Constructing Partitioning Trees from Bezier-Curves for Efficient Intersections and Visibility

Bruce Naylor and Lois Rogers
AT&T Bell Laboratories
Murray Hill, NJ

Abstract

While a very effective method for designing objects is to describe their surfaces using non-linear parametric representations, these are not necessarily the best for executing all of the computations required for modeling and rendering. These computations include set operations for CSG, collision detection, ray-surface intersections, visible surface determination, shadow calculations, radiosity transfer, and view-volume clipping, all of which involve computing intersections, possibly in visibility order. Intersections are intrinsically easier to compute using the implicit rather than the parametric form; however, implicitizing parametric surfaces can produce polynomials of too high a degree. An alternative that we introduce in this paper is to convert the non-linear parametric form into a piecewise-linear, multi-resolution, implicit form, viz. the Binary Space Partitioning Tree. This provides a hierarchical organization of the many linear pieces, resulting in acceleration of intersection and visibility calculations. Our method yields a definition of non-linear sets as infinite trees which can be adaptively pruned/truncated to produce a finite tree meeting a target approximation error. We describe how to construct such a tree representation of a region of 2-space whose boundary is a piecewise-Bezier curve of any degree. We then describe how this construction can be used in 3-space to build trees representing generalized cylinders. This provides a very effective method for constructing multi-resolution trees that are "good", as measured by expected cost, for intersections and visibility.

Introduction

Computing intersections and visibility between sets are very fundamental operations in Geometric Computation. Intersections (set operations) are used in geometric modeling for constructive solid geometry and interference detection, in dynamics for collision/contact detection, and in rendering for view-volume clipping/culling. Intersections combined with visibility orderings are used in rendering (radiation propagation) for ray-tracing, beam-tracing, shadow volumes, and radiosity transfer calculations. The speed and accuracy at which these computations can be performed depends primarily upon the computational representations of geometry used. Among the various choices for representations, there is a fundamental distinction between the parametric and implicit forms, as well as between linear and non-linear representations. We will argue below that sets represented in the implicit form are intrinsically better suited for intersections than parametrics because they provide directly a set membership function. As for the linear/non-linear dichotomy, computing the intersection between two sets defined using only linear equations, say between two polygons, is simple; while the same computations involving non-linear sets can be far more complicated or even impractical. We will describe a solution to the problem of computing intersections and visibility for the class of solids known as generalized cylinders, which are objects that can be defined using at most three curves: cross-section, profile and path curves. Translational and rotational swept objects are special cases of generalized cylinders. In our method, the curves are defined as piecewise Bezier curves of any degree and continuity, and we use these to produce a multi-resolution, piecewise linear, implicit form, viz. the Binary Space Partitioning Tree (BSP Tree or, as we prefer, Partitioning Tree) [Fuchs, Kedem, and Naylor 80]. These trees then provide the representation that accelerates intersection and visibility calculations.

Currently, the most popular form of non-linear parametric representations is the B-spline form (NURB's) [Farin 88]. These can be easily converted into the piecewise Bezier form by employing the Blossoming algorithm, and so these two forms define the same sets. The computation of intersections between objects defined using parametric splines can be solved analytically if one or both objects is first converted into its implicit form. However, the algebraic degree of bi-parameter patches of parametric degree n is $2n^2$. So for example, bi-cubic patches are of algebraic degree 18, and the intersection curve between two such patches is of algebraic degree $18^2 = 324$. This, of course, makes the algebraic representation of such an intersection curve impractical. A standard approach to reducing the complexity of operations involving non-linear equations is to approximate the sets by a piecewise linear

representation. Linear sets, i.e. those define using only linear equations, have the unique property among algebraic sets: their intersection curves are the same degree as the original sets, i.e. of degree 1. But the cost of piecewise linear approximations is that the number of linear pieces can be quite large if the error due to the approximation is to be maintained at an acceptable level. Thus, using linear approximations constitutes a tradeoff of algebraic complexity for combinatorial complexity. So while the algebraic geometry of intersections has been made simpler, an unorganized set of linear pieces will require $O(n^2)$ operations for both intersection and visibility calculations. This deficiency can be ameliorated significantly by organizing the possibly thousands of pieces into a hierarchical data structure. If in addition, the representation has the multi-resolution property as well, then the number of pieces used for the approximation can be adaptively selected in order to meet a target approximation error.

In this paper, we extend the notion of implicitizing piecewise Bezier curves in 2-space by presenting a method that uses the de Casteljau algorithm for recursive midpoint subdivision to directly generate a 2D Partitioning Tree representing the curve. We will then describe how to extend this method to the construction of trees representing generalized cylinders. In this schema, sets with non-linear boundaries are represented by infinite trees whose paths either converge monotonically to the boundary or terminate inside or outside the set. Tree pruning, analogous to the truncation of an infinite series, yields an approximation whose error is well defined and easily computed, and introduces a multi-resolution character to the representation. In addition, intersections and visibility orderings between sets represented by independent trees can be computed by merging their respective trees [Naylor, Amanatides and Thibault 90]. Tree merging can be interpreted as merging two bounding-volume hierarchies, and so is very efficient. Since the result of merging two trees is also a tree, all essential properties are preserved, including the multi-resolution character. If the Bezier definition has been retained at the appropriate leaves of the tree, it is possible to grow trees dynamically in the region in which their surfaces intersect as part of the tree merging process, i.e. a lazy evaluation schema (we have not yet implemented this).

Comparisons to Alternative Approaches

There has been a considerable amount of prior work addressing the problem of computing intersections and visibility with parametric surfaces, and much of it employs hierarchical subdivision in some fashion. One area of research has focused on the ray-surface intersection problem. One of the earliest solutions used implicitization, resultants and iterative root finding [Kajiya 82]. In [Toth 85] multi-variate Newton iteration is employed, combined with interval arithmetic in order to solve the problem of finding a good initial starting point for the iteration. [Joy and Bhetanabhotla 86] presented a similar approach using quasi-Newton methods, but use ray-coherence and spatial subdivision to facilitate the problem of finding a good starting point. These methods may prove to be effective for, say bi-cubic patches (algebraic degree 18), but they become rapidly less so for higher degrees. Finally, [Nishita, Sederberg and Kakimoto 90] present a method most similar to ours based on Bezier clipping, which for a single ray converges faster than midpoint subdivision.

Rendering without ray-tracing of parametric surfaces has relied upon recursive subdivision, e.g. [Catmull 74]. However, these methods generate a set of polygons that require, for example, a depth buffer for visible surface determination. An exception to this is provided by the scan-line algorithm in [Lane et al 80].

The first intersection operations between two Bezier patches appeared in [Carlson 82]. The method uses recursive subdivision and testing for intersection between pairs of control polyhedra, one for each surface. The intersection curve is represented in the parametric domain of each path using a quadtree, thus providing trimmed parametric patches. [Crocker and Reinke 87] also generate trimmed patches, but they do so by intersecting approximating surfaces composed of piecewise linear or quadratic surfaces. [Herzen, Barr and Zatz 90] construct a hierarchy of "Jacobean bounding volumes" based on bounds on the partial derivatives and test for intersection between bounding volumes. [Snyder 92] applies interval arithmetic to the intersection problem, which is loosely speaking a kind of implicit form, since one is computing membership within boxes. This, in fact, produces an octree decomposition of space, although [Snyder 92] does not produce an explicit octree data structure. However, [Duff 92] does generate an octree using interval arithmetic, although the paper does not specifically address parametrics.

In contrast to these methods, our approach, currently limited to generalized cylinders, generates a global evaluation of the surface once, recording the results in a tree. This tree can then be used repeatedly in geometric computations without re-evaluation, and can be used with trees representing objects generated by alternative methods, such as from physical sensing or implicitly defined nonlinear surfaces. For computing surface-surface intersections, tree merging provides the same benefits as the various hierarchical subdivision methods just described, but with the very important added benefit of

producing a structure from which a visibility ordering can be generated. And unlike axis-aligned schemes, Partitioning Trees can be affinely transformed. Thus, the benefits of the evaluation/tree-construction can be reaped in computing collisions and rendering operations over a possibly large number of frames of an animation or interactive sequence.

Similarly for ray-tracing, the single evaluation is shared by all rays, rather than doing an independent recursive subdivision for each ray. Ray-tracing can also exploit the multi-resolution aspect by adaptively terminating the ray-surface intersection search at a depth determined by the size of the projected surface area and the angle of incidence: the greater the angle of incidence, the deeper the search, since the highest resolution is required at silhouettes. This should offset the fact that the bi-section algorithm inherited of midpoint subdivision does not converge as fast as the Newton-type root finder or Bezier clipping; and we do not have the problem of finding an initial starting point. Also in our approach, when CSG is used with ray-tracing, the set operations are performed once, instead of repeatedly for every frame where every ray is intersected with all primitives surfaces and set operations are performed in "ray-space". This can result in significant savings, since no ray-surface intersections are computed for subsets of primitives that are removed by set operations.

Implicits vs. Parametrics

Geometric sets are commonly specified using continuous functions, and the distinction between parametric and implicit arises from whether the set of interest is contained in the range or the domain of the functions. For parametrics, the set lies in the range of a vector valued function $\mathbf{F: X^m \Rightarrow Y^n}$, where as for implicits, the set lies in the domain and is defined by a function of the form: $\mathbf{G(Y) = 0}$.

The essence of parametric representations is their power to enumerate points in the set **S**. By enumerating points in the parameter space, one generates corresponding points on **S** in the range. If the parameter points are vertices of a topological decomposition **K** of the parameter space, e.g. a triangulation, then applying **F** to **K** produces an embedding of **K** in **S**. This fact is commonly used to generate polygonal approximations of parametrically defined surfaces in 3D. The enumeration property is also used in polygon drawing to enumerate all pixels lying with the projection of a polygon (scan-conversion). In contrast, implicit functions are of the form $\mathbf{G: Y^n \Rightarrow Z^1}$. The set $\mathbf{G(Y) = 0}$ is called a *hypersurface*. We can use this to define solids (in general, Lebesque measurable sets) by defining $\mathbf{S \equiv \{ Y \mid G(Y) <= 0 \}}$. The set is, therefore, defined by a *membership function*, also called a *characteristic function*, which is a boolean-valued total function that is TRUE whenever a point is a member of the set. This then is the essential nature of implicit representations.

The most popular forms of parametric and implicit functions use polynomials. Sets defined implicitly by a single polynomial define the class of sets called *Algebraic Sets*. Hypersurfaces defined parametrically by (rational) polynomial coordinate functions are also algebraic sets, but not all algebraic sets admit such a rational parameterization. Thus, the parametrics are a subset of implicits when only polynomials are involved. Generating the implicit form of a parametric hypersurface is accomplished by a process called *implicitization*. The algebraic degree of an algebraic set is simply the degree of the polynomial of the implicit form, and it gives the maximum number of intersections between a line and the hypersurface.

The distinction between sets for which one has a membership function, as with implicits, and those for which one has an enumeration function, as with parametrics, lies at the very foundations of the theory of computation. In particular, in order for a countable set to be characterized as computable, the set must have a computable membership function. Such sets are called Recursive Sets. Those sets for which there exist computable enumerating (or generating) functions are called Recursively Enumerable Sets. This tell us that the distinction between implicits and parametrics is not an artifact say of using polynomials but corresponds to an important difference in methods of defining sets. Indeed, the difference in the computational efficacy of the two representations can be largely understood in terms of the difference between membership and enumeration functions.

Determining set membership is required for performing any intersection operation. Such operations include point classification, ray-surface intersection, clipping to a view-volume, collision detection, constructive solid geometry, shadow volumes, continuous-space visible surface determination, etc. On the other hand, enumeration is used directly for polygonalization of a curved surface and scan-conversion of polygons. It is also used to compute intersections with an implicit form, as is done when a parametric representation of a line/ray/edge is substituted into an implicit equation of a hypersurface. Various values of the parameter are "enumerated" either analytically if the degree is 1 or 2, or numerically, otherwise. So while computing intersections is an intrinsic property of implicits, a parametric can quite effectively be paired with an implicit to perform such operations. Intersections between two implicits can be computed symbolically by using Resultants or Grobner Bases, but no such comparable algorithms are known for two

parametrics, although there do exist tracing algorithms for this case. (Here we are considering recursive subdivision schemes which rely on testing for intersection of bounding volumes, including intervals, to be using implicits, since plane equations are used.)

Historically, the distinction between implicits and parametrics has been focused on sets defined by a single $\mathbf{C^1}$ function. However, given the importance of piecewise representations, it is natural to extend the notion to representations using many such functions. This is already common practice in the case of B-splines, which are considered parametrics even though they are only piecewise-polynomial. We can also consider boundary representations of linear polytopes as being a parametric form. This is easy to see by first noting that when one restricts the degree of B-splines to be 1, the result is a boundary representation (although typically with limitations on the topology of the pieces, say to being 4-sided). It is also common to parameterize polygons in b-rep form by specifying texture coordinates at the vertices. In keeping with this view, we can then interpret Partitioning Trees as an implicit representation, since it is composed of a set of linear polynomials (hyperplanes). Therefore, the topic of this paper, constructing Partitioning Trees from Bezier curves, can be interpreted as a type of implicitization of parametric curves, in particular, a linear, multi-resolution implicitization. However, before describing this construction, we first need an understanding of what kinds of trees are desirable.

Good Partitioning Trees

Unlike topological representations, any given set may be represented by an arbitrary number of different Partitioning Trees. This is analogous to the fact that for any computable function, there is a countably infinite number of syntactically different programs that compute that function. Indeed, a Partitioning Tree may be interpreted as a computation graph that specifies a particular search of space. Analogous to the property that not all programs/algorithms are equally efficient, not all searches/trees are equally effective. Thus the question arises as to what constitutes *goodness* for Partitioning Trees (computing or even defining the optimal is very problematic). In [Naylor 93], the notion was introduced of identifying good trees with those that represent the set as a sequence of approximations, or actually a tree of approximations, and which consequently provide a multi-resolution representation. Various approximations of the set can be created by pruning the tree at various depths. This is analogous to the pruning of decision trees used in machine learning.

By way of an introduction to this notion of goodness, we show in Figure 1 two quite different ways to represent a convex polygon, only the second of which employs the sequence of approximations idea. The tree on the top subdivides space using lines radiating from the polygonal center, splitting the number of faces in half at each step of the recursive subdivision. The hyperplanes containing the polygonal edges are chosen only when the number of faces equals one, and so are last along any path. If the number of polygonal edges is **n**, then the tree is of size **O(n)** and of depth **O(log n)**. In contrast, the lower tree uses the idea of a sequence of approximations. The first three partitioning hyperplanes form a first approximation to the exterior while the next three form a first approximation to the interior. This divides the set of edges into three sets. For each of these, we choose the hyperplane of the middle face by which to partition, and by doing so refine our representation of the exterior. Two additional hyperplanes refine the interior and divide the remaining set of edges into two nearly equal sized sets. This process proceeds recursively until all edges are in partitioning hyperplanes. Now this tree is also of size **O(n)** and depth **O(log n)**, and thus the worst case for point classification is the same for both trees. Yet they appear to be quite different.

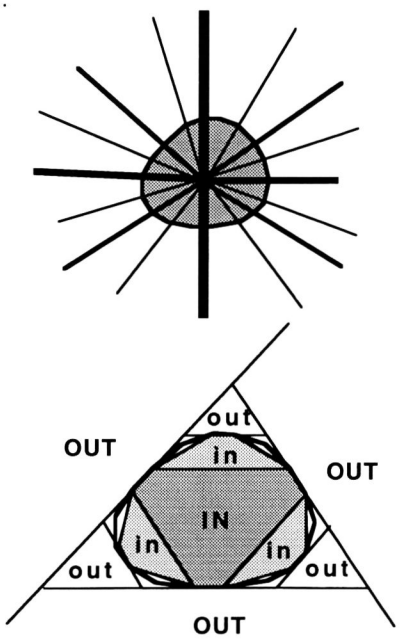


Illustration of bad vs. good trees
Figure 1

This apparent qualitative difference can be made quantitative by, for example, considering the expected case for point classification. With the first tree, all cells are at depth **log n**, so the expected case is the same as the worst case regardless of the sample space from which a point is chosen.

However, with the second tree, the top three out-cells would typically constitute most of the sample space, and so a point would often be classified as OUT by, on average, two point-hyperplane tests. Thus the expected case would converge to O(1) as the ratio of polygon-area/sample-area approaches 0. For line classification, the two trees differ not only in the expected case but also in the worst case: **O(n)** vs. **O(log n)** (in the bad tree, a query line will intersect all n radiating lines, but with the good tree, the process becomes a search for the two intersection points between the line and the polygon, each of which takes **log n**). For merging two trees (e.g. set operations) the difference is $\mathbf{O(n^2)}$ vs. **O(n log n)**. This reduces to **O(log n)** when the objects are only contacting each other, rather than overlapping, as is the typically the case for collision detection.

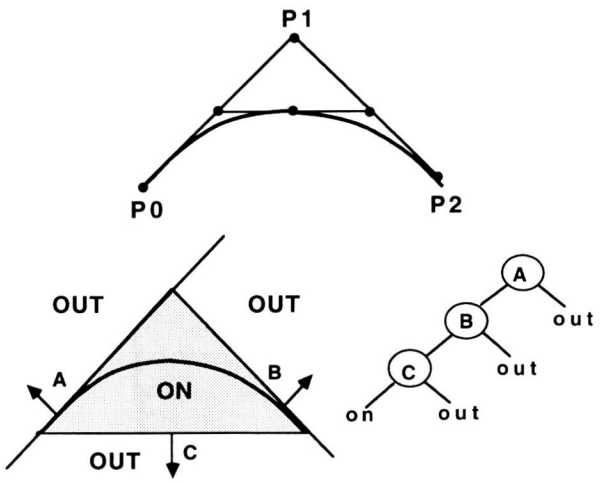
The subject of good trees is explored in greater depth in [Naylor 93], and the main algorithmic result there is a method for constructing good trees from a boundary representation using expected case models to drive a relatively time consuming search over the space of possible trees. What we will describe in the rest of this paper is a schema in which good trees are created directly from the Bezier form without any expensive searching. This represents a substantial improvement in the time required to produce good, i.e. multi-resolution, trees. To accomplish this, we will exploit the well known fact that the control polygon of Bezier curves provides an approximation of the curve segment and that recursive subdivision of the control polygon produces a tree of control polygons that converges to the curve.

2D Bezier Curve -> Partitioning Tree

We begin with the simplest component of our schema, which is to take a 2D parametric curve defined by a set of Bezier control points and produce a Partitioning Tree that will be, in our extended sense, an implicit form of the curve. To obtain an implicit representation, we must use the curve to induce a classification on space. Usually this means creating a membership function that assigns **in** or **out** to every point, which in the case of trees, means classifying the cells of the partitioning into in-cells and out-cells. However, this will not suffice for representing non-linear sets by linear sets, since the later can only approximate the former. Instead of using the usual 2-valued logic, we use a 3-valued logic in which the third value will be **on**. This will be used to denote a region of space in which the boundary of the set is known to lie, but in which we do not yet know exactly where it lies, at least in terms of the tree representation. An on-region can be interpreted as a region of uncertainty with regard to the classification of points into **in** and **out**. In contrast, in-cells and out-cells are regions of certainty. However, this uncertainty can be reduced to an arbitrarily small amount through recursive subdivision of the Bezier curve. Note that the **on** value, while it indicates the presence of the (d-1)-dimensional boundary, will be used as a classification of a d-dimensional region, i.e. the same dimension as the **in** and **out** regions, and so all three types are treated uniformly.

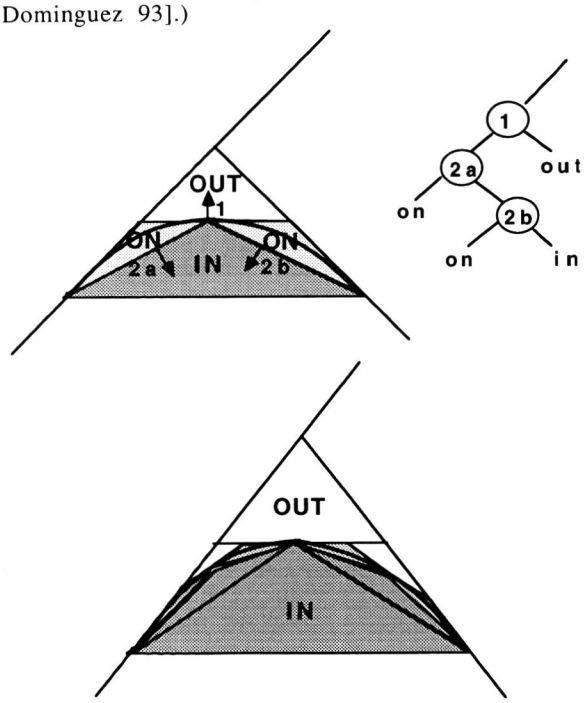
For our Bezier curve to tree conversion, there is no restriction on the parametric degree, i.e. on the number of control points. But we do require for the construction given below that the curve segment be convex. A sufficient though not necessary condition for convexity is for the control polygon to be convex, a consequence of the variation diminishing property of Bezier curves. If this is not the case, and the curve/control-polygon does not self-intersect, we can subdivide the curve into convex pieces by splitting the curve at every inflection point (0 curvature). The parametric values of the inflection points can be found numerically with recursive subdivision: simply perform midpoint subdivision until all sub-curve control polygons are convex. Inflection points occur between two consecutive control polygons that do not "agree in sign", i.e. that do not when taken together form a convex polygon. We can then perform a single subdivision for each inflection point, taken in increasing parametric order, to obtain a collection of convex curve segments.

In Figure 2, we show on the top a quadratic Bezier curve specified by 3 control points, along with a single midpoint subdivision step. In the remainder of Figure 2, we show the beginnings of the corresponding Partitioning Tree. The initial tree is a bounding triangle which completely contains the curve segment and whose vertices are the control points. We assign to the outside of the bounding triangle the classification of **out**, and within the triangle we assign the classification of **on**.



Quadratic Bezier with initial tree
Figure 2

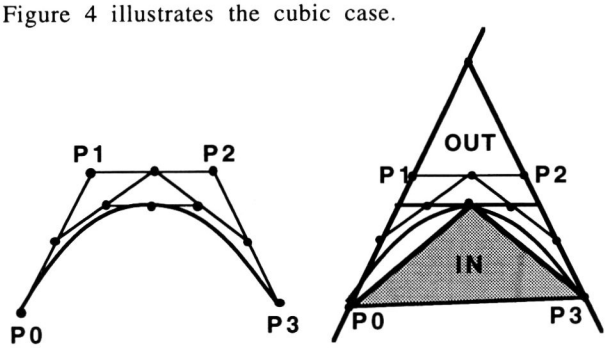
Any on-region containing a Bezier control polygon can be refined by a single application of the midpoint subdivision algorithm to the control points. This produces the parametric midpoint of the curve, the tangent line through this point, and the two sets of control points for each half of the curve. Since the curve segment is convex, it lies completely to one side of the tangent line. We can add a node to the tree, as shown in Figure 3, with the tangent line as the partitioning hyperplane so that only one child region contains the curve, and this region will be designated by convention as an out-cell. Now for each half of the split curve, the two tangent lines at its endpoints are already partitioning hyperplanes of the tree. We can complete a bounding triangle for the segment by simple adding to the tree the line connecting the two endpoints. Doing this for both curve-halves also creates a fourth triangular region that lies completely to one side of the original curve segment, and we choose to designate it as an in-cell. We have thus refined our knowledge about the location of the boundary and have created two independent curve segments lying in on-regions. We can recursively apply this process of subdividing the curve and adding tree nodes indefinitely. The lower illustration in Figure 3 shows one additional level in the process. (For a somewhat similar approach, see [Gunther and Dominguez 93].)



Tree representation of midpoint subdivision
Figure 3

The tree produced by this process, which we refer to as a *segment-tree*, is a multi-resolution, piecewise-linear, implicit representation of the Bezier curve segment. And when we interpret the tree as a computation graph, the tree is in some sense a "remembering" of the recursive subdivision. We can also define an infinite tree that converges to the curve and which can be truncated at any level of the tree. For any given truncation, an absolute measure of the error of the representation introduced by truncation is the sum of the areas of all on-regions, which in the limit is 0. We call this metric a *covering metric* since we are measuring the size of a collection of disjoint open sets that cover a lower dimensional set. This metric is much simpler to compute than the Hausdorff metric and more accurate than max-distance metric. It is also similar to the definition of the Hausdorff-Besicovitch dimension used to define the fractal dimension of a set[1].

For higher degree curves, we have two choices for constructing a tree, which in the quadratic case are identical. Either we can "remember" every step in the subdivision process by adding a degree dependent number of tree nodes, or we can imitate the quadratic case by adding only three nodes per subdivision, a possibility afforded by the fact that our construction depends only upon convexity. We have chosen the later, since it produces smaller trees, and makes the tree construction process independent of the degree of the curve. This independence will prove useful later when we discuss the construction of generalized cylinders. Figure 4 illustrates the cubic case.



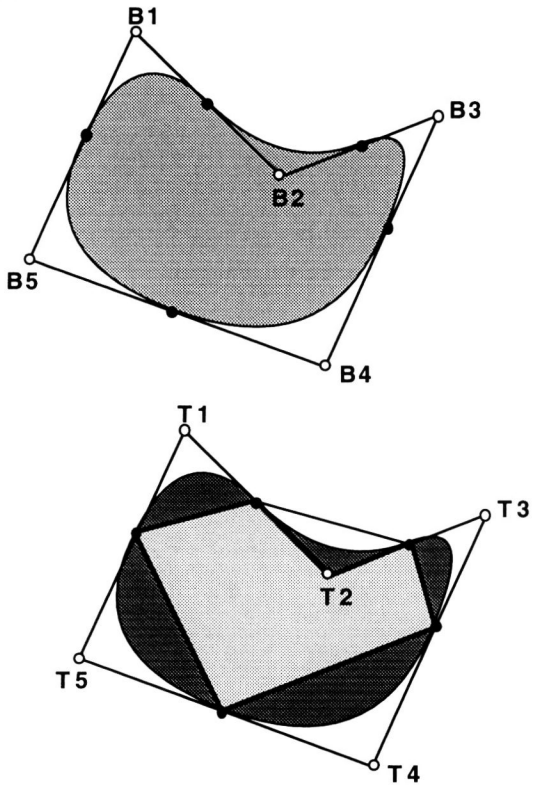
Cubic Bezier subdivision with tree
Figure 4

Piecewise-Bezier -> Partitioning Tree

In Figure 5, we illustrate a piecewise Bezier quadratic curve that is $\mathbf{C^1}$. The set of control points are of two types: those located at the segment endpoints, indicated by solid circles, and those that may be considered as "internal", indicated by hollow circles. The internal control points are in fact the same points as the set of B-spline control points needed to define the same curve, and are labeled with **Bn**. In the Bezier form, $\mathbf{C^1}$ can be maintained by constraining the segment endpoints to lie on the

[1] The fractal dimension is the exponential rate at which the number of elements in a covering varies with the radius of the open sets used in the covering.

line determined by the two intervening internal control points. Movement of these control points has the effect on the corresponding B-spline definition of changing the knot spacing (the "non-uniform" aspect of NURB's). A somewhat analogous correspondence between a B-spline definition and a piecewise Bezier definition exists for all degrees and may be obtained through the use of the Blossoming algorithm. Thus, the original curve definition may be a NURB, even though our method of constructing segment-trees requires the Bezier form. However, for low degree curves, we have actually found that the Bezier form to be more intuitive as a user interface, since it replaces adjusting knot spacing with manipulating control points. It is also very easy for the user to specify segments differing in degree within the same curve.



Bezier specification of a region of 2-space
Figure 5

Given a piecewise Bezier definition of a curve that is closed and does not self intersect, we will now address building a single tree representing a region of 2-space whose boundary is such a curve. The idea is to build individual segment-trees for each convex Bezier curve segment (see Figure 5 lower half), and then merge these trees together to form a single tree. There are three parts to this. The first is to apply the methods above to produce a collection of segment-trees. The second is to determine for each segment whether it is a "convex" segment or "concave" segment; i.e. whether the internal control points lie in the exterior of the set or in the interior. The later case, which occurs in Figure 5 for T2, will require the segment tree to be complemented (actually, just the subtree inside the outer most bounding triangle). The third part is to "fill in" the interior of the region that lies inside the B-spline control polygon, but does not lie within any segment tree (indicated in Figure 5 by the lightly shaded region). The vertices of this polygonal region are a subset of the Bezier control points. They are either the curve segment endpoints, if the segment-tree is uncomplemented, or the internal control points of complemented segment-trees. We construct a tree for this region using the standard b-rep -> tree conversion routine, and then union the result with the collection of segment-trees. This yields the desired single tree representing the region of 2-space bounded by the curve.

Tree Representation of Generalized Cylinders

We will now address applying the foregoing 2-space curve representation schema to the problem of creating 3-space objects. In particular, we will describe how to construct a good tree for the class of objects called *generalized cylinders* (for a recent paper on these, see [Snyder and Kajiya 92]). This class is obtained by extending the ideas embodied in swept objects. They constitute a rich class of objects, which include rotational and translational sweeps, and they are very easy to design since they are specified using a few 2-space curves "drawn" on a flat surface. When combined with set operations, many commercially manufactured objects can be designed using generalized cylinders as the primary "free form" modeling primitive.

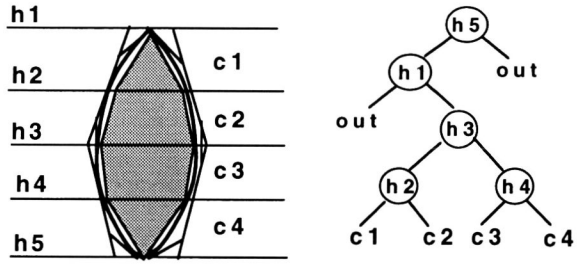
The simplest member of the class of generalized cylinders is the translational swept object. This is the set defined by "sweeping" a 2-space curve along some fixed direction in 3-space. Such an object can be defined by "placing" the curve in the xy-plane and letting the sweeping occur along the z-axis. All other translational sweeps can be obtained by affinely transforming this form. To generate a Partitioning Tree of such an object, all that is needed is to take the 2-space tree created from a curve, treat it as an infinite cylinder, and intersect it with two halfspaces orthogonal to the z-axis corresponding to the limits of the sweep. This can be accomplished simply by making the two bounding halfplanes the first two nodes of the tree and then placing the curve tree "in-between" these by attaching it as the "in-child" of the second plane.

It is customary to employ three curves to define generalized cylinders. The first corresponds to the curve used to define translational sweeps, and is often called the *cross-section-curve*. The

second replaces the axis of translational sweeps with a curve, called the *path-curve*, which is usually constrained so that the object does not self-intersect. The third provides scaling of the cross-section curve as it is swept along the path, and is often referred to as the *profile-curve*. As an example of this, rotational swept objects (surfaces of revolution) are specified by a circle for the cross-section, a straight line for the path (axis of revolution), while the profile plays the traditional role of the user specified curve to be revolved.

To build trees of generalized cylinders, we will begin by describing the construction of a subclass in which we restrict the path-curve to be a straight line, as with surfaces of revolution. Thus, we are only concerned with the cross-section and profile curves. The approach we take is to, in effect, simulate the sweeping of the cross-section curve along the z-axis, which can also be thought of as the time axis of the sweeping process. Since we are constructing a piecewise linear approximation, we will sample time (z) at a finite number of points and interpolate between these. The value of the swept cross-section curve at each sampled point is obtained by translating the curve to the plane orthogonal to the z-axis containing the sample, and then scaling the curve symmetrically in xy by an amount equal to the distance of the profile curve from the z-axis. To create the swept surface, we then need to interpolate linearly between these transformed cross-section curves.

Now to build a tree representing this object, we simply emulate what we have just outlined using an approach first described in [Ihm and Naylor 91] (however, in that work the cross-section curve was derived from a digitized representation of the curve, rather than a Bezier definition). We first generate a linear approximation of the profile-curve using the usual Bezier subdivision. The z-values of the resulting vertices will serve as our sample points. We then build a good tree representing this subdivision of the z-axis into intervals, creating a collection of horizontal planes subdividing 3-space. This tree will be the "top" part of the entire tree (see Figure 6). Within each interval, we will place a 3-space cross-section subtree that interpolates between the two transformed cross-sections lying in the z-planes which bound the interval. This subtree can be generated by applying an inverse perspective transform to the original untransformed cross-section tree lying in the xy-plane. This transform will map the planes of this tree, which are all orthogonal to the xy-plane, to a set of planes each of which contains a specified center of projection. This center of projection lies on the z-axis, and is found by intersecting with the z-axis the line containing the profile-curve line segment corresponding to the target interval. The projection plane is the xy-plane. The result can then be translated and scaled so that the cross-section curve lying in the xy-plane is mapped to the lower of the two swept versions of the curve.

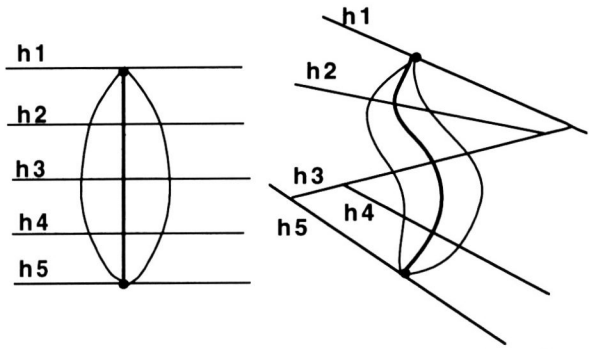


Tree representation of a generalized cylinder (undeformed case)
Figure 6

However, this method does not allow for independent adaptive subdivision of the two cross-sections required to reflect their differing sizes. If this is desired, then we must approach the problem differently while not allowing the introduction of "cracks". To do this, we must repeat the cross-section tree generation algorithm for each 3-space cross-section independently. We will now need to generate segment-trees that interpolate between two transformed convex Bezier segments. Therefore, we transform two copies of the Bezier control points as determined by the target profile segment and simultaneously subdivide both curves adaptively. The partitioning hyperplanes, which in the 2-space case where determined by two control points, must now contain four control points, two from each curve. However, all four points are coplanar (otherwise the first method using the perspective transformation would not work) and so a single partitioning plane is produced (from any three of these points), analogous to the 2-space case. Now since adaptive subdivision permits the two curves to terminate subdivision at differing levels, we must deal with cracks. This can be achieved by only a slight modification to what we have described. We will continue to, in effect, subdivide the terminated curve, but using only the linear form, i.e. two control points. We will also "bias" the choice of which three control points to use to produce the partitioning hyperplane. These will be the two control points from the unterminated curve plus the control point from the other "linear curve" generated by midpoint subdivision. This will produce a triangular face of the object instead of a quadrilateral one produced when both curves are being subdivided.

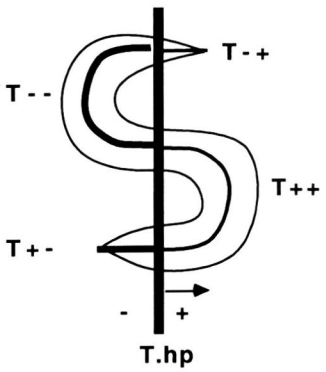
We are now prepared to augment this tree construction method in order to permit more general path-curves, and so obtain the class of generalized cylinders. We will interpret the effect of the path curve as specifying a "spinal deformation" of an object whose path curve (spine) is a straight line, i.e. as deforming the type of object whose construction we have just described. The path-

curve will be interpreted as defining a special kind of deformation of space. In particular, we will define it in terms of a mapping of horizontal planes to a set of planes whose normals are tangent to the path-curve (see Figure 7 for a hand-drawn illustration). More precisely, we treat the undeformed spine as having a uniform parameterization, say **P(t)**. For each value of **t**, we can compute a corresponding point on the path-curve, **Q(t)**, and the tangent vector **Q'(t)**. We then define the mapping to be the 3-space translation and rotation which maps **P(t)** to **Q(t)** and **P'(t)** to **Q'(t)** (of course **P'(t)** is always the z-direction).



Spinal deformation of generalized cylinder
Figure 7

The spinal deformation can be easily applied during the segment-tree generation process. Since our control points have been constrained to lie in horizontal planes, we can apply the above transformation directly to these and proceed as before, with one exception. Whereas previously, we could generate a single partitioning hyperplane that would contain four control points, forming a quadrilateral, the deformation obviates this property, and so in general we must generate two partitioning hyperplanes, each containing three of the control points forming two triangles. An additional issue arises whenever the spinal deformation results in parts of the object lying on both sides of what was in the undeformed case a horizontal plane. This case can be handled using tree partitioning and merging algorithms [Naylor, Anamatides and Thibault 90]. For any formerly "horizontal" tree node **T**, its two deformed subtrees **T-** and **T+**, can be partitioned by **T.hp**, producing four trees, **T--**, **T+-**, **T-+** and **T++**. The new **T-** is created by merging **T--** and **T+-**, and similarly for the new **T+** (see Figure 8).



Treatment of "extreme" deformation
Figure 8

Examples

We now show a few examples of our work. In Picture 1, we illustrate the design of a mushroom. Shown are the three curves, in this case all rational quartic curves, defining the generalized cylinder as well as the resulting object. The lines on the object show the edges of the polyhedral faces, which also constitute all of the intersections between the tree and the boundary of the object. Notice the absense of split faces. Picture 2 shows another object defined using rational cubic curves, two of which contain inflection points, and where only C^0 is maintained. If one looks closely, it is possible to discern where cracks are avoided by using triangular rather than quadrilateral faces. Pictures 3 and 4 give other examples. In Picture 4, the radish body and stem are two separate objects unioned together.

Future Work

The work presented here is based on curves. Individual surface patches are not explicitly created and manipulated but rather are defined as the cross-product of two curves. The obvious next extension to this work is to apply the ideas presented here to arbitrary surface patches. We have considered this issue at sufficient length to know that solutions exist, but we have yet to begin implementations. A second and very important avenue for improving this work it to provide the ability to continue Bezier subdivision at a leaf node on demand. With such a capability, the tree merging algorithm could refine the approximations of two surfaces in the neighborhood of their intersection curve to whatever extent was needed to meet the desired error tolerance.

References

[Carlson 82]
Wyane E. Carlson, "An Algorithm and Data Structure for 3D Object Synthesis Using Surface Patch Intersections", **Computer Graphics** Vol. 16(3), pp. 255-263, (July 1982).

[Catmul 74]
Edwin Catmul, "A Subdivision Algorithm for Computer Display of Curved Surfaces", Ph. D. Thesis in Computer Science, University of Utah (July 1974).

[Crocker and Reinke 87]
Gary A. Croker and William F, Reinke, "Boundary Evaluation of Non-Convex Primitives to Produce Parametric Trimmed Surfaces", **Computer Graphics** Vol. 21(4), pp. 129-136, (July 1987).

[Duff 92]
Tom Duff, "Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry", **Computer Graphics** Vol. 26(2), pp. 131-138, (July 1992).

[Farin 88]
Gerald Farin, *Curves and Surfaces for Computer-Aided Geometric Design*, **Academic Press** (1988).

[Fuchs, Kedem, and Naylor 80]
H. Fuchs, Z. Kedem, and B. Naylor, "On Visible Surface Generation by a Priori Tree Structures," **Computer Graphics** Vol. 14(3), pp. 124-133, (June 1980).

[Gunther and Dominguez 93]
Oliver Gunther and Salvador Dominguez, "Hierarchical Schemes for Curve Representation", **IEEE Computer Graphics and Applications** Vol. 13(3), pp. 55-63, (May 1993).

[Herzen, Barr and Zatz 90]
Brian Von Herzen, Alan H. Barr and Harold R. Zatz, "Geometric Collisions for Time-Dependent Parametric Surfaces", **Computer Graphics** Vol. 24(4), pp. 39-48, (July 1990).

[Ihm and Naylor 91]
Insung Ihm and Bruce Naylor, "Piecewise Linear Approximations of Curves with Applications," Proceeding of Computer Graphics International '91, Springer-Verlag (June 1991).

[Joy and Bhetanabhotla 86]
Kenneth I. Joy and Murthy N. Bhetanabhotla , "Ray Tracing Parametric Surface Patches Utilizing Numerical Techniques and Ray Coherence", **Computer Graphics** Vol. 20(4), pp. 279-285, (August 1986).

[Lane et al 80]
Jeffery Lane, Loren Carpenter, Turner Whitted and James Blinn, "Scan Line Methods for Displaying Parametrically Defined Surfaces", **CACM**, Vol. 23(1), (Jan. 1980).

[Kajiya 82]
James T. Kajiya, "Ray Tracing Parametric Patches", **Computer Graphics** Vol. 16(3), pp. 245-254, (July 1982).

[Naylor, Amanatides and Thibault 90]
Bruce F. Naylor, John Amanatides and William C. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations", **Computer Graphics** Vol. 24(4), pp. 115-124, (August 1990).

[Naylor 93]
Bruce F. Naylor, "Constructing Good Partitioning Trees", **Graphics Interface '93**, Toronto CA, pp. 181-191, (May 1993).

[Nishita, Sederberg and Kakimoto 90]
Tomoyuki Nishita, Thomas W. Sederberg and Masanori Kakimoto, "Ray Tracing Trimmed Rational Surface Patches", **Computer Graphics** Vol. 24(4), pp. 337-345, (August 1990).

[Snyder 92]
John M. Snyder, "Interval Analysis for Computer Graphics", **Computer Graphics** Vol. 26(2), pp. 121-130, (July 1992).

[Snyder and Kajiya 92]
John M. Snyder and James T. Kajiya, "Generative Modeling: A Symbolic System for Geometric Modeling", **Computer Graphics** Vol. 26(2), pp. 369-378, (July 1992).

[Toth 85]
Daniel Toth, "On Ray Tracing Parametric Surfaces", **Computer Graphics** Vol. 19(3), pp. 171-179, (July 1985).

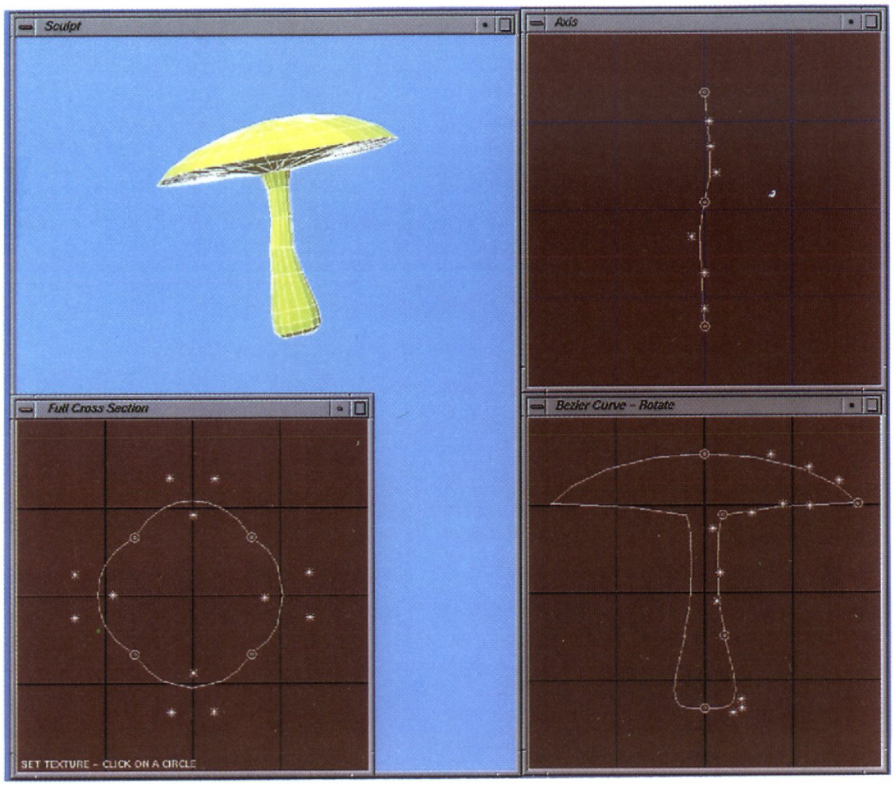
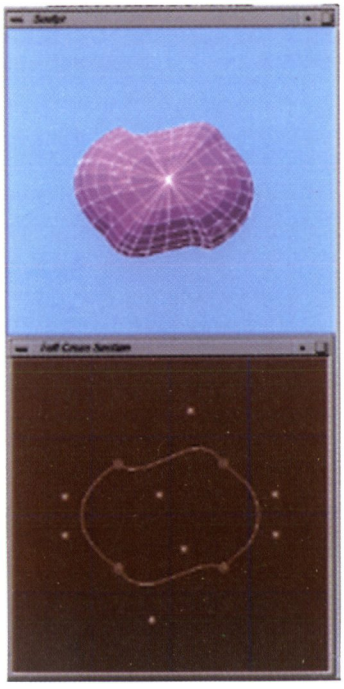


Figure 1



Above: A sample workspace shows an object being modeled. Its cross section, side view and axis are each being defined using quartic Bezier curves.

Left: Bezier curve segments with inflections can appear in the cross section, as shown here, or in the side view or axis.

Figure 2

Figure 3

Pumpkin and Radishes

The radish tops were created separately from the bottoms and later joined to make the radish object.

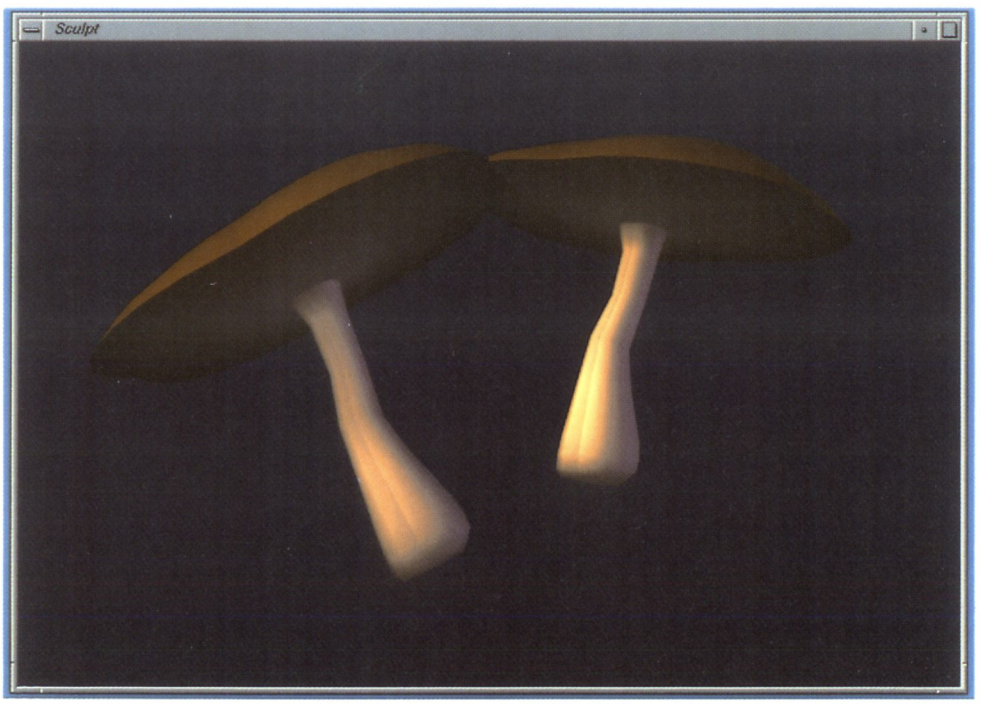


Figure 4

Two Mushrooms

The illustration of the sample workspace shows the mushroom object in the process of being created.