' Boolean operations, while for boundary models (BRep)[14] this is achieved by using Euler operators[56] (when augmented by suitable intersection checking which is a form of regularization – see the discussion of inter-object intersection at the end of this section).

An example of the CSG approach would be to use half planes as primitives and then form objects as a Boolean combination of half planes. Figure 3 illustrates

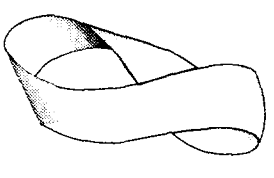
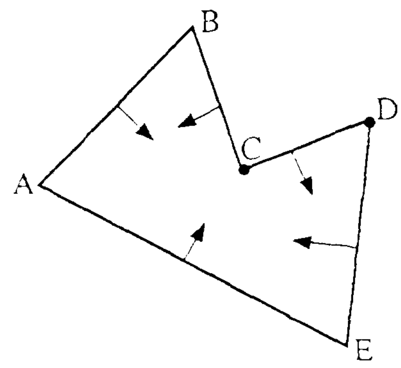


Figure 2. Möbius strip



$$(AB \cap BC \cap EA) \cup (CD \cap DE \cap EA)$$

Figure 3. Example of construction of CSG representation of pentagon

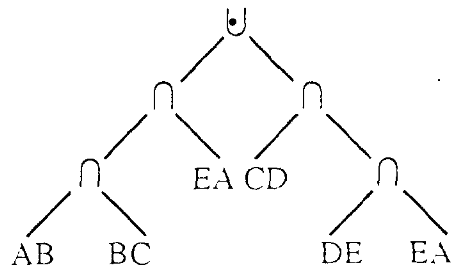
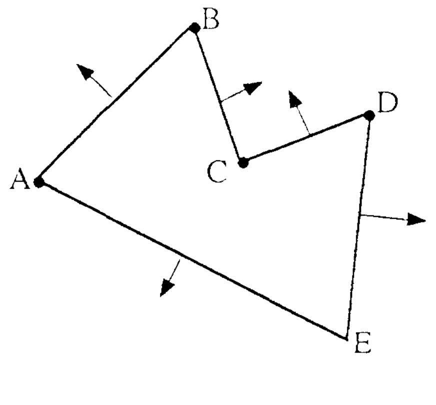


Figure 4. Data structure corresponding to Figure 3

a pentagon defined as a Boolean combination of half planes in 2D space. The corresponding data structure (Figure 4) is a tree where the internal nodes correspond to the operators in the Boolean expression and the leaf nodes correspond to the half plane primitives. For CSG, ensuring that only valid objects result from a Boolean expression is equivalent to ensuring that every intersection is either empty or creates an object with the same dimensionality as the operands to the intersection operation (e.g., the intersection of two half planes is not permitted to result in a line). This is achieved by modifying the definition of the Boolean operators to be regularized Boolean operators[15], where these details are taken into account.

This list of line segments in Figure 5 is an example of the boundary model for the pentagon of Figure 3. Notice that the line segments indicate their constituent vertices as well as the orientation of the line with respect to the interior of the object. The orientation information is important because a boundary defines two objects, one on the inside of the boundary and one on the outside of the boundary. In order to determine the object that has been defined, we need to know which direction is to be viewed as 'inside.' For the boundary model, ensuring that only valid objects are created becomes the problem of ensuring that there is always a consistent labelling of inside and outside for the primitives that make up the objects being modelled. This is guaranteed by using a collection of operators called the Euler operators[16,56]. Thus an object like that



$\overline{AB}$, $\overline{BC}$, $\overline{CD}$, $\overline{DE}$, $\overline{EA}$

Figure 5. Boundary model representation corresponding to Figure 3

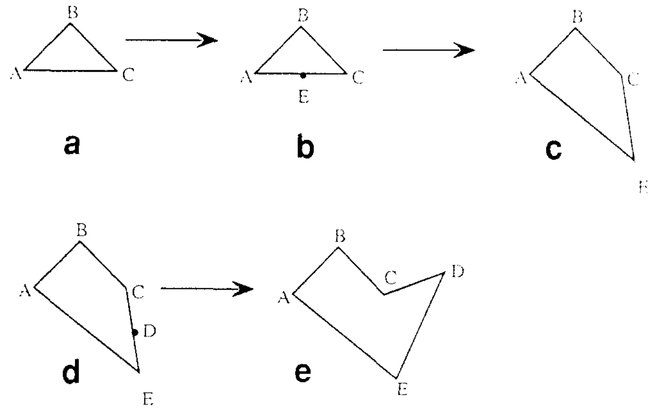


Figure 6. Euler operator sequence leading to construction of Figure 5

in Figure 5 would be built by the sequence shown in Figure 6.

Let us look more closely at how Figure 6 is constructed. We will use the MakeTriangle, MakeVertex, and MoveVertex operators. The first step is to generate the smallest possible valid polygon (e.g., MakeTriangle yielding Figure 6(a)). The next step is to add a new vertex to the polygon using a MakeVertex operator (e.g., Figure 6(b)). MakeVertex creates a new vertex as well as updates the edge database by removing edge CA and adding edges CE and EA. At this point, the location of vertex E is moved by applying the MoveVertex operator (e.g., Figure 6(c)). MoveVertex checks to make sure that moving the vertex does not cause any of the line segments to intersect at any points that are not vertices. MoveVertex also checks the consistency of the orientation labelling. Next, insert point D using the MakeVertex operator (Figure 6(d)), and move it to its correct location using a MoveVertex operator (Figure 6(e)).

Evaluating intersections of Bézier primitives in a manner analogous to that used in CSG can be very expensive. The problem is that we must compute the intersections of 3D objects, which would require that we work with cubic tetrahedral Bézier primitives. In particular, the number of control points in each cubic tetrahedral Bézier primitive is twice (i.e., 20 instead of 10) the number of control points in each cubic triangular Bézier primitive, which is the primitive used for the analogous boundary model.

When comparing the boundary model with CSG we observe that the dimensionality of the Bézier primitives comprising the boundary model is one less than that of the analogous Bézier primitives comprising CSG. This is because the dimensionality of the border is one less than that of the object that is being bounded. Therefore, there is a tendency to use the boundary model to reduce the dimensionality of the Bézier primitive and thus minimize its complexity.

Now, let us see how we can adapt the concept of a Euler operator to the Bézier primitive. The traditional development of Euler operators is in terms of the creation and destruction of vertices, edges, and faces. A typical operator is one that changes an edge into two edges while adding a vertex (recall the MakeVertex operator illustrated in Figure 6). For example, applying the MakeVertex operator to a triangular face would create a quadrilateral face. However, in a system of triangular Bézier primitives, a quadrilateral face is meaningless. Thus, when adding a vertex, say V, to an edge, say AB, of a triangular Bézier primitive ABC in a system of triangular Bézier primitives, it is not sufficient for edge AB to be split into two new edges, AV and BV. Instead, a third edge VC is added connecting vertex V to the vertex C of triangular Bézier primitive ABC (which is opposite edge AB). This results in the split of the triangular Bézier primitive ABC into two new triangular Bézier primitives AVC and BVC (see Figure 7). Furthermore, if edge AB happened to be shared with a neighbouring triangular Bézier primitive, say ABD, then a fourth edge VD would have to be added, thereby splitting neighbouring triangular Bézier primitive ABD into two new triangular Bézier primitives AVD and BVD as well.

We propose the following six operators as a minimal set of Euler operators appropriate for maintaining a system of triangular Bézier primitives. These operators are named MakeTetrahedron, DeleteTetrahedron, MakeVertexOnEdge, RemoveVertexOnEdge, Merge-Triangles, and CutTriangle. Figure 8 illustrates the functionality of each of these operators.

MakeTetrahedron defines a minimal solid object bounded by four triangular Bézier primitives. The Bézier primitives are initialized to correspond to equilateral planar triangles. The size and orientation of the tetrahedron are parameters of the MakeTetrahedron

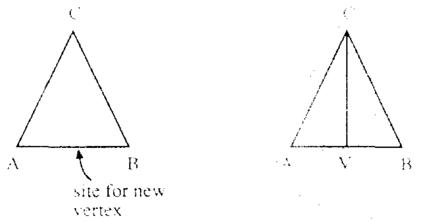


Figure 7. Example of application of MakeVertex operator to triangular Bézier primitive

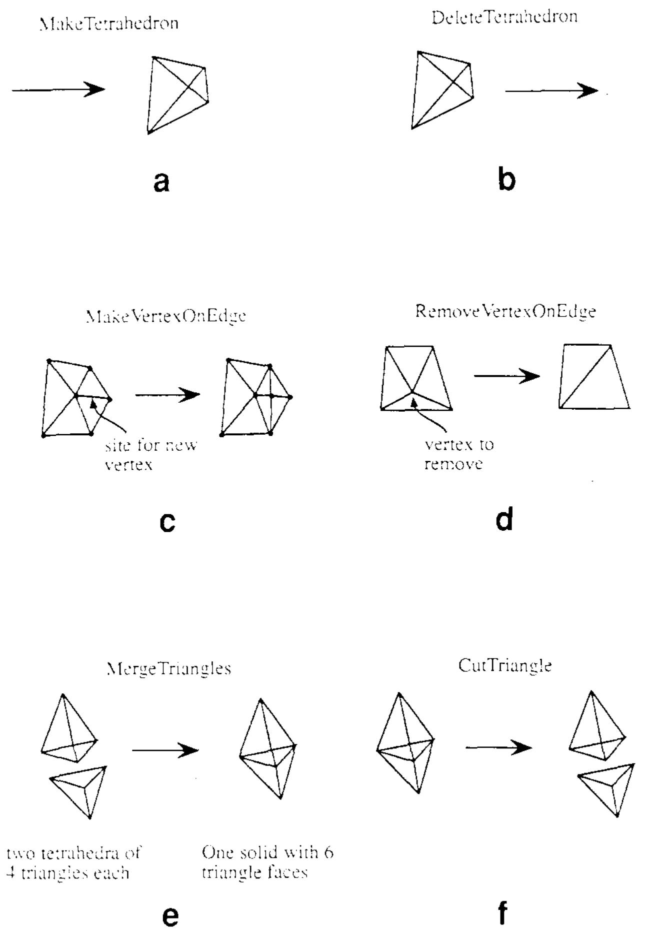


Figure 8. Euler operators for maintaining system of triangular Bézier primitives

operator. When a tetrahedron is created within an already existing object, the new tetrahedron determines a hole within the object. DeleteTetrahedron removes the four Bézier primitives of the tetrahedron from the collection of Bézier primitives.

The MakeVertexOnEdge operator takes as parameters an edge, say AB, and the two triangular Bézier primitives ABC and ABD that share the edge AB, and a point, say V, on edge AB. V becomes a vertex control point shared by the four triangular Bézier primitives AVC, BVC, AVD, and BVD. The remaining control points for these four triangular Bézier primitives are computed using the Bézier subdivision algorithm.[5]

The RemoveVertexOnEdge operator is the inverse of MakeVertexOnEdge. It can only be applied to a vertex, say V, that is the common corner of four triangular Bézier primitives AVC, BVC, AVD, and BVD that form the quadrilateral ABCD. When RemoveVertexOnEdge is applied, the four edges that meet at vertex V are removed (i.e., AV, BV, CV, and DV), and are replaced by a new edge (i.e., AB or CD) specified as a parameter to the function. Note that two new internal control points R and S need also to be specified so that triangular Bézier primitives ABC and ABD will be valid Bézier primitives.

MergeTriangles takes two triangular Bézier primitives as parameters, say ABC and DEF. It removes both triangular Bézier primitives ABC and DEF from the collection of Bézier primitives. All references to vertices D, E, and F in the collection of Bézier primitives are changed to references to A, B, and C, respectively. This operator can be used for forming handles in an object or for merging two solid objects into one solid object.

CutTriangle is the inverse of the MergeTriangles operator. Its parameter is a sequence of three distinct edges, say AB, BC, and CA, from the collection of Bézier primitives, such that there exists no triangular Bézier primitive ABC. The edges that share exactly one vertex with the non-existent triangle ABC are partitioned into two subsets, say L and R, depending on the side of ABC on which they lie. Next, triangular Bézier primitive ABC is added to the collection of Bézier primitives (thereby requiring the user to specify an internal control point for it, say V). The user must also specify a second Bézier triangular primitive DEF (and a control point for it), which is also added to the collection of Bézier primitives. All references to vertices A, B, and C in edges of the subset R are replaced by references to the vertices D, E, and F, respectively. On the other hand, note that all references to vertices A, B, and C in edges of the subset L remain unchanged. This operator has the effect of cutting a solid into two parts (if the solid has no holes) or, depending on the location of the cut, reducing the number of holes in a solid by one.

Implementing the operators described above requires that we be able to query the collection of Bézier primitives efficiently. In particular, we need to be able to determine (i.e., query) which triangular Bézier primitives share an edge with a given triangular Bézier primitive. There are a number of ways this can be achieved. One approach is to add three index fields to the record which represents each Bézier primitive. For each triangular Bézier primitive ABC, the index fields point at the Bézier primitives adjacent to the constituent edges AB, BC, and CA, respectively. This would add 12 bytes to the size of each record that corresponds to a Bézier primitive. Using the final representation disussed above means that the amount of storage necessary to represent one million Bézier primitives increases from 84 megabytes to 96 megabytes.

An alternative representation is to associate the adjacency information with the edge records rather than with the records of the Bézier primitives. This is similar to the winged-edge representation of Baumgart[56]. In this case, for each edge we store the indices of the two adjacent Bézier primitives. This would increase the size of the edge record by 8 bytes. Since there are three edges for each triangular Bézier primitive, and all edges are shared by two triangular Bézier primitives, the amount of storage required for this approach would be the same as for the previous approach that added three index fields for each Bézier primitive.

Another implementation decision is determining which side of each Bézier primitive (i.e., face) is 'outward'. This is achieved by adopting the convention that the indices to the edges and vertices in a Bézier primitive appear in a clockwise order. In this way, when we compute the clockwise crossproduct of the derivative of two curved borders of the Bézier primitive, evaluated at the curved borders' common vertex, we get a vector pointing in the 'outward' direction.

As with the traditional Euler operators, it is necessary to check for inter-object intersection in order to ensure that a valid solid is being created. The Euler operators by themselves only guarantee the topological validity of the objects created. Additional checking is necessary to make sure that objects have not been specified that have boundary patches that coincide (thus creating zero-width regions). The only points that are permitted to belong to more than one Bézier primitive are those that are vertex points or points on the boundary curve defined by the edge control points. Doing this efficiently requires the use of spatial data structures as discussed in the fourth section. There is also the issue of whether an individual Bézier primitive is self-intersecting, has holes, etc. Because there is a continuous mapping between a triangle and the triangular Bézier primitive (defined by barycentric coordinates and the Bernstein polynomials[5]) the triangular Bézier primitive will not have holes or handles. Indeed, it can also be observed that the image of the boundary of the triangle will form the boundary of the triangular Bézier primitive. Thus we need not worry about holes appearing inside of a triangular Bézier primitive, provided that the border of the primitive does not self-intersect. However, it is possible for the mapping to cause a Bézier primitive to intersect itself.

In the case of triangular Bézier primitives, there are a few known results relevant to the self-intersection problem[8,9,10]. The most simple is a theorem stating that the surface defined by a convex network of control points is also convex. Since a convex shape does not self-intersect, Bézier primitives with convex control networks are 'guaranteed' to not self-intersect. Thus if we restrict ourselves to triangular Bézier primitives with convex control networks we need not worry about the self-intersection of primitives. If we further constrained the Bézier primitives so that the four control points on each edge lie in a plane, thereby ensuring that each edge of the Bézier primitive was planar, then we would have an example of one of the proposed splinehedra data structures[17].

If triangular Bézier primitives with non-convex control networks are to be used, then it becomes necessary to check the primitive, say P, directly for self-intersection. This can be achieved by subdividing P into convex subprimitives and then testing for self-intersection of this network of convex subprimitives. At worst, the Bézier primitive might need to be subdivided to a level at which each primitive is planar. At this point, the problem reduces to testing for self-intersection of a polygonal network.

REPRESENTING GEOMETRY OF BÉZIER PRIMITIVES

For sufficiently small collections of Bézier primitives, the methods of the previous section are sufficient. However, as the number of primitives gets large, it has been observed that individual updates to the collection tend to involve only a small number of primitives that are spatially near one another[18]. For example, when connecting two previously disjoint objects by performing a MergeTriangles operation, it is necessary to determine if the merger caused intersection of the border of the two objects anywhere other than at the location of 'triangular' region where the two objects are being merged. Performing this determination using the data structures discussed in the third section (i.e., CSG and BRep) requires that we compare every patch in the first object with every patch in the second object. However, if the primitives were organized so that patches that were close to each other in the model were close to each other in the graph of the data structure, then the number of comparison operations could be reduced substantially. Thus, by using a spatial data structure that takes into account the location of the primitives (i.e., their geometry) as opposed to merely keeping track of the interconnectivity of the primitives (i.e. their topology), large collections of Bézier primitives can be efficiently updated and displayed.

In order to develop a spatial data structure to organize Bézier primitives, it is worthwhile to consider the procedure by which primitives are tested for intersection. The most straightforward approach is to treat the locus of points comprising the Bézier primitive as being the actual data that is to be stored in the data structure. This approach is the subject of the next section. In the two-dimensional case, the Bézier primitive (i.e., the patch) is a complicated curved surface and determining the intersection points involves the solution of nontrivial polynomial equations.

On the other hand, the convex hull property guarantees that, for example, if a line fails to intersect the polyhedron defined by the convex hull of the control points of the Bézier primitive, then that line will also fail to intersect the primitive itself. Thus, an alternative view would be to build a spatial data structure for a collection of possibly overlapping convex polyhedra and to perform the necessary spatial searching in that data structure. However, since determining the convex hull of 10 points is computationally expensive, there is a tendency for implementations[19,20] to use the rectilinear bounding box of the control points to spatially bound a Bézier primitive rather than the convex hull. Thus spatial data structures relevant to Bézier primitives focus on storing overlapping rectilinear boxes rather than overlapping polyhedra. The next section discusses data structures for spatially organizing collections of rectilinear boxes (i.e., by sorting them), hierarchical data structures[21,22] being used to perform the necessary spatial organization.

In the case of biquadratic Bézier primitives, Brunet and Ayala[23] propose a non-rectilinear polyhedron that tends to yield a tighter approximation than a rectilinear bounding box but a looser one than the convex hull. The rationale for using such a polyhedron is the observation that the edges of a biquadratic primitive are planar. Brunet and Ayala observe further that for any given edge E of a primitive P, the remainder of P will tend to lie on the same side of the plane in which E lies (i.e., primitive P will tend not to lie on both sides of that plane). For more details about this technique interested readers should consult the original paper[23].

The hierarchical data structures discussed below are based on the principle of recursive decomposition (similar to divide and conquer methods)[24]. These data structures are useful because of their ability to focus on the interesting subsets of the data. This focusing results in an efficient representation and in improved execution times. Thus, they are particularly useful for performing search and set operations. Nevertheless, it is true that many of the operations for which they are used can often be performed as efficiently, or more so, with other data structures. However, hierarchical data structures are attractive because of their conceptual clarity and ease of implementation. They serve primarily as devices to sort data of more than one dimension and different spatial types, and hence their use provides a spatial index. The term quadtree is often used to describe this class of data structures. This particular approach to hierarchical spatial data structures is the focus of the next two sections. For a more extensive treatment of heirarchical spatial data structures, see[21,22].

Spatially organizing the network of surfaces

There are two ways of representing a region. The first is by its interior and the second is by its boundary. We first describe the region quadtree[25,57] which represents a region by its interior. Assume that the data is binary and consists of an image array of picture elements (termed pixels), where 1s correspond to interior points and 0s correspond to exterior points. The region quadtree is built by successively subdividing the image array into four equal-size quadrants. If the array does not consist entirely of 1s or entirely of 0s (i.e., the region does not cover the entire array), it is then subdivided into quadrants, subquadrants, etc., until blocks are obtained (possibly single pixels) that consist entirely of 1s or entirely of 0s. As an example of the region quadtree, consider the region shown in Figure 9a which is represented by the $2^3 \times 2^3$ binary array in Figure 9(b). The resulting blocks for the array of Figure 9(b) are shown in Figure 9(c). This process is represented by a tree of degree 4 (Figure 9(d)).

In the tree representation, the root node corresponds to the entire array. Each son of a node represents a quadrant (labelled in order NW, NE, SW, SE) of the region represented by that node. The leaf nodes of the tree correspond to those blocks for which no further subdivision is necessary. A leaf node is said to be black or white, depending on whether its corresponding block is entirely inside or entirely outside of the represented region. All non-leaf nodes are said to be grey. The quadtree representation for Figure 9(c) is shown in Figure 9(d). Of course, region quadtrees can also be used to represent non-binary images. In this case, we apply the same merging criteria to each colour.

Quadtree-like data structures can also be used to represent images in three dimensions and higher. The region octree[25-27,58] data structure is the 3D analogue of the region quadtree. It is constructed in the following manner. We start with an image in the form of a cubical volume and recursively subdivide it into eight congruent disjoint cubes (called octants) until blocks are obtained of a uniform colour or a predetermined level of

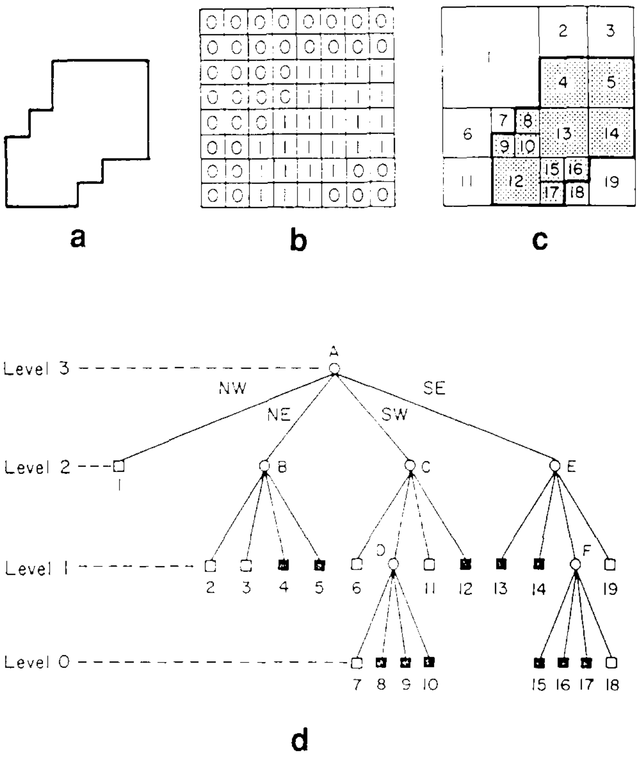


Figure 9. Region, its binary array, its maximal blocks, and corresponding quadtree. (a) Region; (b) binary array; (c) block decomposition of region in (a)—blocks in region are shaded; (d) quadtree representation of blocks in (c)

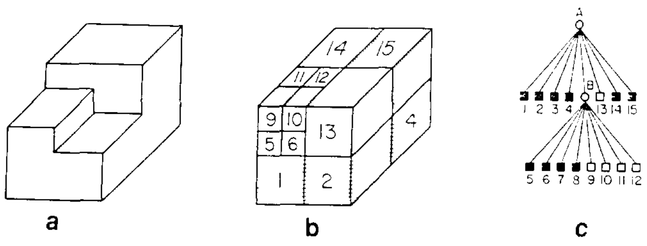


Figure 10. (a) Example three-dimensional object; (b) its octree block decomposition; and (c) its tree representation

decomposition is reached. Figure 10(a) is an example of a simple 3D object whose raster octree block decomposition is given in Figure 10(b) and whose tree representation is given in Figure 10(c).

A key to the analysis of the execution time of quadtree algorithms is the quadtree complexity theorem,[25,28] which states that the size of the quadtree representation of a region (i.e., the number of blocks) is linear in the perimeter of the region. This result also holds for 3D data[29] where perimeter is replaced by surface area, as well as higher dimensions, say d, for which it is cast in terms of the $(d-1)$-dimensional interfaces between the d-dimensional objects being represented. The quadtree complexity theorem also directly impacts the analysis of the execution time of algorithms. In particular, most algorithms that execute on a quadtree (or octree) representation of an image instead of an array representation have an execution time that is proportional to the number of blocks in the image rather than the number of array elements. In its most general case, this means that the application of a quadtree algorithm to a problem in d-dimensional space executes in time proportional to the analogous array-based algorithm in the $(d-1)$-dimensional space of the surface of the original d-dimensional image. Therefore, quadtrees (and octrees) act like dimension-reducing devices.

One of the deficiencies of the region octree is that if the faces of the object (or objects) represented by it are not rectilinear, then the representation is not exact in the sense that it is an approximation. The only exception is if the faces are mutually orthogonal, in which case a suitable rotation operation can be applied to yield rectilinear faces. We prefer an exact representation. In the following, we describe the PM octree[30–36], an approach that is based on the interior and the boundary of the region. We first discuss its use for objects with planar faces. Next, we show how it can be used for objects whose faces are described by more general methods such as Bézier primitives.

In the PM octree, the resulting decomposition ensures that each octree leaf node corresponds to a single vertex, a single edge, or a single face. The only exceptions are that a leaf node may contain more than one edge if all the edges are incident at the same vertex. Similarly, a leaf node may contain more than one face if all the faces are incident at the same vertex or edge. The above subdivision criteria can be stated more formally as follows:

- At most, one vertex can lie in a region represented by an octree leaf node.
- If an octree leaf node's region contains a vertex, then it can contain no edge or face that is not incident at that vertex.
- If an octree leaf node's region contains no vertices, then it can contain at most one edge.
- If an octree leaf node's region contains no vertices and contains one edge, then it can contain no face that is not incident at that edge.
- If an octree leaf node's region contains no edges, then it can contain at most one face.
- Each region's octree leaf node is maximal.

An implementation of the PM octree consists of leaf nodes of type vertex, edge, and face. For our purposes, it is permissible to have more than two faces meet at a common edge. However, such a situation cannot arise when modelling solids that are bounded by compact, orientable two-manifold surfaces (i.e., only two faces may meet at an edge and the surface is two-sided). Nevertheless, it is plausible when 3D objects are represented by their surfaces.

The space requirements of the PM octree are considerably harder to analyse than those of the region octree[37]. Nevertheless, it should be clear that the PM octree for a given object is much more compact than the corresponding region octree. For example, Figure 11(b) is a PM octree decomposition of the object in Figure 11(a).

For the proper execution of many operations, the fact that a node is of type vertex, edge, or face is not sufficient to properly characterize the object. It has

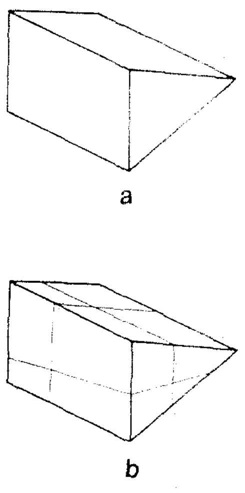


Figure 11. (a) Example three-dimensional object; and (b) its corresponding PM octree

been proposed that each node contains the polygons determined by the boundary of the object that intersects the node[31]. This is somewhat cumbersome and requires a considerable amount of extra information. Moreover, the vertices of the polygons cannot be specified exactly as most often they are not the vertices of the object. In particular, they are artifacts of the decomposition process. A more efficient solution is to store with each face node the equation of the plane of the face that is associated with the node[37]. In addition, the direction of the object relative to the face must be noted. For nodes of type edge and vertex, the configuration of the faces (i.e., the planes) that make up the edge and vertex must also be recorded. This information is used to classify a point inside the node with respect to the object so that the PM octree can be built.

The PM octree approach that we have described is based on permitting just one face in an octant with some exceptions. An alternative approach is to disregard the effects of edges and vertices and to use a simpler decomposition rule which is based on the number of faces that are permitted to occupy an octant. This is termed a bucketing approach by Samet[21]. It is used by Mäntylä and Tamminen,[18] who only permit a maximum of three faces to be in an octant. See Nelson and Samet[38,39] for the use of this technique for the representation of collections of line segments.

PM octree techniques have also been extended to handle curved surfaces. Primitives including cylinders and spheres have been used in conjunction with a decomposition rule limiting the number of distinct primitives that can be associated with a leaf node[40,41]. Another approach[23,42] extends the concepts of face node, edge node, and vertex node to handle faces represented by biquadratic Bézier primitives. The use of biquadratic Bézier primitives enables a better fit with fewer primitives than can be obtained with planar faces, thereby reducing the size of the octree. On the other hand, using biquadratc Bézier primitives may require more primitives than the use of the traditional bicubic Bézier primitive. Comparing the modelling efficiency of biquadratic Bézier primitives and cubic triangle-based Bézier primitives is less straightforward.

When working with linear features (such as polygonal faces and edges in PM octrees), the intersection of two linear objects (e.g., polygonal faces) is another linear object (e.g., a collection of points and line segments). However, for most Bézier primitives, the intersection of two Bézier primitives is not another Bézier primitive of the same type. Thus, although the borders of a Bézier surface primitive can be represented by Bézier curves, an edge formed by the intersection of two Bézier surface primitives might not be representable by a Bézier curve. This leads to two types of edge nodes and two types of vertex nodes – i.e., those that arise from the conventional presence of Bézier primitives (termed edge-*a* and vertex-*a*) and those that arise from the intersection of two Bézier primitives (termed edge-*b* and vertex-*b*)[23,42]. This complicates the extension of PM octree methods to Bézier primitives, since some design systems may wish to permit the user to create edges for which there is no simple representation. Brunet and Ayala[23] overcome this complication by requiring that such curves be adequately representable by their linear approximations. Of course, storing such linear approximations defeats the compactness and accuracy that result from representing the nonlinear features directly.

Brunet[43] makes use of the linear approximation method as a basis for the following decomposition rule to yield a face octree. A face octree consists of interior (black), exterior (white), and face nodes. Given a surface S and a tolerance ε, the decomposition is such that each face node is associated with just one part of the surface. For each face node, say F, there exists an oriented plane π_{ℓ} such that for every point P of S in F's cube, the distance from P to π_{ℓ} is less than or equal to ε. Each face node contains the explicit equation of the associated plane. Brunet shows that the set of planes can be limited by restricting the set of feasible normal directions and the distance to the origin. Every face node can be interpreted as a band which spans a distance equal to the tolerance on both sides of the plane that is associated with the node. Thus the whole boundary of the object is contained in the region defined by the 'union' of the bands of the face nodes in the octree, termed a thick surface. This means that the whole surface can be analysed just by studying the properties of the thick surfaces and there is no need to consider artificial geometric elements such as boundaries between smoothly connected patches.

From the quadtree complexity theorem we know that the number of nodes in the region octree is proportional to the surface area of the object being represented – i.e., the decomposition is at a maximum where the border of the objects passes through a node. Using a planar approximation of the surface within a node (as done by the face octree) can be expected to result in the number of nodes being proportional to the length of its edges – i.e., the decomposition is at a maximum where the edges of the surface of the object pass through a node. In this case, we must make some allowance for the curvature of the surface (the greater the degree of non-planarity of the surface, the greater the number of planar pieces necessary to obtain an accurate approximation of the surface). A further refinement of the PM octree decomposes until each node contains just one vertex (see the discussion of the PR quadtree in the next section). Thus the decomposition is at a maximum where the separation between two distinct vertices of the object is at a minimum. Such a decomposition rule may be difficult to achieve for curved surfaces. Nevertheless, it is worth noting that as the number of octree nodes decreases, the complexity of describing (and hence manipulating) the contents of a leaf node increases. Thus the question of whether to use an approach based on faces, edges, or vertices will depend on the application.

The difficulty in representing curved surfaces by using a PM octree lies in devising efficient methods of calculating the intersection between a Bézier primitive (or any other curved surface representation) and an octree node. This is computationally expensive in general. One possible way to overcome this expense is to use a decomposition rule such that a Bézier primitive is in a region if its bounding box is in the region (see the next section). However, in general, this rule does not halt in cases where the bounding boxes overlap while the Bézier primitives have no points in common. Thus the Bézier primitives have to be subdivided to determine which Bézier primitives are actually within which regions. The control structure for an algorithm based on this approach has been described in the first section. Observe that in this approach we are organizing a collection of patches (e.g., Bézier primitives) in the space that they occupy. This is in contrast to decomposing a single patch in its parametric representation by use of quadtree techniques (e.g.,[44]).

Spatially organizing bounding boxes of the surfaces

There are two principal methods of representing bounding boxes: point-based and region-based. Point-based representations treat each box as a point (e.g., its centroid, a combination of some of its extreme points, etc.) and then make use of data structures for points. Region-based representations treat each box as a region an hence take its extent into account. In order to simply the presentation, we restrict our example to two-dimensional data. This means that our examples will consist of rectangles. We first discuss point-based methods. This is followed by region-based methods.

One point-based representation reduces each bounding box to a representative point in a higher dimensional space, and then treats the problem as if we have a collection of points. This is the approach of Hinrichs and Nievergelt[45,46]. Each bounding box is a Cartesian product of two one-dimensional intervals where each interval is represented by its centroid and extent. The collection of bounding boxes is, in turn, represented by a grid file[47], which is a hierarchical data structure for points. The problem with this approach is that although the extent of the object is reflected in the representative point, the final mapping of the representative point in the 4D space does not result in the preservation of proximity in the 2D space from which the bounding boxes are drawn. In other words, two bounding boxes may be very close (and possibly overlap), yet the Euclidean distance between their representative points in four-dimensional space may be quite large, thereby masking the overlapping relationship between them.

Another approach which does not require a transition into a higher-dimensional space represents each bounding box by its centroid, which is a point in the 3D space within the bounding boxes. One approach to storing these centroids is to use a PR quadtree[21,48]. It is an adaptation of the region quadtree to point data which associates data points (that need not be discrete) with quadrants. The PR quadtree is organized in the same way as the region quadtree. The difference is that leaf nodes are either empty (i.e., white) or contain a data point (i.e., black) and its coordinates. Each block contains at most one data point. For example, Figure 12 is the PR quadtree corresponding to a set of points in a 2D space.

Data points are inserted into a PR quadtree by searching for them. Actually, the search is for the block in which the data point, say A, belongs (i.e., a leaf node). If the block is already occupied by another data

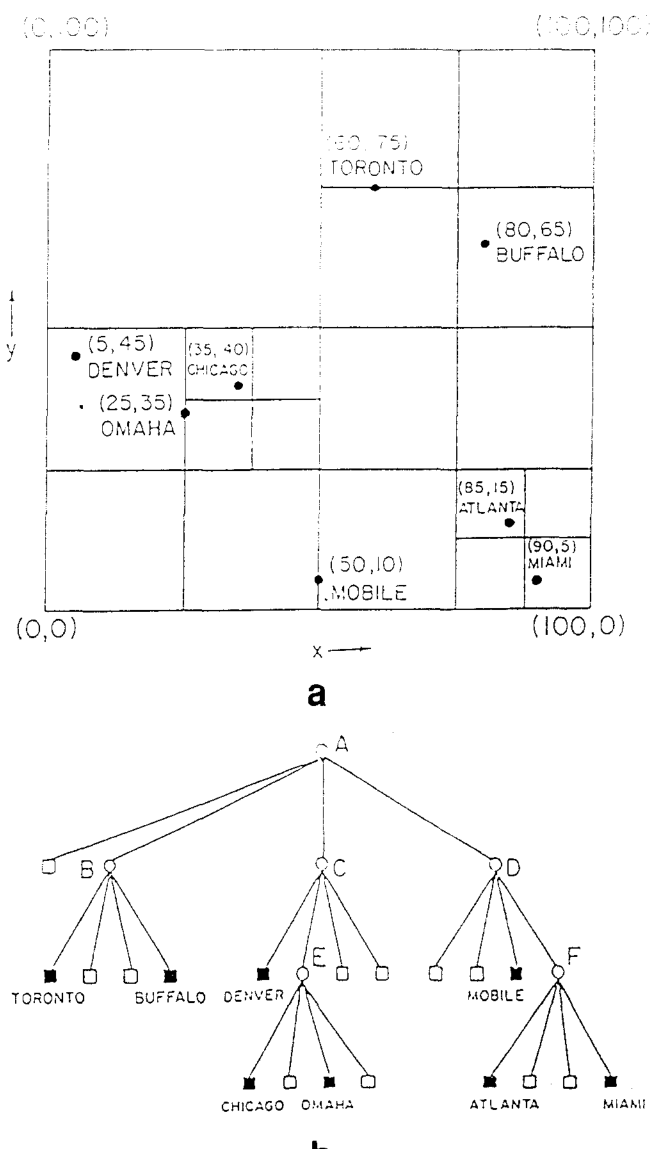


Figure 12. PR quadtree (b) and records it represents (a)

point with different x and y coordinates, say B, then the block must repeatedly be subdivided (termed splitting) until nodes A and B no longer occupy the same block. This may result in many subdivisions, especially if the Euclidean distance between A and B is very small. The shape of the resulting PR quadtree is independent of the order in which data points are inserted into it. Deletion of nodes is more complex and may require collapsing of nodes i.e., the direct counterpart of the node splitting process outlined above.

PR quadtrees, as well as other quadtree-like representations for point data, are especially attractive in applications that involve search. A typical query is one that requests the determination of all records within a specified distance of a given point. The efficiency of the PR quadtree lies in its role as a pruning device on the amount of search that is required. Thus many records will not need to be examined. For example, suppose that in the hypothetical database of Figure 12 we wish to find all points within eight units of a data point with coordinates (84,10). In such a case, there is no need to search the NW, NE, and SW quadrants of the root (i.e., (50,50)). Thus we can restrict our search to the SE quadrant of the tree rooted at root. Similarly, there is no need to search the NW, NE, and SW quadrants of the tree rooted at the SE quadrant (i.e., (75,25)). Note that the search ranges are usually orthogonally defined regions such as rectangles. Other search regions are also feasible as the above example demonstrated (i.e., a circle).

As another example, suppose that we wish to locate the nearest point to a given point, say P. This is achieved by a top-down recursive algorithm. Initially, at each level of the recursion, we explore the subtree that contains P. Once the leaf node containing P has been found, the distance from P to the nearest primitive in the leaf node is calculated (empty leaf nodes have a value of infinity). Next, we unwind the recursion so that at each level, we search the subtrees that represent regions overlapping a circle centred at P whose radius is the distance to the closest primitive that has been found so far. When more than one subtree must be searched, the subtrees representing regions nearer to P are searched before the subtrees that are farther away (since it is possible that a primitive in them might make it unnecessary to search the subtrees that are farther way).

For example, consider Figure 13 and the task of finding the nearest neighbour of P in node 1. We first visit node 1. If we visit nodes in the order NW, NE, SW, SE, then as we unwind for the first time, we visit nodes 2 and 3 and the subtrees of the eastern brother of 1. Once we visit node 4, there is no need to visit node 5 since node 4 contained A. However, we must still visit node 6 containing point B (closer than A), but now there is no need to visit node 7. Unwinding one more level reveals that due to the distance between P and B, there is no need to visit nodes 8, 9, 10, 11, and 12. However, node 13 must still be visited, as it could contant a point that is closer to P than B.

The situation is more complex when we are dealing with bounding boxes instead of points. In this case, the

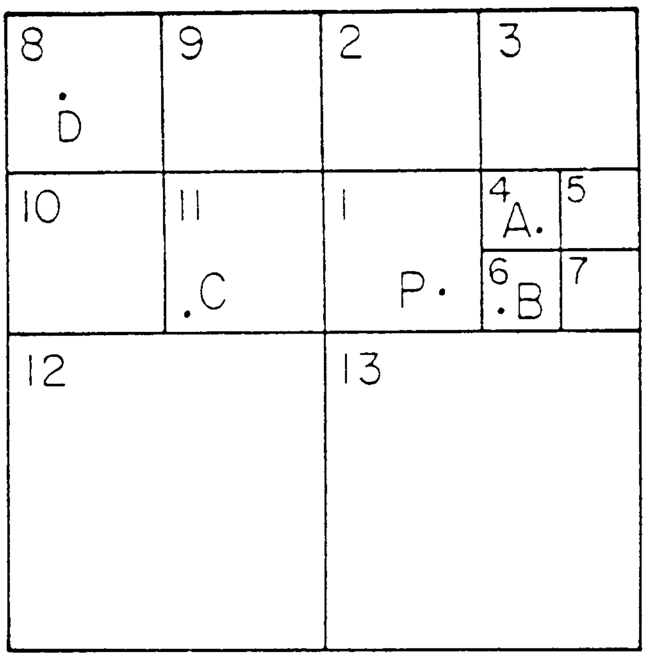


Figure 13. Example illustrating neighbouring object problem. P is location of pointing device. Nearest object is represented by point B in node 6

maximum amount of pruning that is possible is not always realized. The problem is that the object represented by the point may extend beyond the boundaries of the quadtree block in which it lies. Moreover, bounding boxes may overlap. Thus more blocks may have to be searched. The search process can be made more efficient by storing at each subtree of the quadtree the minimum and maximum values of the x and y coordinates of all the bounding boxes that are associated with the points stored in the subtree.

The second representation is region-based in the sense that the subdivision of the space from which the bounding boxes are drawn depends on the physical extent of the bounding box – not just one point in the bounding box. Representing the collection of bounding boxes, in turn, with a tree-like data structure has the advantage that there is a relation between the depth of a node in the tree and the size of the bounding box(es) associated with it. In the remainder of this section, we given an example of a region-based representation.

The MX-CIF quadtree of Kedem[49] (see also Abel and Smith[50]) is a region-based representation where each bounding box is associated with the quadtree node corresponding to the smallest block which contains it in its entirety. Subdivision ceases whenever a node's block contains no bounding box. Alternatively, subdivision can also cease once a quadtree block is smaller than a predetermined threshold size. This threshold is often chosen to be equal to the expected size of the bounding box[49]. For example, Figure 14 is the MX-CIF quadtree for a collection of bounding boxes. Note that bounding box F occupies an entire block and hence it is associated with the block's father. Also bounding boxes can be associated with both terminal and non-terminal nodes.

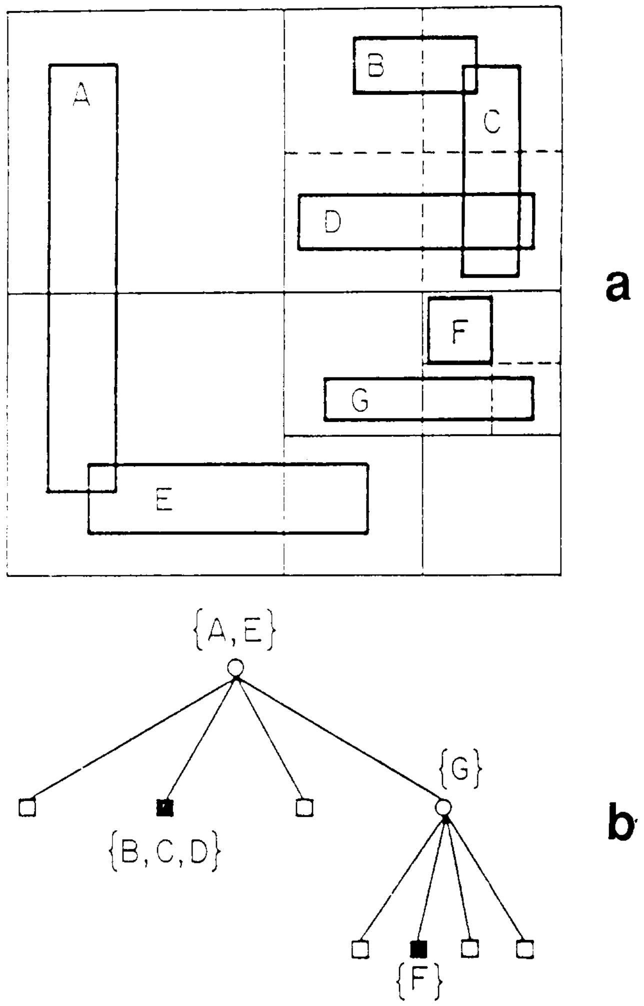


Figure 14. MX-CIF quadtree. (a) Collection of rectangles and block decomposition induced by MX-CIF quadtree. (b) Tree representation of (a)

It should be clear that more than one bounding box can be associated with a given enclosing block and, thus, often we find it useful to be able to differentiate between them. Kedem[49] proposes to do so in the following manner. Let P be a quadtree node with centroid (CX,CY), and let S be the set of bounding boxes that are associated with P. Members of S are organized into two sets according to their intersection (or collinearity of their sides) with the lines passing through the centroid of P's block – i.e., all members of S that intersect the line $x = CX$ form one set and all members of S that intersect the line $y = CY$ form the other set.

If a bounding box intersects both lines (i.e., it contains the centroid of P's block), then we adopt the convention that it is stored with the set associated with the line through $x = CX$. These subsets are implemented as binary trees (really tries), which in actuality are 1D analogues of the MX-CIF quadtree. For example, Figure 15 illustrates the binary tree associated with the y axes passing through the root and the NE son of the root of the MX-CIF quadtree of Figure 14. Interestingly, the MX-CIF quadtree is a 2D analogue of the interval tree[51,52], which is a data structure that is used to support optimal solutions based on the plane-sweep paradigm[53] to some

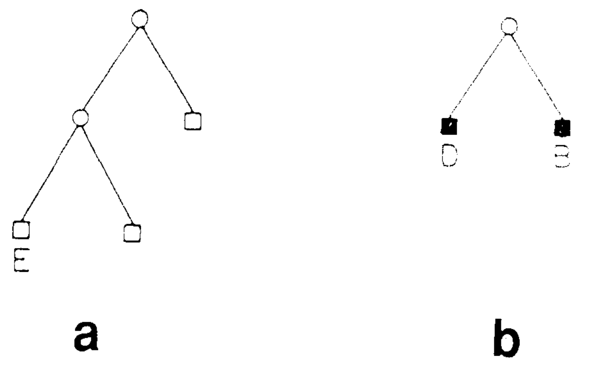


Figure 15. Binary trees for the y axes passing through (a) the root of MX-CIF quadtree in Figure 5 and (b) NE son of the root of MX-CIF quadtree in Figure 14

rectangle problems in computational geometry and VLSI designs.

In the case of 3D data we would use a PR octree as well as an MX-CIF octree. The PR octree is a straightforward extension of the PR quadtree. The MX-CIF octree can be defined in a number of ways. Recall that the MX-CIF quadtree is a two-level data structure in the sense that it also makes use of two one-dimensional MX-CIF quadtrees to differentiate between the cases where more than one bounding rectangle can be associated with a particular enclosing quadtree block. In the case of 3D data, we can use either a three-level structure or a two-level structure when more than one bounding box is associated with an enclosing octree block. In particular, the second level consists of three MX-CIF quadtrees (one for each of the *xy*, *xz*, and *yz* planes) while the third level consists of two 1D MX-CIF quadtrees for each of the three MX-CIF quadtrees of the second level. The third level is not used if we restrict ourselves to a two-level structure.

CONCLUDING REMARKS

A survey has been presented of data structure issues that arise in Bézier-based modelling. We have discussed both low-level implementation details as well as some higher level ones. The low level discussion has been in terms of the number of bytes that would be required to store the raw data corresponding to the Bézier primitives. The higher-level discussion has been with respect to supporting the maintenance of the topological integrity of the objects being modelled. This included an outline of a technique for adapting Euler operators to Bézier primitives. We also discussed how to increase the efficiency of certain operations by using hierarchical spatial data structures. Such representations facilitate the execution of operations that depend on spatial proximity (i.e., which patch is closest to another patch). The actual integration of these methods into existing implementations remains to be done and will undoubtedly lead to new results and additional research.

It is worth noting that our analysis of the results of using pointers (indices) to share vertex data among neighbouring Bézier primitives only leads to a minor reduction in the storage requirements in the case of Bézier control points. Thus, there is merit in not sharing the data for the control points so that all the data relevant to a particular Bézier primitive is contiguous in memory. This would lead to the greatest locality of references when a spatial organization is used to determine the actual location of the data in memory. Similar issues arise in geographic information systems[34].

ACKNOWLEDGEMENTS

The support of the National Science Foundation under Grant IRI-88-02457 is gratefully acknowledged.

REFERENCES

1 **Bézier, P** 'Mathematical and practical possiblities of UNISURF', in *Computer Aided Geometric Design* **Barnhill, R A and Riesenfeld, R F (eds)** Academic Press, New York (1974)

2 **Knuth, D E** *METAFONTbook* Addison-Wesley, Reading, MA (1986)

3 **Bézier, P** *The Mathematical Basis of the UNISURF CAD System*, Butterworth, Guildford, UK (1986)

4 **Bernstein, S** 'Dèmonstration du thérème de Weirstrass fondeé sur le calcul de probabilité' *Harkov Soobs. Matem ob-va 13* (1912), pp 1–2

5 **Farin, G** *Curves and Surfaces for Computer Aided Geometric Design*, Academic Press, San Diego (1988)

6 **de Casteljau, P** 'Courbes et surfaces à pôles' *Technical Report, Société André Citroen*, Paris (1963)

7 **de Casteljau, P** *Shape Mathematics and CAD*, Kogan Page, London (1986)

8 **Chang, G Z and Davis, P J** 'The convexity of Bernstein polynomials over triangles' *J. Approx. Theory* Vol 40 No 1 (1984) pp 11–28

9 **Chang, G Z and Hoschek, J** 'Convexity and variation diminishing property of Bernstein polynomials over triangles,' in *Multivariate Approximation Theory III*, **Schempp, W and Zeller, K (eds)** Birkhäuser Verlag, Basel, Switzerland (1985) pp 61–70

10 **Goodman, T N T** 'Variation diminishing properties of Bernstein polynomials on triangles' *J. Approx. Theory* Vol 50 No 2 (1987) pp 111–126

11 **Catmull, E** 'Computer display of curved surfaces' *Proc. Conf. Comput. Graph., Pattern Recognition, and Data Structure*, Los Angeles (1975) pp 11–17

12 **Lane, J M, Carpenter, L C, Whitted, T and Blinn, J F** 'Scan line methods for displaying parametrically defined surfaces' *Communications of the ACM* Vol 23 No 1 (1980) pp 23–34

13 **Baumgart, B G** 'Geometric modeling for computer vision' *PhD. dissertation, STAN-CS-463* Computer Science Department, Stanford University, Stanford, CA (1974)

14 **Mäntylä, M** *An Introduction to Solid Modeling* Computer Science Press, Rockville, MD (1987)

15 **Requicha, A A G** 'Representations of rigid solids: theory, methods, and systems' *ACM Computing Surveys* Vol 12 No 4 (1980) pp 437–464

16 **Mäntylä, M and Sulonen, R** 'GWB: A solid modeler with Euler operators' *IEEE Comput. Graph. Appl.* Vol 2 No 7 (1982) pp 17–31

17 **Dobkin, D P and Souvaine, D L** 'Computational geometry in a curved world' *Algorithmica* Vol 5 No 3 (1990) pp 421–457

18 **Mäntylä, M and Tamminen, M** 'Localized set operations for solid modeling' *Computer Graphics* Vol 17 No 3 (1983) pp 279–288 (also *Proceedings of the SIGGRAPH'83 Conference*, Detroit (1983))

19 **Lane, J M and Riesenfeld, R F** 'A theoretical development for the computer generation and display of piecewise polynomial surfaces' *IEEE Trans. Pattern Anal. Mach. Intell.* Vol 2 No 1 (1980) pp 35–46

20 **Lasser, D** 'Intersection of parametric surfaces in the Bernstein–Bézier representation' *Comput.-Aided Des.* Vol 18 No 4 (1986) pp 186–192

21 **Samet, H** *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA (1990)

22 **Samet, H** *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA (1990)

23 **Brunet, P and Ayala, D** 'Extended octtree representation of free form surfaces' *Comput.-Aided Geom. Des.* Vol 4 Nos 1–2 (1987) pp 141–154

24 **Aho, A V, Hopcroft, J E and Ullman, J D** *The Design and Analysis of Computer Algorithms* Addison-Wesley, Reading, MA (1974)

25 **Hunter, G M** 'Efficient computation and data structures for graphics' *PhD. dissertation*, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ (1978)

26 **Jackins, C L and Tanimoto, S L** 'Oct-trees and their use in representing three-dimensional objects' *Comput. Graph. and Image Processing* Vol 14 No 3 (1980) pp 249–270

27 **Meagher, D** 'Geometric modeling using octree encoding' *Comput. Graph. and Image Processing* Vol 19 No 2 (1982) pp 129–147

28 **Hunter, G M and Steiglitz, K** 'Operations on images using quad trees' *IEEE Trans. Pattern Anal. Mach. Intell.* Vol 1 No 2 (1979) pp 145–153

29 **Meagher, D** 'Octree encoding: a new technique for the representation, the manipulation, and display of arbitrary 3-D objects by computer' *Electrical and Systems Engineering Technical Report IPL-TR-80-111*, Rensselaer Polytechnic Institute, Troy, NY (1980)

30 **Ayala, D, Brunet, P, Juan, R and Navazo, I** 'Object representation by means of nonminimal division quadtrees and octrees' *ACM Trans. Graph.* Vol 7, No 1 (1985) pp 41–59

31 **Carlbom, I, Chakravarty, I and Vanderschel, D** 'A hierarchical data structure for representing the spatial decomposition of 3-D objects' *IEEE Comput. Graph. Appl.* Vol 5 No 4 (1985) pp 24–31

32 **Fujimura, K and Kunii, T L** 'A hierarchical space indexing method' *Proc. Comput. Graph. '85*, Tokyo (1985) T1-4, 1-14

33 **Hunter, G M** *Geometrees for interactive visualization of geology: an evaluation* System Science Department, Schlumberger-Doll Research, Ridgefield, CT (1981)

34 **Quinlan, K M and Woodwark, J R** 'A spatially-segmented solids database—justification and design *Proc. CAD'82 Conference*, Butterworth, Guildford, UK (1982) pp 126–132

35 **Tamminen, M** 'The EXCELL method for efficient geometric access to data' *Acta Polytechnica Scandinavica* Mathematics and Computer Science Series No. 34, Helsinki, Finland (1981)

36 **Vanderschel, D J** 'Divided leaf octal trees' *Research Note* Schlumberger-Doll Research, Ridgefield, CT (1984)

37 **Navazo, I** 'Contribució a les tècniques de modelat geomètric d'objectes polièdrics usant la codificació amb arbres octals' *PhD. dissertation*, Escola Tècnica Superior d'Enginyers Industrials, Departament de Metodes Informatics, Universitat Politècnica de Catalunya, Barcelona, Spain (1986)

38 **Nelson, R C and Samet, H** 'A consistent hierarchical representation for vector data' *Comput. Graph.* Vol 20 No 4 (1986) pp 197–206 (also *Proc. SIGGRAPH'86 Conference*, San Francisco (1987) pp 270–277

39 **Nelson, R C and Samet, H** 'A population analysis for hierarchical data structures' *Proc. SIGMOD Conference*, San Francisco (1987) pp 270–277

40 **Fujimoto, A, Tanaka, T and Iwata, K** 'ARTS: accelerated ray-tracing system' *IEEE Comput. Graph. Appl.* Vol 6 No 4 (1986) pp 16–26

41 **Wyvill, G and Kunii, T L** 'A functional model for constructive solid geometry' *Visual Computer* Vol 1 No 1 (1985) pp 3–14

42 **Navazo, I, Ayala, D and Brunet, P** 'A geometric modeller based on the exact octree-representation of polyhedra' *Comput. Graph. Forum* Vol 5 No 2 (1986) pp 91–104

43 **Brunet, P** 'Face octrees: involved algorithms and applications' *Technical Report LSI-90-14* Departament de Llenguatges i sistemes informàtics, Universitat Politècnica de Catalunya, Barcelona, Spain (1990)

44 **Carlson, W E** 'An algorithm and data structure for 3D object synthesis using surface patch intersections' *Comput. Graph.* Vol 16 No 3 (1982) pp 255–264 (also *Proc. SIGGRAPH'82 Conf.* Boston (1982))

45 **Hinrichs, K and Nievergelt, J** 'The grid file: a data structure designed to support proximity queries on spatial objects' *Proc. WG'83 (International Workshop on Graphtheoretic Concepts in Comput. Science)*, **Nagl, M and Perl, J (eds)** Trauner Verlag, Linz, Austria (1983) pp 100–113

46 **Hinrichs, K** 'The grid file system: implementation and case studies of applications' *PhD. dissertation*, Institut für Informatik, ETH, Zurich, Switzerland (1985)

47 **Nievergelt, J, Hinterberger, H and Sevcik, K C** 'The grid file: an adaptable, symmetric multikey file structure' *ACM Trans. Database Systems* Vol 9 No 1 (1984) pp 38–71

48 **Orenstein, J A** 'Multidimensional tries used for associative searching' *Inf. Process. Lett.* Vol 14 No 4 (1982) pp 150–157

49 **Kedem, G** 'The quad-CIF tree: a data structure for hierarchical on-line algorithms' *Proc. Nineteenth Design Automation Conference* Las Vegas (1982) pp 352–357

50 **Abel, D J and Smith, J L** 'A data structure and algorithm based on a linear key for a rectangle retrieval problem' *Comput. Vision, Graph. Image Processing* Vol 24, No 1 (1983) pp 1–13

51 **Edelsbrunner, H** 'Dynamic rectangle intersection searching' *Institute for Information Processing Report 47*, Technical University of Graz, Graz, Austria (1980)

52 **McCreight, E M** 'Efficient algorithms for enumerating intersecting intervals and rectangles' *Xerox Palo Alto Research Center Report CSL-80-09*, Palo Alto, CA (1980)

53 **Preparata, F P and Shamos, M I** *Computational Geometry: An Introduction*, Springer-Verlag, New York (1985)

54 **Shaffer, C A, Samet, H and Nelson R C** 'QUILT: a geographic information system based on quadtrees *Int. J. Geographical Inf. Systems* Vol 4 No 2 (1990) pp 103–131

55 **Bézier, P** *Emploi des Machines à Commude Numérique* Masson & Cie Editeurs, Paris (1970) (translated into English by **Forrest, A R and Pankhurst, A F** as *Numerical Control: Mathematics and Applications* Wiley, London (1972)

56 **Baumgart, B G** 'A polyhedron representation for computer vision' *Proc. National Comput. Conf. 44* Anaheim, CA (1975) pp 589–596

57 **Klinger, A** 'Patterns and search statistics', in *Optimizing Methods in Statistics* **Rustagi, J S (ed)** Academic Press, New York (1971) pp 303–337

58 **Reddy, D R and Rubin, S** 'Representation of three-dimensional objects' *CMU-CS-78-113 Computer Science Department*, Carnegie-Mellon University, Pittsburgh (1978)