Curves in Computer Graphics

Tom Davis
tomrdavis@earthlink.net
http://www.geometer.org/mathcircles
July 25, 2002

The only primitive geometric objects that computer graphics hardware typically can draw are points, lines and polygons. Usually there are special mechanisms to draw images, text characters, et cetera, but these are not commonly considered "geometric" since they are not usually transformed by geometric operations like rotation and perspective projection.

Obviously it is important that graphics programs be able to draw more complex geometric forms from simple circles to complex surfaces in three dimensions. In real computer hardware, curves are usually drawn as a series of short straight line segments, and surfaces as meshes of polygons, usually triangles or quadrilaterals.

A major problem in the design of graphics libraries is to provide a high level interface to the hardware that allows users to draw curves and surfaces from higher level descriptions rather than to require them to reduce curves and surfaces to line segments and polygons themselves.

In this paper we will discuss some methods that are used to draw curves; many surface generating techniques are similar, but needless to say, the situation there is often far more complicated.

1 Preliminaries

In the discussion that follows, we assume that the reader is at least aware of how points are transformed from their object coordinate system to screen coordinates by means of matrix multiplications. If not, a fairly complete elementary description of this can be found in:

`http://www.geometer.org/mathcircles/cghomogen.pdf`

In that paper can be found derivations for all of the transformation matrices used below, for example.

If you are unfamiliar with matrix multiplications of pairs of matrices or of matrices with vectors, look at:

`http://www.geometer.org/mathcircles/Matrices.pdf`

As a quick review, here are the key things to remember:

- Points are written as column vectors with an extra coordinate. If two-dimensional points are considered, the column vector will have three entries, usually called the x, the y, and the w coordinate. In three dimensions, the column vector will have four components : x, y, z, and w. In most cases, the final w coordinate is equal to 1.0 and the other coordinates are unchanged from their usual values in a cartesian coordinate system.

  If the w coordinate is not equal to 1.0, then the cartesian coordinates corresponding to the column vector $(x, y, z, w)^T$ is $(x/w, y/w, z/w)$. (The "T" that looks like an exponent above means the transpose of the row vector to make a column vector. This just saves space in the text.) In the examples below, vectors are written in their transposed forms.[1]

- Almost all of the common geometric operations can be performed on these column vectors by means of a premultiplication by a matrix. In two dimensions, it is a 3×3 matrix, and in three

[1] This system for identifying points is called "homogeneous coordinates", and is used in a branch of mathematics called projective geometry. If we allow the w coordinate to equal zero (as we do in projective geometry), we have a reasonable way to deal with "points at infinity" that are transformed to finite "vanishing points" by perspective projections.

dimensions, it will be a 4×4 matrix. These common transformations include translation, rotation, scaling, mirroring, shearing, and perspective projections, as well as any combination of these transformations.

Following are a few examples that illustrate typical operations and the corresponding matrices in three dimensions. To convert to the two-dimensional forms, simply erase the third row and column of each matrix (except, of course, in the case of the perspective projection where the eye is looking along the z-axis).

Rotation about the origin in the $x - y$ plane[2] by an angle θ:

$$\mathcal{R}_{z,\theta}\begin{pmatrix}x\\y\\z\\1\end{pmatrix} = \begin{pmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x\\y\\z\\1\end{pmatrix} = \begin{pmatrix}x\cos\theta - y\sin\theta\\ x\sin\theta + y\cos\theta\\ z\\ 1\end{pmatrix}.$$

Translation by t_x, t_y, and t_z in the x, y, and z directions, respectively:

$$\mathcal{T}_{t_x,t_y,t_z}\begin{pmatrix}x\\y\\z\\1\end{pmatrix} = \begin{pmatrix}1 & 0 & 0 & t_x\\ 0 & 1 & 0 & t_y\\ 0 & 0 & 1 & t_z\\ 0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x\\y\\z\\1\end{pmatrix} = \begin{pmatrix}x + t_x\\ y + t_y\\ z + t_z\\ 1\end{pmatrix}.$$

Finally, here is the matrix corresponding to a very simple perspective projection that apparently projects the points in a $90°$ box from the origin to the plane $z = 1$ as indicated in Figure 1. Notice that this is our only example where the w coordinate is affected:

$$\mathcal{P}\begin{pmatrix}x\\y\\z\\1\end{pmatrix} = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}\begin{pmatrix}x\\y\\z\\1\end{pmatrix} = \begin{pmatrix}x\\ y\\ z + 1\\ z\end{pmatrix}.$$

- All the above transformations can be combined simply by multiplying the corresponding matrices. Thus there is a single 4×4 matrix that represents the result of a rotation followed by a translation followed by a different rotation about a different axis, and finally by a perspective projection and it is equal to the product of the four matrices corresponding to the individual operations.

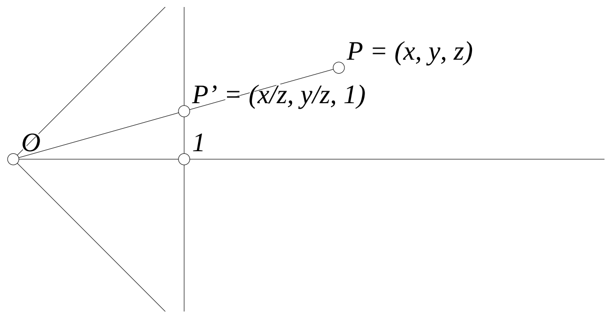

Figure 1: A Simple Perspective Projection

[2]The subscript z of $\mathcal{R}$ indicates that the rotation is about the z-axis, which is perpendicular to the $x - y$ plane and passes through the origin.

2 Parametric Curve Descriptions

From now on, to simplify things somewhat, we will consider only two-dimensional curves in the $x - y$ plane. In every case, these ideas can be extended to three dimensions in the obvious way, simply by describing what happens to the z coordinates in the same way as we do the x and y coordinates.

Most people are familiar with certain simple ways to describe special curves. For example, the equation that describes a circle centered at the point (c_x, c_y) and having radius r is this:

$$(x - c_x)^2 + (y - c_y)^2 = r^2.$$

This is very nice for some applications, but it is almost useless to a computer. Unless the computer somehow knows that this is the equation of a circle, how does it even begin to guess what values of x are appropriate to use to calculate the corresponding y values (of which there can be zero, one, or two, depending)? And even if it could, there is the nasty requirement that a square-root be taken for each evaluation—an operation that may slow things considerably.

The other problem is that although standard forms like the equation of a circle above exist for a few curves, quite often a graphics programmer may want something completely non-standard. What, for example, is the form of a curve that traces the backbone of a dinosaur in Jurassic Park VII?

If possible, the best way to represent curves on a computer is by means of parametric equations. In other words, it is nice to be able to describe the x and y coordinates of a curve (and the z coordinate if you're using curves that twist through three dimensions like the backbone of that dinosaur) in terms of a single parameter t that varies over some fixed range.

Sometimes this is trivial. For example, if we wish to describe the parabola $y = x^2$ from $x = -1$ to $x = 1$, we can do it as follows:

$$x(t) = t, \quad y(t) = t^2, \quad -1 \leq t \leq 1. \tag{1}$$

Notice that the equations for $x(t)$ and $y(t)$ satisfy the original equation: $y(t) = (x(t))^2$, and that as t varies from -1 to 1, then x does as well.

From the computer's point of view, this is a wonderful representation. If it is desired to approximate the curve as 20 short straight line segments, just connect the following points in order: $(x(-1.0), y(-1.0))$, $(x(-.9), y(-.9)), \ldots, (x(1.0), y(1.0))$. Notice that this will grind out a series of 21 points equally spaced along the x axis. The coordinates of each pair are easy to calculate; Equation 1 gives explicit formulas to calculate both $x(t)$ and $y(t)$.

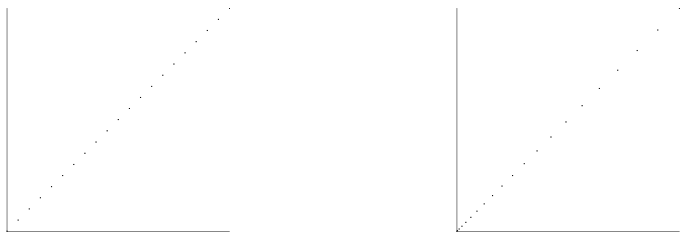

Figure 2: $x(t) = y(t) = t$ versus $x(t) = y(t) = t^2$

There are many ways to represent the same curve parametrically. For example, both of the following descriptions represent the same straight line segment $x = y$ from $x = 0$ to $x = 1$:

$$x(t) = t, \quad y(t) = t, \quad 0 \leq t \leq 1, \tag{2}$$

and

$$x(t) = t^2, \quad y(t) = t^2, \quad 0 \leq t \leq 1. \tag{3}$$

Both parameterizations are illustrated with equally-spaced values of t in Figure 2. The results of plotting Equation 2 are on the left, and those for Equation 3 are on the right.

In this case, Equation 2 is probably better in every way. The functions $x(t)$ and $y(t)$ are easier to calculate, and in addition, the points described in Equation 3 are not equally spaced—they are closer together near $t = 0$ than near $t = 1$, so if the curve is calculated with evenly-spaced values of t, the curve is more accurately approximated in some parts than in others.

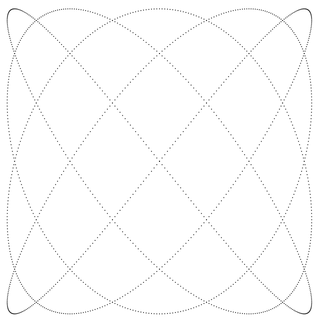

Figure 3: $x(t) = \sin 4t, y(t) = \sin 5t$

Although the function $\sin t$ is not particularly simple to calculate, some of the power of a parametric representation is illustrated in Figure 3, where $x(t) = \sin 4t$ and $y(t) = \sin 5t$, for $0 \leq t \leq 2\pi$. By changing the 4 and 5 to other integers (it works best if the integers you choose are relatively prime), very different curves can be obtained.

It is possible to describe a circle parametrically, but it is surprisingly messy. Here is one of the best parametric descriptions for the circle of radius 1 centered at the origin:

$$x(t) = \frac{t^2 - 1}{t^2 + 1}, \quad y(t) = \frac{2t}{t^2 + 1}, \quad -\infty < t < \infty. \tag{4}$$

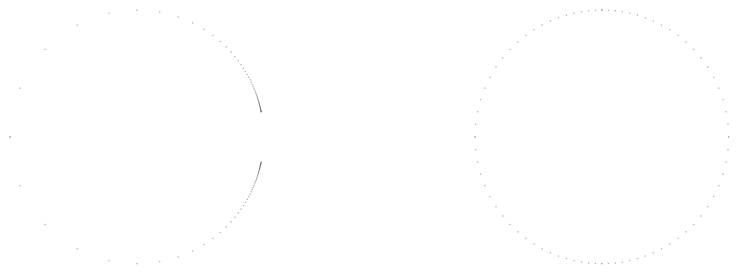

Figure 4: Two Circle Parameterizations

It is easy enough to check that the parameterization works; simply calculate $(x(t))^2 + (y(t))^2$ and verify that it is always equal to 1:

$$(x(t))^2 + (y(t))^2 = \frac{t^4 - 2t^2 + 1}{t^4 + 2t^2 + 1} + \frac{4t^2}{t^4 + 2t^2 + 1} = \frac{t^4 + 2t^2 + 1}{t^4 + 2t^2 + 1} = 1.$$

The equations themselves are not particularly nice; both involve a floating-point division (which is usually a relatively slow operation in computer hardware), and some method must be found to choose suitable values for t in the doubly-infinite range $-\infty < t < \infty$.

Even with these moderately ugly equations, there is an additional problem. Even if we allow t to vary over all real numbers, the parameterization above still omits one point on the circle, namely, the point $(1, 0)$ that is approached either as $t \to -\infty$ or as $t \to +\infty$.

The image on the left in Figure 4 shows the result of evaluating t in uniform steps of length 0.2 from $t = -10.0$ to $t = 10.0$. The steps are far too close on the right, and they do not even come close enough together to make a reasonable circle there. The steps on the left are probably too far apart for a good circle.

There is one trick we can perform that seems magical at first, but those versed in projective geometry will see that it is a natural and obvious result, given that the circle is a special form of a conic section. Since we know that the computer hardware is working not on two-dimensional vectors, but on three-dimensional vectors with an additional w-coordinate, and since the denominators of $x(t)$ and $y(t)$ are the same, why not describe the parameterization as follows:

$$x(t) = t^2 - 1, \quad y(t) = 2t, \quad w(t) = t^2 + 1, \quad -\infty < t < \infty. \tag{5}$$

Since the values of $x(t)$ and $y(t)$ are automatically divided by the value of $w(t)$ in the computer hardware this will work perfectly. There is still a division, of course, but that division had to be done anyway if the scene is viewed in perspective.

The problem of having an infinite domain for t can also be solved without too much trouble. For example, if we only allow t to vary from -1 to 1 in Equations 4 or 5, then only a half-circle is drawn. A similar equation can be written for the other half of the circle (just flip the sign in the definition of $x(t)$) and the circle can be drawn in two halves. Notice that even if t is restricted to this smaller range, the points generated are not evenly spaced on the circle.[3] The results of this double parameterization are displayed in the illustration on the right in Figure 4. Both the left and right sides of the circle are evaluated by stepping t by 0.05 from $t = -1.0$ to $t = 1.0$.

Notice that circles (and other curves generated in a similar way with a variable w coordinate) are transformed correctly by any of the standard transformation matrices. They can be rotated, translated, scaled, mirrored, or put through perspective projections. Since any conic section (ellipse, parabola, hyperbola, or circle) is just a projection of a circle, it is clear that any of those curves can be generated by some parametric equation with a varying value of w in the same way as are circles. In fact, if you know the values of the entries in the appropriate projection matrix you can just multiply the column vector $(t^2 - 1, 2t, t^2 + 1)^T$ by that matrix to obtain the parameterization of any conic section.

3 A Hardware Hack

Graphics hardware is generally highly parallel, and there are usually plenty of available adders, multipliers, et cetera, available to perform somewhat non-standard operations, assuming some sucker can be talked into writing a bit of microcode.

Since the hardware is usually built to work in three dimensions using 4×4 matrices, there is a fairly simple hardware hack that can be done that will generate points not only on segments of conic sections, but on any curve where the parameterizations of $x(t)$, $y(t)$, $z(t)$, and $w(t)$ are all cubic polynomials of the form $at^3 + bt^2 + ct + d$, where a, b, c, and d are constant real numbers. Obviously we can let $a = 0$ if we want quadratic equations, or $a = b = 0$ for linear equations, et cetera.

Parametric curves whose components are polynomials are usually called “splines” or “spline curves”, especially if the value of the w component is the constant 1.0. If w varies, they are usually called “rational splines”. Commonly used splines include the B-spline, the Bézier spline, the cubic spline, the Hermite spline, et cetera.

Let us begin with a few observations. In the most general case, we would like to generate automatically a set of equations of the following form:

$$\begin{aligned} x(t) &= x_3t^3 + x_2t^2 + x_1t + x_0 \\ y(t) &= y_3t^3 + y_2t^2 + y_1t + y_0 \\ z(t) &= z_3t^3 + z_2t^2 + z_1t + z_0 \\ w(t) &= w_3t^3 + w_2t^2 + w_1t + w_0, \end{aligned}$$

[3]Usually circles drawn this way are divided into four quarters which gives better overall accuracy with fewer values of t evaluated.

where $x_0, \ldots, w_3$ are all constant real numbers. A bit of thought shows that this is equivalent to:

$$\begin{pmatrix} x(t) \\ y(t) \\ z(t) \\ w(t) \end{pmatrix} = \begin{pmatrix} x_3 & x_2 & x_1 & x_0 \\ y_3 & y_2 & y_1 & y_0 \\ z_3 & z_2 & z_1 & z_0 \\ w_3 & w_2 & w_1 & w_0 \end{pmatrix} \begin{pmatrix} t^3 \\ t^2 \\ t \\ 1 \end{pmatrix}. \tag{6}$$

Thus, if we have a nice way to generate the values of t^3, t^2, and t, we can just plug them into the vector on the right of Equation 6 and premultiply by the matrix on the left of the t-vector. Note that it may be important to transform these points with rotations, perspective projections, et cetera, but those operations simply involve premultiplications by other 4×4 matrices. All these premultiplications can be combined so successive points can be generated by successively plugging in new values of the powers of t into the column vector and then multiplying it by the matrix of polynomial coefficients for the cubic functions to obtain points that trace out the curve in question.

This is pretty good, but we can do better!

We will simplify things here by assuming that we want t to vary from 0 to 1, but the values below can be adjusted suitably to work for any finite range of values of t. We will generate successive values of the vector $(t^3, t^2, t, 1)^T$ using only a series of additions.

Let d be the difference we want between successive values of t. For example, if we want to evaluate t for 10 equal steps between 0 to 1, we would set $d = 1/10$. Consider the following matrix:

$$\begin{pmatrix} 6d^3 & 6d^3 & d^3 & 0 \\ 0 & 2d^2 & d^2 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The right column is $(0, 0, 0, 1)^T$, the value of the desired vector $(t^3, t^2, t, 1)^T$ when $t = 0$. Now consider the following operation on the matrix: Add the third column to the fourth, then the second column to the third, and finally, the first column to the second. Leave the first column unchanged, and we obtain:

$$\begin{pmatrix} 6d^3 & 12d^3 & 7d^3 & d^3 \\ 0 & 2d^2 & 3d^2 & d^2 \\ 0 & 0 & d & d \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The right column represents the value of our vector $(t^3, t^2, t, 1)^T$ when $t = d$. Repeat the same addition operation to obtain:

$$\begin{pmatrix} 6d^3 & 18d^3 & 19d^3 & 8d^3 \\ 0 & 2d^2 & 5d^2 & 4d^2 \\ 0 & 0 & d & 2d \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The right column corresponds to the value of that same vector when $t = 2d$. Verify yourself that if the operation is repeated two more times that the values for $t = 3d$ and $t = 4d$ are obtained. This always works. Why?

So now, instead of having to do multiplications to obtain values of t^3, t^2 and t, we simply have to do a set of parallel additions. This is not a particularly great savings, since floating-point additions are as ugly as multiplications, but we can go one step further—we don't even have to do the matrix multiplication in Equation 6! All we need to do is perform the multiplication:

$$\begin{pmatrix} x_3 & x_2 & x_1 & x_0 \\ y_3 & y_2 & y_1 & y_0 \\ z_3 & z_2 & z_1 & z_0 \\ w_3 & w_2 & w_1 & w_0 \end{pmatrix} \begin{pmatrix} 6d^3 & 6d^3 & d^3 & 0 \\ 0 & 2d^2 & d^2 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7}$$

and iterate that by adding the columns as above, and this will generate all the points on the curve. It is not hard to prove that this is so, but let us just work out a few points on the circle described in Equation 5 beginning at $t = 0$ and with $d = 0.1$. We will just assume that the z-coordinate is the constant zero.

Plugging the values of 0, .1, .2, and .3 for t into Equation 5, we obtain:

$$\begin{aligned} t = 0 &\quad:\quad (-1, 0, 0, 1)^T \\ t = .1 &\quad:\quad (-.99, .2, 0, 1.01)^T \\ t = .2 &\quad:\quad (-.96, .4, 0, 1.04)^T \\ t = .3 &\quad:\quad (-.91, .6, 0, 1.09)^T \end{aligned}$$

Now let's see what we get by doing the matrix multiplication in Expression 7, where:

$$\begin{pmatrix} x_3 & x_2 & x_1 & x_0 \\ y_3 & y_2 & y_1 & y_0 \\ z_3 & z_2 & z_1 & z_0 \\ w_3 & w_2 & w_1 & w_0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & -1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

and

$$\begin{pmatrix} 6d^3 & 6d^3 & d^3 & 0 \\ 0 & 2d^2 & d^2 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} .006 & .006 & .001 & 0 \\ 0 & .02 & .01 & 0 \\ 0 & 0 & .1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

We obtain:

$$\begin{pmatrix} 0 & 1 & 0 & -1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} .006 & .006 & .001 & 0 \\ 0 & .02 & .01 & 0 \\ 0 & 0 & .1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & .02 & .01 & -1 \\ 0 & 0 & .2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & .02 & .01 & 1 \end{pmatrix} \tag{8}$$

.

We seem to be on the right track; the right column of the matrix on the right of Equation 8 is the value corresponding to $t = 0$. If we iterate that matrix three more times, we obtain:

$$\begin{pmatrix} 0 & .02 & .03 & -.99 \\ 0 & 0 & .2 & .2 \\ 0 & 0 & 0 & 0 \\ 0 & .02 & .03 & 1.01 \end{pmatrix}, \begin{pmatrix} 0 & .02 & .05 & -.96 \\ 0 & 0 & .2 & .4 \\ 0 & 0 & 0 & 0 \\ 0 & .02 & .05 & 1.04 \end{pmatrix}, \begin{pmatrix} 0 & .02 & .07 & -.91 \\ 0 & 0 & .2 & .6 \\ 0 & 0 & 0 & 0 \\ 0 & .02 & .07 & 1.09 \end{pmatrix}.$$

The values in the right columns are exactly the ones we need!

But there is still more. We can also premultiply by all the transformation matrices that might rotate, scale, or put into perspective the points on the curve, and then iterate that. So for the cost of one premultiplication, we can generate an arbitrary number of points on the curve with 12 additions each, and those additions can be done four at a time with appropriately designed parallel hardware.

There are moderately simple ways to modify this method to generate points on any sort of rational cubic spline curve, which covers perhaps 99.99% of all curves ever drawn on a computer screen.

4 Splines

One of the easiest ways to draw and control curve shapes is using splines[4]. There are various sorts of splines, but the shape of each is controlled using a set of control points. The curve does not necessarily pass through the control points, although it is required to for some splines, but it is drawn toward those points.

[4]The name "spline" comes from a flexible piece of wood called a spline used by draftsmen in the old days to draw curves.

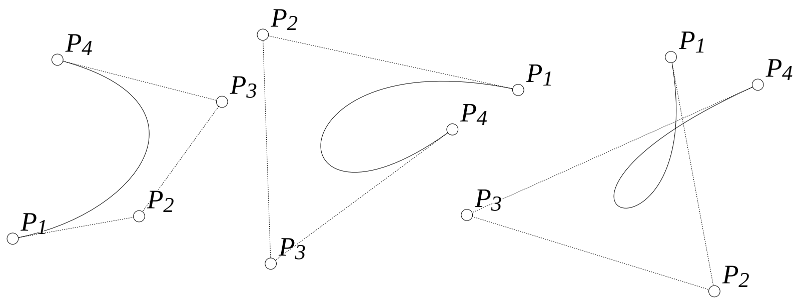

Figure 5: A few Bézier Splines

There are many types of splines, but here we will consider only cubic Bézier splines (sometimes called cubic Bézier curves). A few examples of these are illustrated in Figure 5. In that figure, the control points are labeled P_1, P_2, P_3, and P_4. The splines are the curves beginning at P_1 and ending at P_4.

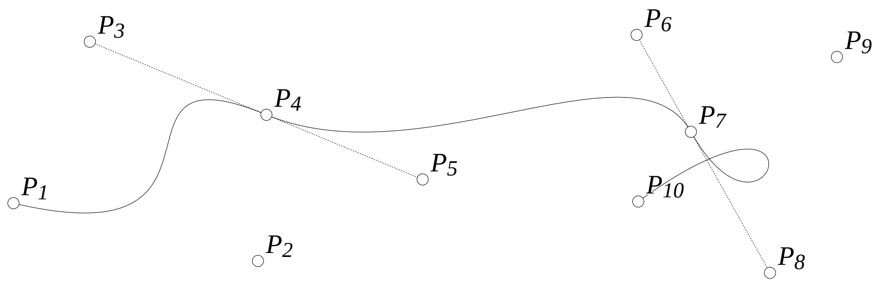

Figure 6: Linking Bézier Splines

The dotted straight lines connecting the control points are drawn simply to illustrate a property of Bézier curves, namely, that the initial and final parts of the curve are tangent to the lines P_1P_2 and P_4P_3. This tangency requirement makes it easy to construct more complex curves that fit together nicely using multiple segments as shown in Figure 6. Notice that as long as points P_3, P_4 and P_5 lie on the same line and P_6, P_7 and P_8 also lie on a line, the curves will come together with matching tangents and will appear to make a smooth transition[5].

It is quite simple to write down the parametric curve for the cubic Bézier spline defined by the four control points P_1, P_2, P_3 and P_4. If we consider the control points to be vectors (in either two or three dimensions), the parametric definition of the Bézier spline associated with them is given by:

$$\mathcal{B}_{P_1,P_2,P_3,P_4}(t) = (1-t)^3P_1 + 3(1-t)^2tP_2 + 3(1-t)t^2P_3 + t^3P_4. \tag{9}$$

The functions $(1-t)^3$, $3(1-t)^2t$, $3(1-t)t^2$ and t^3 are called the “Bézier basis” for the spline curves. Notice that they are the results of expanding the expression $((1-t)+t)^3$. The Bézier basis for $n+1$ control points is obtained by replacing the 3 in the exponent by n. Every other polynomial spline can be described using exactly the same form as in Equation 9 but with the basis functions replaced by other polynomials in t. It is easy to see that when $t=0$ the value will be P_1, and when $t=1$, the value will be P_4. Intermediate values of t generate points along some curve connecting P_1 with P_4.

In Section 3 we described a method to generate for a circle a matrix that can be iterated in a somewhat strange way using 12 additions such that its rightmost column will step through points on the curve with

[5]Actually, to get true smoothness, the distance P_3P_4 must be the same as P_4P_5 and similarly P_6P_7 must be equal to P_7P_8. This is because the tangent vectors at the endpoints have magnitudes proportional to the vectors connecting the control points.

each iteration. There is a wonderful way to do exactly this for *every* curve that can be described as a cubic spline or rational cubic spline.

Without proof, here is the construction.

Let $P_i = (p_{i1}, p_{i2}, p_{i3}, p_{i4})^T$ for $i = 1, 2, 3, 4$. Let t vary from $t = 0$ to $t = 1$ in steps of size d. In addition, there is a 4×4 basis matrix that depends only on the spline type; for Bézier splines that basis matrix will be:

$$\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Perform the following matrix multiplication:

$$\begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \\ p_{41} & p_{42} & p_{43} & p_{44} \end{pmatrix} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 6d^3 & 6d^3 & d^3 & 0 \\ 0 & 2d^2 & d^2 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{10}$$

where the matrix involving d is the same as the one used in Expression 7 and it will generate a matrix that can be iterated to grind out successive points on the Bézier curve defined by the control points P_1, P_2, P_3 and P_4 with spacing d. To see why, look at the top row of the product of the first two matrices:

$$(-p_{11} + 3p_{12} - 3p_{13} + p_4, 3p_{11} - 6p_{12} + 3p_{13}, -3p_{11} + 3p_{12}, p_{11}). \tag{11}$$

Recalling that the matrix on the right effectively generates under iteration points of the form $(t^3, t^2, t, 1)^T$, when multiplied by the top row shown in Expression 11 of the product of the two matrices on the left of Expression 10 yields:

$$(-p_{11} + 3p_{12} - 3p_{13} + p_{14})t^3 + (3p_{11} - 6p_{12} + 3p_{13})t^2 + (-3p_{11} + 3p_{12})tp_{11}. \tag{12}$$

A little algebra shows that Expression 12 is equivalent to:

$$\begin{aligned} & p_{11}(1 - 3t + 3t^2 - t^3) + p_{12}(3 - 6t + 3t^2) + p_{13}(-3t^3 + 3t^2) + p_{14}t^3 \\ &= (1-t)^3 p_{11} + 3(1-t)^2 t p_{12} + 3(1-t)t^2 p_{13} + t^3 p_{14}, \end{aligned}$$

just as it should. The other rows multiply out in exactly the same way and hence the grand product of the three matrices in Expression 10 can be iterated to generate points on the Bézier curve defined by the control points P_1, P_2, P_3 and P_4.