Introduction to Curves

The simplest graphic "object" we deal with in this course is the **point**, identified completely by giving its (x, y) or (x, y, z) coordinates in some coordinate system. The next most complicated geometric structure is the **straight line segment**, specified most easily by giving the coordinates of its two end points.

We can get more general geometric shapes by simply combining line segments to give what is commonly called a **polyline** or something similar. (As you'll see before this course is done, in some sense computer graphics in practice deals with nothing more complicated than a polyline!). However, only a relatively small percentage of geometric shapes that arise in computer graphics are adequately represented by sequences of a few straight line segments (or surfaces composed of an array of a few planar facets -- a further generalization in three dimensions), and so the problem of generating representations of curves and curved surfaces arises.

Straight line segments are easy to construct and unambiguous. Once you've specified the locations of the two endpoints, there are very efficient algorithms for generating the coordinates of a sequence of pixels between them giving the image of a unique straight line. On the other hand, simply knowing the coordinates of a few points on a curve does not generally give us enough information to unambiguously create the rest of the curve. This is one major problem with which we must deal.

Images of curves on a video screen are usually constructed of a sequence of many straight line segments, each short enough that the overall image is perceived as a smooth curve. So, one could perhaps think of specifying a curve by specifying a large number of closely spaced points along that curve. This has the disadvantages of requiring some way to compute the coordinates of that large set of points (so we're back to the first major problem described above), of requiring the storage and manipulation of possibly large amounts of data, and probably most critically, of being ill-suited to changing the scale of the drawing. If such a "curve" were magnified, eventually the short line segments would become visible as straight line segments, and the quality of the image of the curve would become unacceptably crude. We need some way of representing the shape of the curve in terms of a relatively small amount of easily manipulable information, from which we can produce the image of the curve to some degree of precision whenever required.

A variety of methods have been developed for generating formulas or equations of curves based on the locations of a relatively small number of **control points**. Once equations are obtained, the image can be scaled and manipulated without the quality of the image deteriorating.

Equations for some familiar basic geometric curves are well-known (for example, for circular arcs, elliptical arcs, parabolas, etc. -- though the equations you find for these standard curves in textbooks may not always be very computationally efficient). However, most curved shapes in life (hence in computer graphics!) are not simple circular arcs, etc. Instead, we may either have an approximate but non-standard shape in mind, or we may wish to "play around" with the shape until some esthetic quality is achieved, or the shape may be something that arises "on the fly" to match the shapes and positions of other objects in the image (as in the development of movement paths in animation).

In this document, we will look at several approaches to the representation of curves in two-dimensions. The most useful of these methods generalize in obvious ways to the specification of curves in three dimensions. Further, the specification of curved surfaces in three dimensions is most commonly done by specifying a grid of curves and so the results presented here are preliminary to the development of methods for representing curved surfaces as well.

Straight Line Segments

We have discussed various formulations of the equation of a straight line in two or three dimensions in BASGEOM.DOC -- formulas from which we can obtain the y-coordinate of any point on the line segment by substitution of its x-coordinate. In two dimensions, the equation of a straight line can be written in the **algebraic form**

$$y = a_0 + a_1 x \qquad \text{(CURV0-1)}$$

where a_0 and a_1 are constants for the line and x is a variable, or, using a **parametric form**, as,

$$x = a_0 + a_1 t$$
$$y = b_0 + b_1 t \qquad \text{(CURV0-2)}$$

where again, a_0, a_1, b_0, and b_1 are constants and t is a variable. The property of these formulas that associates them with straight line graphs is that all variables appear to the first power only. We say that (CURV0-1) is a linear equation for y in terms of x, and we would say that (CURV0-2) is a set of linear equations for x and y in terms of t.

Polynomials

The fact that the equation, (CURV0-1), for a straight line contains two general constants to be determined -- a_0 and a_1 -- is a consequence of the geometrical fact that we require the location of two points to nail down a particular straight line. Thus, from a mathematical point of view, it would seem that the equation of what might well be a curve through three given points need just have an additional term carrying an additional constant. The simplest new term would be one involving x^2, since $x^0 = 1$ and x^1 have already been taken.[1] So, the equation of a curve passing through, or **interpolating**, three given points is expected to have the general form

$$y = a_0 + a_1 x + a_2 x^2 \qquad \text{(CURV0-3)}$$

And, indeed, this can be made to work, as illustrated in the following (useless) example.

Example:
Find the equation of the curve passing through the points (1,1), (2, 4) and (5, 2).

Solution:
If the form (CURV0-3) is to be the equation of the curve passing through these three points, then the (x, y) coordinates of these points must satisfy (CURV0-3). Substituting in these values of x and y, we end up with three equations:

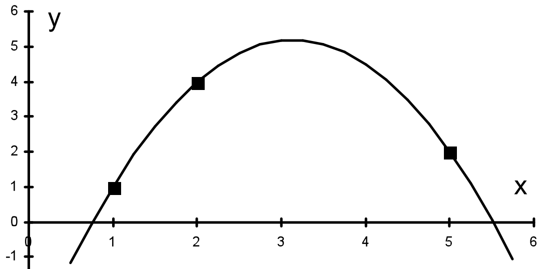

Figure 1

$$1 = a_0 + a_1 + a_2 \qquad \text{using } x = 1, y = 1$$
$$4 = a_0 + 2a_1 + 4a_2 \qquad \text{using } x = 2, y = 4$$
$$2 = a_0 + 5a_1 + 25a_2 \qquad \text{using } x = 5, y = 2.$$

But this is just a set of three linear equations for the three unknown coefficients, a_0, a_1, and a_2 . Solving this system of equations in any valid manner gives the solution $a_0 \cong -3.83333$, $a_1 = 5.75$, and $a_2 \cong -0.91667$, to five decimal places when not exact. Thus, the curve which goes through these three points has the equation

$$y \cong -3.83333 + 5.75x - 0.91667x^2$$

[1] Since only x^0 and x^1 are taken, you might ask, why should the next choice available be x^2 instead of any other power of x, including powers that involve decimal fractions or negative numbers. Good question. In fact, the Excel workbook used to set up Figure 1 above (called threepoint.xls) allows you to substitute other powers than 2 for the third term in (CURV0-3). Doing this changes the shape of Figure 1, but otherwise seems to work. Without getting into a lot of mathematical lore here, the short answer to preferring x^2 is that powers which are negative or decimal fractions cause problems if we allow x to be zero or negative. The higher the power, the more widely (or wildly, if you like) the curve tends to swing between interpolating points. Thus, selecting the next higher whole number power is the best possible choice in this situation.

Figure 1 shows a plot of the three given points (the heavy black boxes), and of the formula just above. If you look closely at the figure, you can just see that the curve consists of a sequence of short straight line segments, each covering a horizontal interval of width 0.2 on this scale. To produce this image on screen, we first need to calculate the (x, y) coordinates of the left-most point and issue a MoveTo() call to that location. Then for successive values of x (in increments here of 0.2), we compute y from the formula and issue a LineTo() call to the new point (x, y). Details will vary slightly from one programming environment to another. In the simplest programs, the computation of (x, y) coordinates from the formula will be done using floating point arithmetic, with the results having to be rounded to whole number values for the calls to the MoveTo() and LineTo() functions in some cases. If high performance was crucial, one might consider looking at ways to achieve the same effect without resorting to floating point arithmetic, but such techniques go beyond the scope of this course.

You see that the curve produced does indeed pass through the three given points, though it's kind of difficult to tell from the brief information available if the shape of curve resulting is actually the sort that might have been of use here. In particular, the curve in Figure 1 seems to have a prominent peak between the second and third points which is otherwise not hinted at from the locations of the three points themselves. We can visualize many other curves that pass through the three given points, underlining how arbitrary the curve picture here really is.

◆◆◆

Formulas of the type (CURV0-3), consisting of a constant term plus terms involving successive powers of x are called **polynomials**. The approach illustrated above suggests that to get the formula of a curve through n points, we need to assume a polynomial with powers of x up to the (n-1)th power. The numerical coefficients in the formula can be obtained by solving the system of n linear equations which results when the coordinates of each point are substituted in turn into the template formula, as we did above in the case n = 3. In fact, even this process is not necessary, because an equivalent formulation of the problem is given by the so-called Lagrange Interpolation formula:

$$y = \sum_{k=1}^{n} L_k(x) \cdot y_k \qquad \text{where} \qquad L_k(x) = \prod_{\substack{j=1 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j} \qquad \text{(CURV0-4)}$$

Here, (x_k, y_k) are the coordinates of the kth point (k runs from 1 to n), Σ is the standard symbol for summation,

$$\sum_{k=1}^{n} L_k(x) \cdot y_k = L_1 \cdot y_1 + L_2 \cdot y_2 + L_3 \cdot y_3 + \ldots + L_n \cdot y_n\,,$$

and Π is the standard symbol for a product in this context:

$$L_k(x) = \prod_{\substack{j=1 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j} = \left(\frac{x - x_1}{x_k - x_1}\right) \cdot \left(\frac{x - x_2}{x_k - x_2}\right) \cdot \left(\frac{x - x_3}{x_k - x_3}\right) \cdot \ldots \cdot \left(\frac{x - x_n}{x_k - x_n}\right)$$

where this last product skips the factor in which j = k, since that factor would have a denominator equal to zero. Despite its rather complicated appearance, this overall formula is quite easy to program, since the evaluation of each $L_k(x)$ just requires one **for**-type loop and the evaluation of y as the indicated sum requires another simple **for**-type loop.

However, despite its ease of implementation, this approach via an (n-1)th order polynomial to fit n points has some very serious problems from a computer graphics point of view. We describe just the most serious as a justification for abandoning this approach in favor of a variety of other methods.

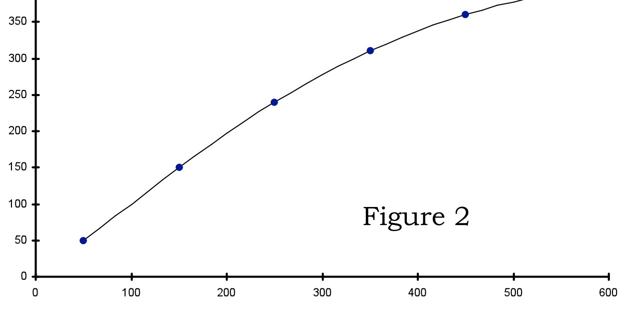

Figure 2

Example
Suppose you need to design the profile of the front of a sleek sports car. To keep it simple, we will use just six points to outline the shape, as shown in Figure 2 to the right. The slope of this profile is quite gradual, not changing drastically over short distances, and the curve drawn through the six points generated using formula (CURV0-4) above seems quite acceptable. Providing the units along the axes are scaled up correctly, this formula could now be used, for example, to program a machine that will actually produce this part of the car.

Now, suppose that you decide to snub the nose of the car a little bit, by moving the left-most of the "control" points in Figure 2 slightly rightwards. Figure 3 shows what happens for this point shifted to two new positions to the right of its original position. The resulting curves still go through the six specified points, but now the resulting profile has become quite unacceptable, with undulations of rather unsatisfactory size occurring between the points. This looks more like the profile of the sports car after an unfortunate collision with an immovable object.

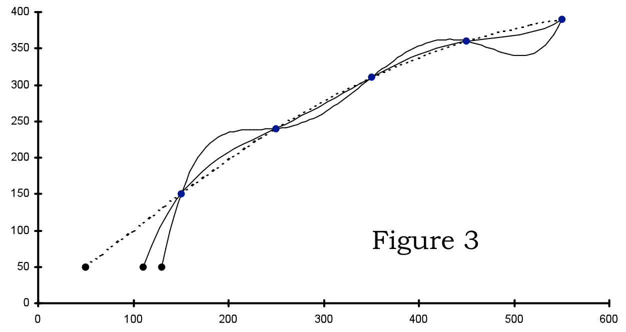

Figure 3

◆◆◆

This example illustrates a serious problem which is unavoidable with polynomial functions having more than a few terms (and, in fact, we can see it beginning to show in in Figure 1 which deals with just a three-term polynomial!). Polynomial functions turn out to be rather stiff, and as the order increases, become less and less able to follow a tightly twisted path that is "nailed" down at certain locations -- instead swinging wide between these control points (picture trying to drive a large transport truck through a slalom course at high spped – you'd probably have to make some rather wide swings to either side of the most direct path through the course!). Useful methods will require us to avoid high-order polynomials as much as possible, and look for other ways to build in "control" of the curve as a whole to eliminate this wavy behavior.

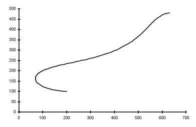
Figure 4

A second decisive problem with polynomial functions expressing y in terms of x as set out above is that such functions are inherently single-valued -- you substitute a specific value of x into the formula and you get one and only one value of y. This precludes expressing a curve which doubles back on itself (see Figure 4, for example, which might be a more extended profile of the front part of our mythical sports car) in terms of a single polynomial expression.

Three Strategies

Three major strategies are used separately or in combination to deal with these broad problem areas:

(i) **use of parametric equations**
Instead of using explicit **algebraic equations** to specify the points on a curve, that is, equations of the form

$$y = f(x) \qquad \text{in two dimensions}$$

or

$$z = f(x, y) \qquad \text{in three dimensions}$$[*]

we will work with **parametric equations**:

$$x = x(t)$$
$$y = y(t) \qquad \text{(CURV0-5)}$$
and
$$z = z(t),$$

expressing each of x, y, and z in terms of a fourth variable t (sometimes we may use other symbols, such as u, for the parameter, but the idea is the same.). This approach pays immediate dividends, because it reflects the way the whole subject is structured around the coordinate triples, (x, y, z), with no one of the three coordinates having any special status relative to the other two. The right-hand sides of equations (CURV0-5) can be regarded as coordinates of a point or as components of a vector, and we can, for example, apply all of our transformation tools to these formulas directly.

When we move from two-dimensions to three-dimensions, the algebraic equation approach involves completely different types of formulas. However, using a parametric equation approach, all we need to do is add a formula for z to our list of formulas for x and y. We don't have to change the way we regard x and y when it becomes necessary to introduce z. We take advantage of this by focussing primarily on two-dimensional examples only in this course, since you can easily extend the methods to handle three-dimensional curves (or **space curves** as opposed to **plane curves**) whenever necessary on your own.

There are a number of more subtle, mathematical advantages to the parametric formulation of the equations, as well. The presence of a new variable, t, does give added flexibility to the situation. Whereas the algebraic approach requires that each value of x give just one value of y (so a plane curve cannot double back on itself), or each pair of values of x and y give just one value of z (so a space curve or surface cannot double back on itself), the same restriction does not apply to (CURV0-5). What we do need is that x, y, and z are single-valued functions of t, but since t does not appear explicitly in the images we generate, this does not restrict the shapes of the curves or surfaces that we can generate. There are some lines in a plane which cannot be represent by an equation of the form y = f(x), but there are no such restrictions using a parametric form. If the equation is being used to generate coordinates of points along a path of movement in animation, the value of the parameter can be used to control the speed of movement along that path – something that is very difficult if not almost impossible to do when using an algebraic equation for the curve forming the animation path.

(ii) **use of control points rather than interpolation**
Notwithstanding the disaster in the sports car example above, we will not abandon the search for interpolation methods (that is, methods that generate curves that actually pass through a sequence of specified points). However, if "smoothness" is a significant goal, then it is often useful to relax the requirement that the curve pass through a sequence of points, and adopt an approach in which the shape of the curve is simply influenced by the locations of specified points. The points determining the shape of the curve are now known as **control points**, and the resulting curve will pass through few, if any, of them. Early use of this approach led to the **Bezier curves**, and more recently to so-called **B-spline curves** and their current state-of-the-art

[*] Actually, we've cheated here a bit. The single equation, z = f(x, y), gives a surface in three dimensions. To get a curve in three dimensions, you need a pair of equations

$$F_1(x, y, z) = 0 \qquad \text{and} \qquad F_2(x, y, z) = 0$$

which must be solved simultaneously to generate coordinates of points on the curve.

"nonuniform rational B-spline curves" or **NURBs**. We will spend some time on the B-spline formulas in this course, but time permits only the briefest of looks at NURBs.

(iii) **piecewise curves**
Finally, to avoid having to work with high order polynomials and so struggle with their inherent "stiffness", many methods – both interpolation methods and others – are based on piecing together a succession of low order polynomials. Third order polynomials are most popular for this task, since they are the simplest polynomials which can be non-planar (recall our earlier example in which we found that a second order or quadratic polynomial is the curve generated by exactly three points, and so must lie in a plane) and has a non-constant curvature (roughly speaking, the rate at which it twists varies from point to point).

The two major issues raised by this approach is (1) what sort of information is to be supplied to specify the shape of the individual segments making up the overall composite curve; and (2) how do we go about attaching the pieces together. There is considerable flexibility in how smooth (or continuous) the joints must be.

The Plan

Even superficial coverage of all of this material will add up to a fair amount of work. To keep these course documents to a manageable size, the rest of the material alluded to above is spread over three inter-related but separate documents. They are:

(i) CURVES_I_BEZIER.DOC, which describes an elegant idea apparently devised by de Casteljau in the 1950's, and the basis of Bezier's work in the late 1950's in developing the mathematical formulation of his now famous curves. We will give an algebraic as well as matrix formulation of Bezier curves in this document. Bezier curves in full generality and of arbitrarily high order are not commonly used in computer graphics applications. However, they do form the basis in concept and detail for more practical methods of curve generation.

(ii) CURVES_II_BSPLINE.DOC, which states the basic relation defining B-splines, and presents some specific formulas for the class of B-spline curves known as uniform non-rational B-splines. The generation of TrueType font glyphs is a familiar application of B-splines. The character outlines are a combination of straight line segments and quadratic or second degree B-splines.

(iii) CURVES_III_CUBICSP.DOC, which covers a variety of interpolation methods based on piecing together cubic or third-degree polynomial segments. This discussion starts with an algebraic development of so-called natural cubic splines, and then a presentation of the formulas for Hermite cubic splines. In addition to direct applications themselves, the Hermite formulas are a useful starting point for the description of a variety of other spline type formulas. We will look briefly at cardinal splines (which includes the Catmull-Rom spline), an interesting variation known as the Kochanek-Bartels spline (most useful in work involving animation paths), and finally an approach that pieces together cubic Bezier curves. This last method is implemented in the Win32 API PolyBezier() and PolyBezierTo() functions.

OK... So What is a "Spline" Anyway?

This word **spline** keeps cropping up in the discussion. Originally, the word "spline" referred to a thin piece of wood or other stiff material that artists and draftspersons used in drawing curves by hand. By bending this spline to fit between fixed locations on the paper, a smooth curve could be traced.

When we get into using mathematics to accomplish a similar goal, the use of the word is not as consistent by practitioners. In its most restricted sense, the term "spline" is applied to curves made up by stringing together low-order polynomials (usually third degree or cubic polynomials – the highest power of the parameter or independent variable is the third power) subject to certain standards of "smoothness" being maintained at the joints. (We are trying very hard here to avoid introducing terminology and concepts from differential calculus – the branch of mathematics which deals with issues of smoothness and continuity, among other things.) However, workers in computer graphics have tended at one time or another to use the term “spline” generally to mean a smooth continuous curve usually generated by piecing together simpler elements. Because of this, we will never use the word “spline” in these documents to imply anything more than a generated curve. If additional implications are not obvious from the context, we will state them explicitly.