The Arc Tree: An Approximation Scheme To Represent Arbitrary Curved Shapes

Oliver Günther
Department of Computer Science
University of California
Santa Barbara, CA 93106

Eugene Wong
Department of EECS
University of California
Berkeley, CA 94720

Abstract

This paper introduces the *arc tree*, a hierarchical data structure to represent arbitrary curved shapes. The arc tree is a balanced binary tree that represents a curve of length l such that any subtree whose root is on the k-th tree level is representing a subcurve of length $l/2^k$. Each tree level is associated with an approximation of the curve; lower levels correspond to approximations of higher resolution. Based on this hierarchy of detail, queries such as point search or intersection detection and computation can be solved in a hierarchical manner. We compare the arc tree to several related schemes and present the results of a practical performance analysis for various kinds of set and search operators. We also discuss several options to embed arc trees as complex objects in an extensible database management system and argue that the embedding as an abstract data type is most promising.

1. Introduction

The exact representation of curved geometric objects in finite machines is only possible if the objects can be described by finite mathematical expressions. Typical examples for such objects are paraboloids or ellipses, which can be described by functional equations such as $x^2/a^2+y^2/b^2=1$. Many applications, however, especially in computer vision and robotics, do not fit this pattern. The objects to be represented are rather arbitrary in shape, and some approximation scheme has to be employed to represent the data. Any finite machine can only store an approximate representation of the data with limited accuracy. In particular, the answer to any query is based on this approximate representation and may therefore be approximate as well.

Of course, the initial description of a curved object, coming from a camera, a tactile sensor, a mouse, or a digitizer may already be an *approximate* description of the real object. In most practical applications, this description will be a sequence of curve points or a spline, i.e. a piecewise polynomial function that is smooth and continuous. To support set, search, and recognition operators, however, it is more efficient to represent the data by a *hierarchy of detail* [Same84, Hopc87], i.e. a hierarchy of approximations, where higher levels in the hierarchy correspond to coarser approximations of the curve. Geometric operators can then be computed in a hierarchical manner: algorithms start out near the root of the hierarchy and try to answer the given query at a very coarse resolution. If that is not possible, the resolution is increased where necessary. In other words, algorithms "zoom in" on those parts of the curve that are relevant for the given query.

In this paper, we develop this theme of hierarchy of detail, focusing on the *arc tree*, a balanced binary tree that serves as an approximation scheme to represent arbitrary curved shapes. Section 2 gives a definition of the arc tree and shows how to obtain the arc tree representation of a given curve. Section 3 generalizes the concept of the arc tree to include other hierarchical curve representation schemes such as Ballard's strip trees [Ball81] and Bezier curves [Bezi74, Pavl82]. Sections 4 and 5 show how to perform point queries and set operations, such as union or intersection. Both sections also discuss the performance of our arc tree implementation. Section 6 outlines how to embed arc trees into an extensible database system such as POSTGRES [Ston86a], and section 7 contains a summary and our conclusions.

2. Definition

A *curve* is a one-dimensional continuous point set in d-dimensional Euclidean space $\mathbf{E}^d$. For simplicity, this presentation is restricted to the case d=2. The generalization to arbitrary d (with the curve remaining one-dimensional) is straightforward. A curve is *open* if it has two distinct endpoints, otherwise it is called *closed*; see figure 1 for some examples. As mentioned in the introduction, in practical applications, curves are usually given as a polygonal path, i.e. a sequence of curve points, or as a spline, i.e. a piecewise polynomial function that is smooth and continuous.

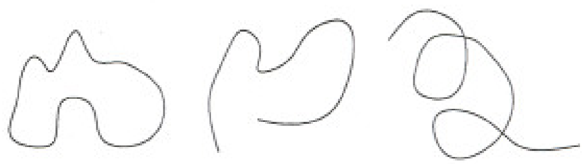


Figure 1: A closed and two open curves.

The arc tree scheme parametrizes a given curve according to its arc length and approximates it by a sequence of polygonal paths. Let the curve C have length l and be defined by a function $C(t)$:$[0,1] \rightarrow \mathbf{E}^2$, such that the length of the curve from $C(0)$ to $C(t_0)$ is $t_0 \cdot l$. The *k-th approximation* C_k (k=0,1,2...) of C is a polygonal path consisting of 2^k line segments $e_{k,i}$ (i=1..2^k), such that $e_{k,i}$ connects the two points $C(\frac{i-1}{2^k})$ and $C(\frac{i}{2^k})$. See figure 2 for an example.

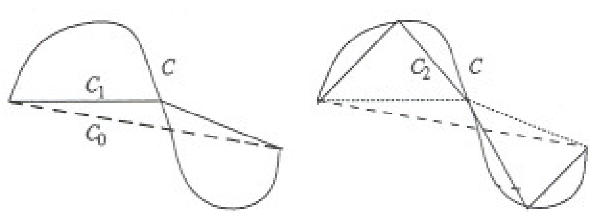


Figure 2: A 0th, 1st and 2nd approximation of a curve

Each edge $e_{k,i}$ can be associated with an arc $a_{k,i}$ of length $l/2^k$, which is a continuous subset of C. $C(\frac{i-1}{2^k})$ and $C(\frac{i}{2^k})$ are the common endpoints of $e_{k,i}$ and $a_{k,i}$. For $k \geq 1$, each k-th approximation is a refinement of the corresponding $(k-1)$-th approximation: the vertex set of the $(k-1)$-th approximation is a true subset of the vertex set of the k-th approximation.

More formally, the k-th approximation of C is defined by a piecewise linear function C_k:$[0,1] \rightarrow \mathbf{E}^2$ as follows. Here, $\underline{t}$ and $\overline{t}$ denote $\frac{\lfloor t \cdot 2^k \rfloor}{2^k}$ and $\frac{\lceil t \cdot 2^k \rceil}{2^k}$, respectively.

$$C_k(t) = \begin{cases} C(t) & t\cdot 2^k = 0..2^k \\ \frac{\overline{t}-t}{\overline{t}-\underline{t}}\cdot C(\underline{t}) + \frac{t-\underline{t}}{\overline{t}-\underline{t}}\cdot C(\overline{t}) & \text{otherwise} \end{cases} \qquad t \in [0,1]$$

There are various criteria in common use that measure the error between a curve C and its k-th polygonal approximation C_k [Imai86]. In the case of the arc tree, one could use the maximum distance between a curve point and the corresponding point of the approximation:

$$\max_{0 \le t \le 1} d(C_k(t), C(t))$$

Here, d denotes Euclidean distance. This criterion will be referred to as (e1). Other possibilities include the maximum distance between a line segment $e_{k,i}$ and the curve points on the corresponding arc $a_{k,i}$ (criterion e2):

$$\max_{0 \le t \le 1} d(e_{k,\lceil t\cdot 2^k \rceil}, C(t))$$

or the maximum distance between the *line* containing $e_{k,i}$ (denoted by $line(e_{k,i})$) and the arc $a_{k,i}$ (criterion e3). For brevity reasons, the following theorem is presented without proof.

Theorem 1: According to any of the error criteria described above, the error between a curve C and its k-th approximation C_k is no more than $l/2^{k+1}$. □

Lemma 2: Using any of the above error criteria, the sequence of approximation functions $(C_k(t))$ converges uniformly towards $C(t)$.

Proof: It follows from theorem 1 that the error converges towards zero for $k \to \infty$, which proves the lemma. □

Moreover, for each approximation C_k there is a well-defined area that contains the curve. We have

Lemma 3: Let $E_{k,i}$ denote the ellipse whose major axis is $l/2^k$ and whose focal points are the two endpoints of the edge $e_{k,i}$, $C(\frac{i-1}{2^k})$ and $C(\frac{i}{2^k})$. Then the arc $a_{k,i}$ is internal to $E_{k,i}$.

Proof: (by contradiction) Let $X \in a_{k,i}$ denote a point external to $E_{k,i}$. Then

$$d(X, C(\frac{i-1}{2^k})) + d(X, C(\frac{i}{2^k})) > l/2^k$$

Thus, the length of $a_{k,i}$ would be greater than $l/2^k$ which is a contradiction. □

Corollary 4: The curve C is internal to the area formed by the union of the bounding ellipses, $\bigcup_{i=0}^{2^k} E_{k,i}$ $(k=0, 1, \ldots)$. □

See figure 3 for an example.

The family of approximations of a given curve C can be stored efficiently in a binary tree. The root of the tree contains the three points $C(0)$, $C(1/2)$ and $C(1)$ and is considered on level 0. If a tree node on level i contains point $C(\frac{x}{2^{i+1}})$ $(x=1 \ldots 2^{i+1}-1)$, then its left son contains point $C(\frac{2x-1}{2^{i+2}})$, and its right son contains point $C(\frac{2x+1}{2^{i+2}})$. We call this tree the *arc tree* of the curve C. The arc tree is an exact representation of C; each of its subtrees represents a continous subset of C. An inorder traversal of the first k $(k \ge 0)$ levels of the arc tree yields the vertices of the $(k+1)$-th approximation, sorted by increasing t. On the other hand, a breadth-first traversal of the first k levels yields these vertices in an order such that the first 2^i+1 vertices form the i-th approximation of C $(i=1,2 \ldots k+1)$. See figure 4 for an example.

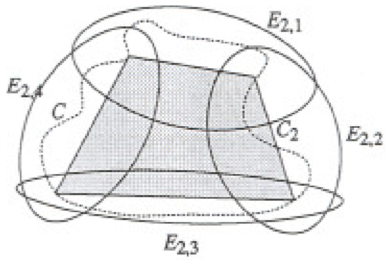


Figure 3: A curve C with its 2nd approximation C_2 and corresponding ellipses $E_{2,i}$.

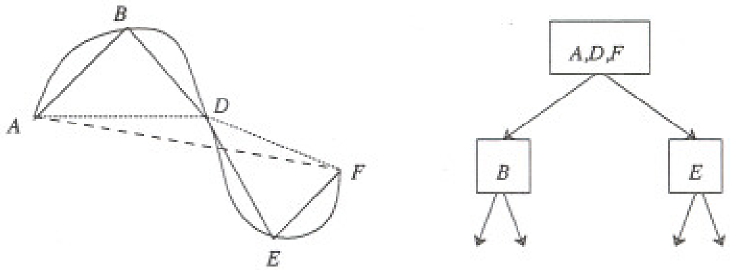


Figure 4: A curve with approximations and its arc tree. For a closed curve, it is $A = F$.

In practice, only a finite number of levels of the arc tree is stored. An arc tree with r levels is called an arc tree of *resolution* r. It is a balanced binary tree and it represents the 0th through $(r+1)$-th approximation of C.

An arc tree of resolution r can be constructed in two traversals of the given curve C. In the first round, one determines the length l of C. If C is a spline (or a polygonal path), l can be computed using the following formula for the arc length of an analytical curve. If the curve is given by $y = f(x)$, its length between the points $P_1(x_1,y_1)$ and $P_2(x_2,y_2)$ is

$$l = \int_{x_1}^{x_2} \sqrt{1+f'^2(x)}dx$$

If it is given by $x = x(t)$, $y = y(t)$, its arc length is

$$l = \int_{t_1}^{t_2} \sqrt{x'^2(t) + y'^2(t)}dt$$

with $x_i = x(t_i)$ and $y_i = y(t_i)$. One may also attach a label to each knot of C indicating the length accumulated so far. This does not require any additional computation, but it will speed up the second round. In the second round, one picks up the curve points $C(\frac{i}{2^r})$ $(i \in \{0,1..2^r\})$ and inserts them into the appropriate tree nodes while performing an inorder traversal of the tree.

Note that arc trees can be used to represent any given curve that can be parametrized with respect to arc length. This requirement poses no problem if the input curve is given as a polygonal path or a spline. Nevertheless, there remain problems with some curves such as fractals, for example [Mand77], or with curves that are distorted by high-frequency noise. In both cases the concept of arc length becomes somewhat meaningless and it is necessary to smooth the curve first before the parametrization can take place.

3. Generalization

The arc tree parametrizes the given curve by arc length and localizes it by means of bounding ellipses. At higher resolutions the number of ellipses increases, but their total area decreases, thus providing a better localization.

The arc tree can be viewed as just one instance of a large class of approximation schemes that implement a hierarchy of detail. Higher levels in the hierarchy correspond to coarser approximations of the curve. Associated with each approximation is a *bounding area* that contains the curve. Set and search operators are computed in a hierarchical manner: algorithms start out near the root of the hierarchy and try to solve the given problem at a very coarse resolution. If that is not possible, the resolution is increased where necessary.

In this section we will present several approximation schemes that are based on the same principle, but that use different parametrizations or bounding areas. For all of these schemes, it is fairly straightforward to obtain the representation of a given curve. Moreover, the algorithms for the computation of set and search operators are very similar to the corresponding arc tree algorithms, which are presented in sections 4 and 5. It is a subject of further research to conduct a detailed practical comparison of these schemes to find out which schemes are suited best for certain classes of curves.

The first modification of the arc tree concerns the choice of the ellipses $E_{k,i}$ as bounding areas. These ellipses provide the tightest possible bound but, on the other hand, ellipses are fairly complex objects, which has a negative impact on the performance of this scheme. For example, it is often necessary to test two bounding areas for intersection; if the bounding areas are ellipses, this operation is rather costly. Our implementation showed that it is in fact sometimes more efficient to replace the ellipses by their bounding circles; see section 5.1. The circles provide a poorer localization of the curve, but they are easier to handle computationally, which caused the total performance to improve. Other alternatives would be to use bounding boxes whose axes are parallel to the coordinate axes or to the axes of the ellipses. Both of these approaches, however, proved to be less effective than the bounding circles.

If the curves to be represented are polygonal paths with relatively few vertices, it is more efficient to break up the polygonal paths at their vertices rather than to introduce artificial vertices $C(\frac{i}{2^k})$ ($i=1\,..\,2^k-1$). If a polygonal path has $n+1$ vertices $v_1\,..\,v_{n+1}$, it can be represented *exactly* by a *polygon arc tree* of depth $\lceil \log_2 n \rceil$ as follows. The root of the polygon arc tree contains the vertices v_1, $v_{\lceil n/2 \rceil+1}$, and v_{n+1}. Its left son contains the vertex $v_{\lceil n/4 \rceil+1}$, its right son the vertex $v_{\lceil 3/4 \cdot n \rceil+1}$, and so on, until all vertices are stored. Clearly, the arc length corresponding to a node is no more implicit; it has to be stored explicitly with each node. In particular, at each node N it is necessary to know the lengths of the subcurves corresponding to N's left and right subtree. An example is given in figure 5.

It is easily seen that some of this length data is redundant. Indeed, with some care it is sufficient to store only one arc length datum per node. For this reason, the storage requirements for a polygon arc tree are only about 20% to 40% higher than for a regular arc tree of the same depth.

A data structure closely related to the polygon arc tree is the *Binary Searchable Polygonal Representation (BSPR)* proposed by Burton [Burt77].

There are other structures that also implement some hierarchy of detail. One of them is the *strip tree*, introduced by Ballard [Ball81]. As the arc tree, the strip tree represents a given curve C by a

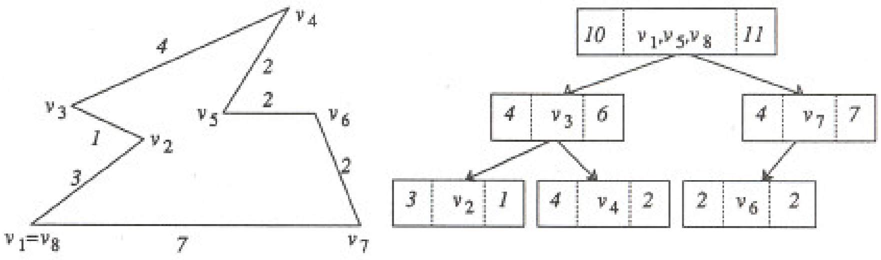


Figure 5: A polygon and corresponding polygon arc tree. The numbers in italics denote arc length.

binary tree such that each subtree T represents a continuous part C_T of C. C_T is approximated by the line segment connecting its endpoints (x_b, y_b) and (x_e, y_e). The root node of T stores these two endpoints and two widths w_l and w_r, thus defining a bounding rectangle S_T (the *strip*) that tightly encloses the curve segment C_T. S_T has the same length as the line segment $((x_b, y_b),(x_e, y_e))$ and its sides are parallel or perpendicular to it. See figure 6 for an example of a curve and a corresponding strip tree. Clearly, this approach requires some extensions for closed curves and for curves that extend beyond their endpoints (fig. 7).

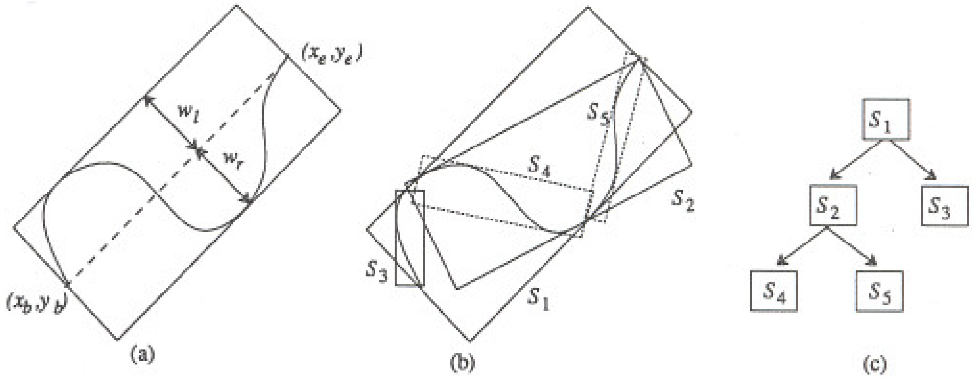


Figure 6: A curve with strip, a hierarchy of strips, and a corresponding strip tree.

When a strip tree is constructed for a given curve C, a curve segment C_T is subdivided further until the total strip width w_l+w_r is below a certain threshold. As it is a non-trivial operation to obtain the strip S_T for every curve segment C_T, the construction of a strip tree for a given curve may be quite costly. To subdivide C_T, one can choose any point of C_T that lies on the boundary of the corresponding strip S_T. Clearly, a strip tree is not necessarily balanced, which has a negative impact on its average-case performance. Note that arc trees are always balanced, which might give them an edge over strip trees in terms of average performance.

Also, a strip tree requires about twice as much space as an arc tree of same depth: each arc tree node stores a minimum of two real numbers and two pointers, whereas a strip tree node stores six real

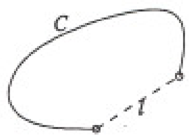


Figure 7: A curve C that extends beyond its endpoints. There is no bounding box of length l that contains C.

numbers and two pointers. Note, however, that strip trees can be modified to require less storage. First, all subdivision points belong to more than one strip and are therefore stored in more than one node. The redundant data may be replaced by pointers or deleted; in the latter case, the strip tree algorithms given by Ballard would have to be somewhat modified. Second, rather than storing w_l and w_r, one may just store the maximum of these two widths. The corresponding strip is potentially wider and provides a poorer localization. In both cases, some loss in performance is likely, but it will probably be minor compared to the savings in storage space.

A generalization of the strip tree to higher dimensions is possible. The *prism tree* of Ponce and Faugeras, for example [Ponc87], approximates free-form solids in three dimensions by means of truncated pyramides. The arc tree, on the other hand, does not have an immediate equivalent in higher dimensions because the parametrization method (by arc length) is impractical to generalize to curved surfaces.

A very different approach to implement a hierarchy of detail is based on curve fitting techniques such as *Bezier curves* [Bezi74] or *B-splines* [Debo78]; see also [Pavl82] for a good survey of these and related techniques. A Bezier curve of degree m is an m-th degree polynomial function defined by $m+1$ *guiding points* $P_1 \,..\, P_{m+1}$. The curve goes through the points P_1 and P_{m+1} and passes near the remaining guiding points $P_2 \,..\, P_m$ in a well-defined manner. The points P_2 through P_m may be relocated interactively to bring the Bezier curve into the desired form. See figure 8 for two examples.

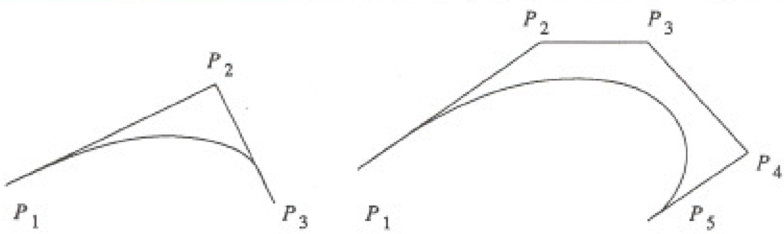


Figure 8: Examples of Bezier polynomials with three and five guiding points.

It can be shown that a Bezier curve lies within the corresponding *characteristic polygon*, i.e. the convex hull of its guiding points. Also, a Bezier curve B can be subdivided into two Bezier curves B_1 and B_2 of same degree. The characteristic polygons of B_1 and B_2 are disjoint and subsets of B's characteristic polygon. They therefore provide a better localization of B; see figure 9.

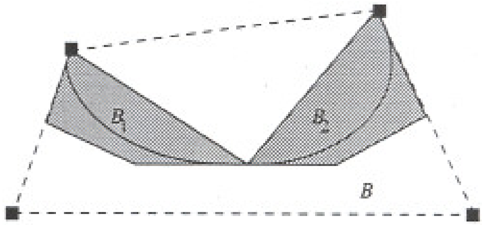


Figure 9: A Bezier curve B subdivided into two curves B_1 and B_2 with characteristic polygons.

Now we can derive a hierarchical representation of a given Bezier curve B as follows. The first approximation is the edge segment connecting B's endpoints; its bounding area is given by B's characteristic polygon. The second approximation is the polygonal path connecting the endpoints of B_1 and B_2; its bounding area is the union of the characteristic polygons of B_1 and B_2, and so on. There are various efficient subdivision algorithms to obtain B_1 and B_2 from a given B; see for example [Pavl82], pp. 221-230.

The main problem with this approach seems to be that not every curve can be approximated well by a low-order Bezier curve. A high-order Bezier curve, however, is harder to subdivide and has a more complex characteristic polygon, which has an adverse impact on the performance of this scheme. In practice, complex curves are often approximated by *several* third-order Bezier curves. This would mean that the bounding area of the first approximation is a union of convex polygons, which is already rather complex. Further approximations are then obtained by subdivisions of each one of these polygons. Nevertheless, this approach seems very promising and should be included in a practical comparison of the various approaches to implement a hierarchy of detail.

We expect arc or strip trees to be superior to Bezier curves if the curves to be represented are initially described by a long sequence of curve points and can only be described by high-order splines or a large number of simpler splines. This is often the case if curves are input from a digitizer pad or a mouse. On the other hand, if a curve is initially given by a few simple splines, it is probably more efficient to keep this representation and use spline subdivision algorithms as described above to implement a hierarchy of detail.

B-splines can be used in a way similar to Bezier curves to implement a hierarchy of detail. For appropriate subdivision algorithms, see [Bohm84].

Certainly, there are many more possibilities to implement a hierarchy of detail as a tree structure similar to the schemes presented above. Note that in all of these schemes it is possible to trade space with time as follows. Rather than storing all lower level approximations explicitly, one could keep the source description of the curve in main memory and compute finer approximations ''on the fly'' when needed. This approach can be viewed as a *procedural arc tree* as finer approximations are defined procedurally, i.e. by means of the appropriate subdivision algorithm that computes finer approximations from coarser ones. This approach seems particularly promising for the Bezier approach where highly efficient subdivision algorithms are available. In the case of arc and strip trees, the computations to obtain finer approximations are probably too complex to be repeated at every tree traversal.

As mentioned above, the algorithms for set and search operations for these various approximation schemes are all essentially the same. In the following two sections, we give the algorithms for the arc tree scheme. In most cases, the corresponding algorithms for the other schemes are simply obtained by replacing the ellipses $E_{k,i}$ by the corresponding bounding areas, viz., the characteristic polygons for the curve fitting approaches or the strips for the strip tree.

4. Hierarchical Point Inclusion Test

To demonstrate the power of the arc tree representation scheme, we first show how to answer point queries on the arc tree. Given a point $A \in E^2$ and a simple (i.e. non self-intersecting) closed curve C, a point query asks if A is internal to the point set enclosed by C, $P(C)$. For simplicity, we also describe this case by stating that A is internal to C, or that $A \in P(C)$.

The point inclusion test is performed by a hierarchical algorithm called *HPOINT*, which starts with some simple approximation C_{app} of C. For each edge $e_{k,i}$ of C_{app} (i=1..2^k), it checks if the replacement of $e_{k,i}$ by the arc $a_{k,i}$ may affect the internal/external classification of A. If there is no such edge $e_{k,i}$, then $A \in P(C_{app})$ is equivalent to $A \in P(C)$; *HPOINT* uses a conventional algorithm for polygons to solve the point query $A \in P(C_{app})$? and terminates with that result. Otherwise, *HPOINT* replaces each edge $e_{k,i}$, whose replacement by $a_{k,i}$ may affect A's classification, by the two edges $e_{k+1,2i-1}$ and $e_{k+1,2i}$. *HPOINT* proceeds with the resulting polygon, which is a closer approximation of C.

If the maximum resolution has been reached without obtaining a result, then the problem cannot be decided at that resolution. In fact, there are boundary points (such as $C(\frac{1}{3})$) that cannot be decided at *any* finite resolution. There are three ways to resolve this situation: (i) the algorithm returns *unclear*, (ii) the algorithm considers the point a boundary point, or (iii) the arc tree is extended at its leaf nodes to include the source description of the curve; then, edges $e_{k,i}$ may eventually be replaced by arcs $a_{k,i}$ to allow an exact query evaluation. For *HPOINT*, we choose option (ii), thus considering the boundary as having a nonzero width. In our definition of the point inclusion test, where the given point set $P(C)$ is closed, *HPOINT* returns $A \in P(C)$, accordingly.

We are left with the problem of how to find out quickly if the replacement of $e_{k,i}$ by $a_{k,i}$ may affect the internal/external classification of A. From lemma 3, we obtain

Lemma 5: Let $C_{k,i}$ denote the curve obtained from C by replacing the arc $a_{k,i}$ by the straight line $e_{k,i}$. If A is external to $E_{k,i}$ then $A \in P(C)$ is equivalent to $A \in P(C_{k,i})$.

Proof: Because A is external to $E_{k,i}$, A may not lie on or between $a_{k,i}$ and $e_{k,i}$. Therefore, the replacement of $a_{k,i}$ by $e_{k,i}$ may not affect the internal/external classification of A. □

It is therefore sufficient to check if A is internal to $E_{k,i}$. If yes, the replacement of $e_{k,i}$ by $a_{k,i}$ may affect the classification of A, otherwise it may not. Letting the initial approximation be C_0, *HPOINT* can be described more precisely as follows.

Algorithm HPOINT

Input: A point $A \in E^2$. The arc tree T_C of a simple closed curve C.

Output: $A \in P(C)$?

(1) Set the approximation polygon C_{app} to C_0, k to zero, and tag the edge $e_{0,1}$ of C_{app}.

(2) For each tagged edge $e_{k,i}$ ($i \in \{1..2^k\}$) of C_{app}, check if A is external to the ellipse $E_{k,i}$. If yes, untag $e_{k,i}$.

(3) If C_{app} has no tagged edges left, use a conventional point inclusion algorithm for polygons to determine if $A \in P(C_{app})$, return the result and stop.

(4) Otherwise, if k is less than the maximum resolution, $depth(T_C)$, replace each tagged edge $e_{k,i}$ by the two tagged edges $e_{k+1,2i-1}$ and $e_{k+1,2i}$, increase k by one and repeat from (2).

(5) Otherwise, A is a boundary point; return *true* and stop.

Step (2) can easily be done by computing the distances from A to the two focal points of $E_{k,i}$. Step (4) can be performed by using C's arc tree in the following manner. Each edge $e_{k,i}$ is associated with the subtree whose root contains the point $C(\frac{2i-1}{2^{k+1}})$. Note that this is the curve point which corresponds to the center point of $e_{k,i}$ and which $e_{k+1,2i-1}$ and $e_{k+1,2i}$ have in common. If $e_{k,i}$ is to be replaced by $e_{k+1,2i-1}$ and $e_{k+1,2i}$, *HPOINT* obtains that point from the tree node and continues recursively on both subtrees of this node.

Steps (2) and (4) can now be performed during a top-down traversal of the arc tree. Each subtree can be processed independently of the others, which offers a natural way to parallelize the algorithm. If C_{app} has no more tagged edges, then the partial results are collected in a bottom-up traversal of the tree and put together to form the boundary of the final approximation polygon C_{app}. At this point, $A \in P(C)$ is equivalent to $A \in P(C_{app})$. Step (4) can be performed by Shamos' algorithm, where one constructs a horizontal line L through A and counts the intersections between L and the edges of C_{app} that lie to the left of A. If the number of intersections is odd, then A is internal, otherwise it is external. Shamos' algorithm requires some special maintenance for horizontal edges; see [Prep85] for details.

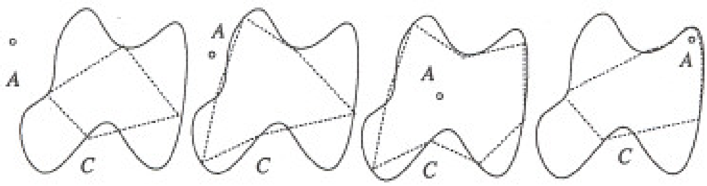
We implemented this algorithm on a VAX 8800 and ran several experiments to see how *HPOINT*'s time complexity correlates with the complexity of the given curve C and with the location of A with respect to C. Our running times should not be considered in absolute terms as we did not make a great effort to optimize our code. However, the figures are appropriate for comparative measurements. Figures 10 and 11 show our results. Here, t is CPU time in *ms*, and r is the resolution at which the query was decided. The dotted polygons are the final approximations C_{app}, respectively.

Note that the use of alternative approximation schemes is unlikely to improve the performance of our algorithms. To test a given point for inclusion in a given ellipse has about the same complexity as the corresponding tests for a characteristic polygon (say, a convex quadrilateral) or a strip. On the other hand, the test is somewhat easier for circles or for boxes whose axes are parallel to the coordinate axes. In both cases, however, the localization of the curve that is provided by these areas is poorer than for the bounding areas above.

Our algorithm *HPOINT* is an application of the idea of *hierarchy of detail*, as described by Samet [Same84] or Hopcroft and Krafft [Hopc87]. It solves the point inclusion problem by starting with a very simple representation of C and introduces more complex representations only if they are required to solve the problem. The algorithm "zooms in" on those parts of C that are interesting in the sense that they may change the internal/external classification of the point A at a higher resolution. As our examples demonstrate, *HPOINT* terminates very quickly if A is not close to C. The closer A gets to C, the higher is the resolution required to answer the point query. Due to a quick localization of the interesting parts of C, the algorithm does not show the quadratic growth in the complexity of C that a worst-case analysis would predict.

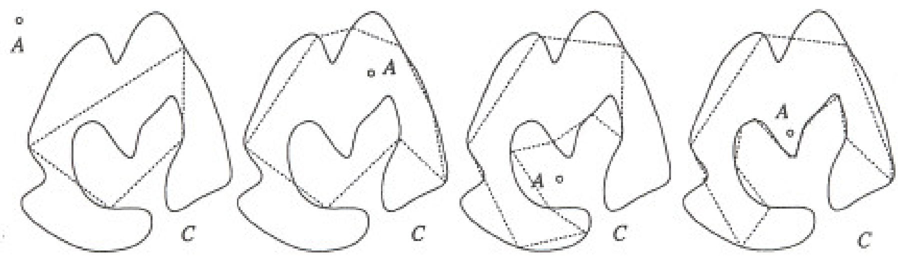
5. Hierarchical Set Operations

In this section, we show how to detect and compute intersections between curves. Other set operations such as union or difference or set operations on areas can be computed in a similar manner [Gunt88]. Again, the idea is to inspect approximations of the input curves by increasing resolution and



(a) r=2, t=4.0 (b) r=3, t=5.1 (c) r=3, t=5.5 (d) r=6, t=8.0

Figure 10: C is a spline with 12 knots.



(a) r=2, t=4.0 (b) r=4, t=6.4 (c) r=4, t=7.1 (d) r=6, t=8.9

Figure 11: C is a spline with 36 knots.

to "zoom in" on those parts of the boundaries that may participate in an intersection.

5.1. Curve-Curve Intersection Detection

We first show how to test two given curves C and D for intersection. The hierarchical algorithm *HCURVES* starts with simple approximations C_{app} and D_{app} of C and D, respectively, and continues with approximations of higher resolutions where necessary. For brevity, the following lemma is presented without proof.

Lemma 6: The arcs $a_{k,i}$ and $b_{k,j}$ corresponding to the edges $e_{k,i}$ of C_{app} and $f_{k,j}$ of D_{app}, respectively, *must* intersect if the following three conditions are met:

(i) $e_{k,i}$ intersects $f_{k,j}$,

(ii) the two endpoints of $e_{k,i}$ are external to the ellipse $F_{k,j}$ corresponding to $f_{k,j}$,

(iii) the two endpoints of $f_{k,j}$ are external to the ellipse $E_{k,i}$ corresponding to $e_{k,i}$. □

Now the algorithm *HCURVES* proceeds as follows. For each pair of edges, $e_{k,i}$ of C_{app} and $f_{k,j}$ of D_{app} $(i,j \in \{0,1..2^k\})$, *HCURVES* checks if their corresponding arcs *may* intersect. According to lemma 3, this can be done by testing if the corresponding ellipses $E_{k,i}$ and $F_{k,j}$ intersect. If yes, *HCURVES* puts tags on $e_{k,i}$ and $f_{k,j}$ and applies lemma 6 to see if the arcs *must* intersect. If yes, *HCURVES* reports an intersection and stops. After all pairs of edges have been processed, *HCURVES* checks if there are any tagged edges. If no, *HCURVES* reports no intersection and stops. Otherwise, *HCURVES* replaces all tagged edges by the corresponding edges of the next higher approximation, increases k by one, and proceeds with the refined curves. If the maximum resolution has been reached and there are still tagged edges, *HCURVES* interprets the situation as an intersection of the boundaries and returns an intersection. More exactly, *HCURVES* can be described as follows.

Algorithm HCURVES

Input: The arc trees T_C and T_D of two curves C and D.

Output: $C \cap D \neq \phi$?

(1) Set the approximation polygons C_{app} to C_0, D_{app} to D_0, and k to zero.

(2) For each pair of edges $e_{k,i}$ of C_{app} and $f_{k,j}$ of D_{app} do

  (2a) Check if the two ellipses $E_{k,i}$ and $F_{k,j}$ intersect.

  (2b) If yes, tag $e_{k,i}$ and $f_{k,j}$; if conditions (i) through (iii) in lemma 6 are met or if $e_{k,i}$ and $f_{k,j}$ share one or two endpoints, return *true* and stop.

(3) If there are no tagged edges, return *false* and stop.

(4) If k is less than the maximum resolution, min(*depth*(T_C),*depth*(T_D)), replace each tagged edge $e_{k,i}$ of C_{app} by the two edges $e_{k+1,2i-1}$ and $e_{k+1,2i}$. Similarly for each tagged edge $f_{k,j}$ of D_{app}. Increase k by one and repeat from (2).

(5) Otherwise, the maximum resolution has been reached; return *true* and stop.

We implemented this algorithm on a VAX 8800 with a few slight modifications to speed up execution. First, the test if the two ellipses $E_{k,i}$ and $F_{k,j}$ intersect is replaced by a test if the two circumscribing circles of $E_{k,i}$ and $F_{k,j}$ intersect. If those do not intersect then the ellipses do not intersect either. Otherwise, we assume that the ellipses may intersect and proceed accordingly. We made several experiments with more accurate tests, such as to test bounding boxes of the two ellipses for intersection, or to test the two ellipses themselves for intersection. In every case, the execution times went up between 25% and 60%. The more accurate tests required a significant amount of CPU time, but they only marginally reduced the number of tagged edges.

Second, rather than performing step (2) for each pair of edges $e_{k,i}$ of C_{app} and $f_{k,j}$ of D_{app}, we maintain a list to keep track which pairs of ellipses $(E_{k,i}, F_{k,j})$ pass the intersection test in step (2a). Then, step (2) is executed for a pair of edges $(e_{k,i}, f_{k,j})$ if and only if the ellipses $E_{k-1,\lceil i/2 \rceil}$ and $F_{k-1,\lceil j/2 \rceil}$, which correspond to their parent edges, intersect. Otherwise, it is known in advance that $E_{k,i}$ and $F_{k,j}$ do not intersect.

Figures 12 and 13 give several examples for the performance of the algorithm. Here, r denotes the resolution at which the algorithm is able to decide the query, and t denotes the CPU time in *ms*.

Again, it is not clear if the use of alternative approximation schemes might yield a better performance. The crucial operation in algorithm *HCURVES* is the test if two bounding areas intersect. In the case of circles, this is a trivial operation: two circles intersect if the distance between their centers is no more than the sum of their radii. The corresponding tests for boxes or characteristic polygons (say, convex quadrilaterals) are about two to three times as complex.

Note that the running times do not grow quadratically with the complexity of the input curves. The example in figure 13 (b) requires a large amount of CPU time due to the fact that the two curves are quite interwoven but do not intersect. It is therefore necessary to get down to fairly high resolutions in order to determine that there is no intersection. It seems that a case like this will require a lot of

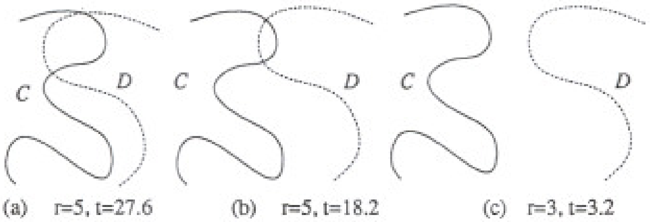


Figure 12: C is a spline with 13 knots, D a spline with 8 knots.

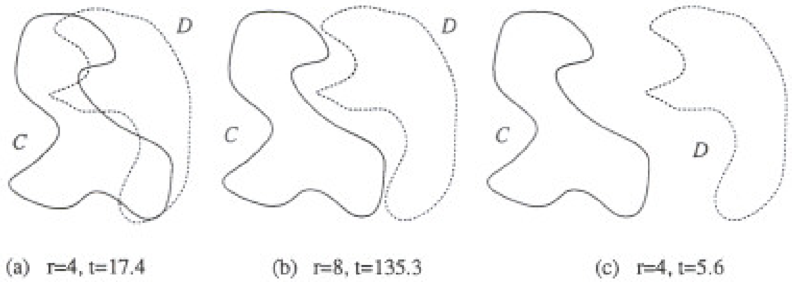


Figure 13: C is a spline with 24 knots, D a spline with 23 knots.

computation with any other intersection detection algorithm as well.

5.2. Curve-Curve Intersection Computation

The intersection is actually computed by the hierarchical algorithm *HCRVCRV*, which is a variation of algorithm *HCURVES*. *HCRVCRV* does not test if two arcs *must* intersect, but continues the refinement until one of the following two conditions is met: (i) there are no more tagged edges, or (ii) the maximum resolution has been reached. In case (i), C and D do not intersect. In case (ii), each tagged edge of C_{app} is tested for intersection with each tagged edge of D_{app}. If two edges intersect, the intersection points are computed and returned.

We implemented this algorithm on a VAX 8800 with the same modifications as in the case of *HCURVES*. Figures 14 and 15 give two examples for the performance of the algorithm at various maximum resolutions r. P is an intersection point, d is the distance between P and its approximation, C_{app} and D_{app} are C's and D's final approximations, and t is CPU time required to compute all intersections.

Note that the running times do not increase quadratically with the number of edges, 2^r, or with the complexity of the input curves. In fact, the increase in CPU time is about cubical in r, i.e. polylogarithmic in the number of edges. The plot in figure 16 shows the increase in CPU time for both figures and for resolutions r=2 through r=7. The broken lines indicate the distance d between the actual intersection point P and the corresponding intersection point returned by the algorithm at maximum resolution r.

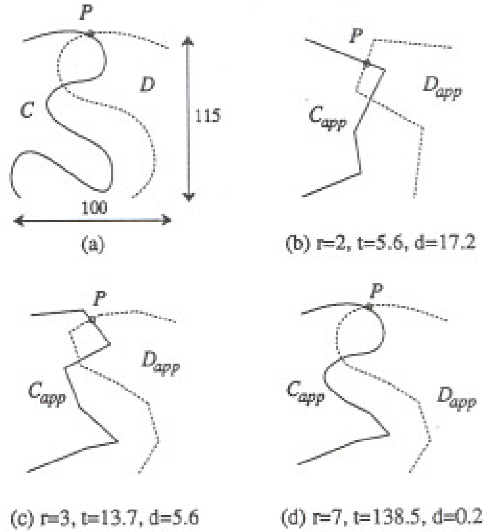


Figure 14: C is a spline with 13 knots, D a spline with 8 knots.

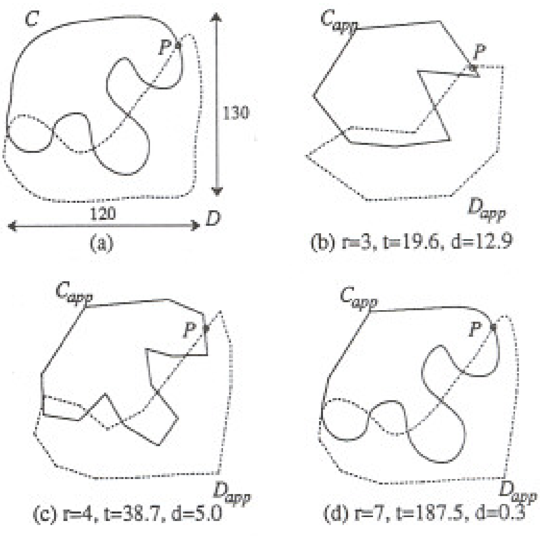


Figure 15: Both C and D are splines with 20 knots.

6. Implementation in a Database System

As the previous sections have shown, the arc tree is an efficient scheme to represent curves. In large-scale geometric applications such as geography or robotics, is is usually most efficient to have a separate data management component and to maintain a geometric database to store a large number of

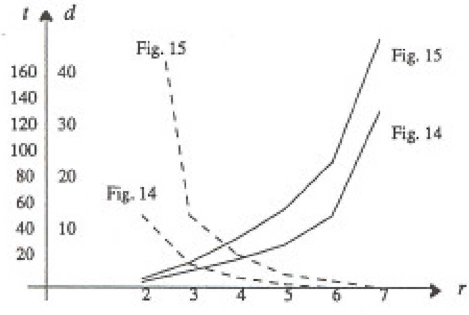


Figure 16: CPU time t and error d of algorithm *HCRVCRV* at various resolutions r.

geometric objects. In order to use the arc tree representation scheme efficiently in this context, it is therefore necessary to embed arc trees as complex objects in the database system.

There are three major ways to implement complex objects in an extensible relational database system such as POSTGRES [Ston86a], DASDBS [Paul87], or XRS [Meie87]. First, one may organize the data of a complex object in relational form and represent the object as a set of tuples, each marked with a unique object identifier. Then the algorithms may be either programmed in an external host language with embedded query language commands [RTI84], or within the database system by means of user-defined data types and operators [Wong85, Kemp87]. These approaches have been used in earlier attempts to extend relational database systems to applications in geography and robotics [Kung84, Gunt87]. Second, one supports a procedural data type to store expressions in the query language or any other programming language directly in the database. This approach is emphasized in the POSTGRES database system [Ston86b]. Third, one may define an abstract data type (ADT) with corresponding operators and abstract indices; see for example [Ston83]. The importance and suitability of ADT mechanisms for geometric data management has also been discussed by Schek [Sche86].

Although the arc tree is a useful representation scheme for the most important geometric operators, it should not necessarily be visible to the user. On the contrary, all set and search operators should be executed *without* revealing the internal representation scheme - the arc tree - to the user. The only operator where the internal representation may be visible to the user is the rendering of approximations of the curve. But even then, it seems preferable to offer an operator that maps an abstract object of type *curve* and a resolution into an approximation of the curve. Note that for none of the common operators the user needs to have explicit access to subtrees or to retrieve or manipulate details of the arc tree. On the other hand, it is important to implement the algorithms for set and search operations as efficiently as possible. The algorithms are complex, and their performance should not be impeded unnecessarily by an insufficient runtime environment or an inadequate implementation language.

Because of these considerations we believe that an embedding of the arc tree as an abstract data type (ADT) into an extensible database system is the superior solution to the problem. An ADT is an encapsulation of a data structure (so that its implementation details are not visible to an outside client procedure) along with a collection of related operators on this encapsulated structure. The canonical example of an ADT is a stack with related operators *new*, *push*, *pop* and *empty*. In our case, the user is given an ADT *curve*; each curve is represented internally as an arc tree, but this fact is completely transparent to the user. For a more detailed discussion of the arc tree embedding into a database system, see [Gunt88].

7. Summary

We presented the arc tree, a balanced binary tree that serves as an approximation scheme for curves. It is shown how the arc tree can be used to represent curves for efficient support of common set and search operators. The arc tree can be viewed as just one instance of a large class of approximation schemes that implement some hierarchy of detail. We gave an overview of several other approximation schemes that are based on the same idea, and indicated how to modify the arc tree algorithms to work with these schemes.

Several examples are given for the performance of our algorithms to compute set and search operators such as point inclusion or area-area intersection detection and computation. The results of the practical analysis are encouraging: in most cases, the computation of boolean operators such as point inclusion or intersection detection can be completed on the first four or five levels of the tree. Also, the computation of non-boolean operators such as intersection computation gives fairly good results even if one restricts the computation to the first few levels. Finally, it is described how to embed arc trees as complex objects in an extensible database system. It is argued that the embedding as an abstract data type is most efficient.

It is subject of future research to conduct a more comprehensive and systematic study of the arc tree algorithms and of the different possibilities to handle arc trees in an extensible database system. Also, we are planning to compare the arc tree to Ballard's strip tree and to Bezier curves, both theoretically and practically.

References

[Ball81] Ballard, D. H., Strip trees: A hierarchical representation for curves, *Comm. of the ACM* **24**, 5 (May 1981), pages 310-321.

[Bezi74] Bezier, P. E., Mathematical and practical possibilities of UNISURF, in *Computer Aided Geometric Design*, Academic Press, New York, NY, 1974, pages 127-152.

[Bohm84] Bohm, W., Efficient evaluation of splines, *Computing* **33** (1984), pages 171-177.

[Burt77] Burton, W., Representation of many-sided polygons and polygonal lines for rapid processing, *Comm. of the ACM* **20**, 3 (March 1977), pages 166-171.

[Debo78] Deboor, C., *A practical guide to splines*, Springer, Heidelberg, 1978.

[Gunt87] Gunther, O., An expert database system for the overland search problem, in *Proc. BTW'87 - Database Systems for Office Automation, Engineering, and Scientific Applications*, Informatik-Fachberichte No. 136, Springer, Berlin, 1987.

[Gunt88] Gunther, O., *Efficient structures for geometric data management*, Lecture Notes in Computer Science No. 337, Springer-Verlag, Berlin, 1988.

[Hopc87] Hopcroft, J. E. and Krafft, D. B., The challenge of robotics for computer science, in *Algorithmic and geometric aspects of robotics*, Advances in robotics, Vol. 1, C. Yap and J. Schwartz (eds.), Lawrence Erlbaum Assoc., Hillsdale, NJ, 1987.