Matrices and Conversions for Uniform Parametric Curves

Ian Millington
idmillington@gmail.com
The R’u’D Guy - http://therndguy.com

May 12, 2004
this revision: November 17, 2009
# Contents

1 Introduction .................................................................................................................. 3  
   1.1 Nomenclature ........................................................................................................... 3  

2 Three Basic Curves .......................................................................................................... 4  
   2.1 General Form ............................................................................................................ 4  
   2.2 Bézier Curves .......................................................................................................... 5  
   2.3 Catmull-Rom Curves ................................................................................................. 6  
   2.4 Hermite Curves ........................................................................................................ 6  

3 Understanding Cubic Curves ............................................................................................. 8  
   3.1 Curves in Two and Three Dimensions ...................................................................... 9  
   3.2 Transformed Curves ................................................................................................. 10  

4 Control Point Conversions ............................................................................................... 11  
   4.1 Catmull-Rom to Bézier ............................................................................................. 12  
   4.2 Bézier to Catmull-Rom ............................................................................................. 12  
   4.3 Hermite to Bézier .................................................................................................... 12  
   4.4 Bézier to Hermite .................................................................................................... 12  
   4.5 Hermite to Catmull-Rom .......................................................................................... 12  
   4.6 Catmull-Rom to Hermite .......................................................................................... 13  

5 B-Splines .......................................................................................................................... 14  
   5.1 B-Spline to Bézier .................................................................................................... 14  
   5.2 Bézier to B-Spline .................................................................................................... 15  

6 Curves with Additional Parameters .................................................................................... 16  
   6.1 Cardinal Curves ........................................................................................................ 16  
      6.1.1 Cardinal to Hermite ......................................................................................... 17  
      6.1.2 Characteristic Matrix ....................................................................................... 17  
      6.1.3 Cardinal to Bézier ............................................................................................ 18  
      6.1.4 Bézier to Cardinal ............................................................................................. 18  
   6.2 Kochanek-Bartels Curve (TCB Curves) .................................................................... 18  
      6.2.1 Kochanek-Bartels to Hermite ......................................................................... 21  
      6.2.2 Characteristic Matrix ....................................................................................... 21  
      6.2.3 Kochanek-Bartels to Bézier .......................................................................... 21  
      6.2.4 Bézier to Kochanek-Bartels .......................................................................... 22  

7 Lower Order Bézier Curves ............................................................................................... 23  
   7.1 Conversion from Quadratic to Cubic Bézier ............................................................ 24  
   7.2 Conversion from Cubic to Quadratic Bézier ............................................................. 24  
   7.3 Linear Elements ....................................................................................................... 24
8 Subdivision
  8.1 Changing Parameters ........................................ 26
  8.2 Generating Subdivided Curve Coefficients .................. 28
  8.3 Generating Subdivided Control Points ..................... 28

9 Implementing Curves ........................................ 30
  9.1 Evaluating Curves ........................................... 31
  9.2 Forward Differencing ....................................... 32
    9.2.1 Linear Forward Differences ............................. 33
    9.2.2 Quadratic Forward Differences ......................... 34
    9.2.3 Cubic Forward Differences ............................... 35
  9.3 Retrieving Control Points .................................. 36
    9.3.1 Recalculating Control Points from Coefficients ....... 36
  9.4 Implementing Subdivision and Curve Trimming .............. 37
  9.5 Implementing Vector Curves ................................ 38
    9.5.1 Sequential Calculation .................................. 38
    9.5.2 Polymorphic or Templated Implementation ............... 39
    9.5.3 Vectorization and GPU Implementation ................... 40

10 Epilogue .................................................... 42
Chapter 1

Introduction

Several times when I’ve been working with parametric curves I’ve been left scrabbling around the web and my bookshelf for the matrices associated with different types of parametric curve. In the end they aren’t that hard to calculate from scratch, but it is useful to just have the values ready to go. The need for a quick-reference is the origin of this paper. As I found a new bit of data I might need again, I added it to the paper.

Around the basic data I’ve gradually added explanation so that the contents are more useful to colleagues and friends who have had a copy. Eventually it has grown into a broad ranging discussion of uniform parametric curves, focusing on their matrix formulations (which are essential for really understanding the curves and being able to implement them efficiently).

1.1 Nomenclature

I have tried to be consistent in this paper distinguishing between curves and splines. A curve, in this sense, is a single parametric equation. A spline is a series of curves stitched together to form a longer and more complex figure.

This nomenclature is relatively clear, I hope, but it isn’t easy to be totally consistent. Some curve shapes are almost never seen except in a spline form, and so it didn’t make sense to rename them. In particular I’ve never heard a single unit of a B-spline called a B-curve, and so naming them seems ugly, so I have kept the conventional naming.
Chapter 2

Three Basic Curves

In this section I’ll look at the three simplest common curve types, the Bézier curve, the Catmull-Rom curve and the Hermite curve.

I will primarily consider their cubic forms. The Bézier curve in particular is commonly used in its quadratic form as well, but discussion of that is left until chapter 7. Explanation of what a curve is, and what makes a particular curve cubic is found in chapter 3. I have placed it after the introduction to the basic curves below, so I’ll have an example to work with. Skip there now if you want some more background.

2.1 General Form

Each cubic parametric curve can be written in matrix form as:

\[ x(t) = \vec{t} M_α \vec{x} \]  \hspace{1cm} (2.1)

where \( M_α \) is the a matrix that is characteristic of the curve type but independent of the particular curve configuration (so all Bézier curves have one matrix, all Catmull-Rom curves another), \( \vec{x} \) is a vector of control values that determine the shape of a particular curve (so two Bézier curves have the same characteristic matrix, but different control values)\(^1\), and \( \vec{t} \) is a row vector of the parameter \( t \) to various powers:

\[ \vec{t} = [ t^3 \quad t^2 \quad t \quad 1 ] \]  \hspace{1cm} (2.2)

I will use matrix form throughout this paper because that is the most convenient for implementation, given that most modern hardware has dedicated vector math processors. It also helps by showing some of the underlying form and similarities in the curves we’ll be discussing.

Because we’re dealing with cubic curves, the \( \vec{t} \) vector has four elements, as above; the characteristic matrix has four rows and columns, and there will be four control values. In general for a curve of order \( n \), there will be \( n + 1 \) rows and columns in the matrix and \( n + 1 \) control values, and the \( \vec{t} \) vector will be:

\[ \vec{t}_n = [ t^n \quad t^{n-1} \quad \ldots \quad t \quad 1 ] \]  \hspace{1cm} (2.3)

\(^1\)Some texts call this the ‘Geometry vector’ for a curve.
2.2 Bézier Curves

The Bézier curve (figure 2.1) is defined as:

\[ x_B(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i x_i \]  

(2.4)

where \( x_0 \ldots x_3 \) are the four control values, and

\[ \binom{n}{i} = \frac{n!}{i!(m-i)!} \]  

(2.5)

is the binomial coefficient. The Bézier curve is also sometimes written as:

\[ x_B(t) = \sum_{i=0}^{n} B^n_i(t) x_i \]  

(2.6)

which is just a factorization of equation 2.4 where

\[ B^n_i(t) = \binom{n}{i} (1-t)^{n-i} t^i \]  

(2.7)

\( B^n_i(t) \) is called the Bernstein polynomial.

![Figure 2.1: A cubic Bézier curve is defined by its four control values.](image)

We are concerned with the cubic case, which expands to:

\[ x_B(t) = (1-t)^3 x_0 + 3(1-t)^2 t x_1 + 3(1-t)t^2 x_2 + t^3 x_3 \]  

(2.8)

or, in matrix form:

\[ x_B(t) = \vec{t} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 \\ -3 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \]  

(2.9)

where the characteristic matrix is given by:

\[ M_B \overset{\text{def}}{=} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 \\ -3 & 3 & 1 \end{bmatrix} \]  

(2.10)

This is the characteristic matrix for all cubic Bézier curves. Higher or lower order Béziers have similar form. The second degree (quadratic), for example, has the matrix:

\[ M_B^2 \overset{\text{def}}{=} \begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 1 \end{bmatrix} \]  

(2.11)
Note that these matrices display a convention I’ve followed throughout this paper: when a matrix is triangular\(^2\), I have omitted the 0 values outside the triangle. This makes the structure of the matrix clearer at first sight, at least to me. For all other matrices I include the 0s.

### 2.3 Catmull-Rom Curves

Catmull-Rom curves (figure 2.2) move between their central two control points, with tangents controlled by the first and last control point. Catmull-Rom curves are normally arranged in a sequence as a spline, where the start point of one curve is the first control point of the next. Because of this we normally indicate their control points as \( x_{-1}, x_0, x_1, \) and \( x_2 \) to suggest that the \( x_i \) values form some longer series.

![Figure 2.2: A Catmull-Rom curve.](image)

In matrix form, the Catmull-Rom curve is:

\[
\mathbf{x}_{CR}(t) = t \mathbf{M}_{CR} \begin{bmatrix} x_{-1} \\ x_0 \\ x_1 \\ x_2 \end{bmatrix}
\]  

where the characteristic matrix \( \mathbf{M}_{CR} \) is given by:

\[
\mathbf{M}_{CR} \overset{\text{def}}{=} \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}
\]  

### 2.4 Hermite Curves

Hermite curves use two points and two derivatives to control the shape of the curve (as opposed to four points used in the other curve types).

The control values are given by the vector:

\[
\begin{bmatrix} x_0 \\ x_1 \\ \dot{x}_0 \\ \dot{x}_1 \end{bmatrix}
\]  

where \( x_0 \) and \( x_1 \) are the start and end of the curve, and \( \dot{x}_0 \) and \( \dot{x}_1 \) are the first derivatives at those points with respect to the parameter, \( t \). Because curves are normally used in two or three

---

\(^{2}\)Strictly a matrix is said to be triangular if its elements either above or below the leading diagonal are zero. I adopt the same convention for matrices that have zeros above or below the trailing diagonal also, as in the Bézier characteristic matrix.
dimensions, these derivatives are more commonly called ‘tangents’ in texts, although the magnitude of the derivative is important\(^3\).

The Hermite curve is given by:

\[
x_H(t) = t M_H \begin{bmatrix} x_0 \\ x_1 \\ \dot{x}_0 \\ \dot{x}_1 \end{bmatrix}
\]  

(2.15)

where the characteristic matrix \(M_H\) is given by:

\[
M_H \overset{\text{def}}{=} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}
\]  

(2.16)

A Hermite curve is shown in figure 2.3. One feature of the Hermite curve that is often confusing is the direction of the tangents. If you are used to working with Bézier curves, it may look like the last control value, \(\dot{x}_1\) is in the wrong direction.

---

\(^3\) The word ‘tangent’ is ambiguous - sometimes it refers to the derivative, sometimes to the direction only. For that reason, and because the math in this paper works in any dimension, including one dimension, I’ll try to avoid calling them tangents.
Chapter 3

Understanding Cubic Curves

At this point I want to depart from the raw information to look at what we’re doing when we use the matrix form of a curve. Now we have the characteristic matrices of our three basic curves, we can use them as an example to inform our intuition. If you are just interested in the meat of this paper, then feel free to skip to the next chapter.

A parametric cubic curve is a function of one variable (the parameter). It can be written:

\[ x(t) = at^3 + bt^2 + ct + d \] (3.1)

where the coefficients \(a, b, c\) and \(d\) are constant for one particular curve, but will differ from curve to curve\(^1\). This equation is defined for all \(t \in \mathbb{R}\), but typically we use only values in some interval, and most commonly we use the closed interval \([0, 1]\). Unless explicitly stated otherwise I will use this interval\(^2\).

Equation 3.1 is the simplest and most general form of the cubic parametric curve. But unfortunately it isn't very user-friendly. If we want a curve of a particular shape, it is difficult to know what coefficient values will give the desired result.

This is where the curve types in this paper excel. Rather than using the four coefficients \(a, b, c\) and \(d\), they use four control values that make more intuitive sense to the human creating the curve. These four control values are then transformed into the coefficients for equation 3.1. In the Bézier case, for example, the control values represent the position of the curve at \(t = 0\) (which I’ll call the ‘start’ of the curve) and at \(t = 1\) (which I’ll call the ‘end’ of the curve) and two additional values that shape the curve in between these parameters. This pair of intermediate control values are such that, when moved interactively by a human, the curve responds in an intuitive manner. This makes it easy for a human operator to get the curve shape they are looking for quickly.

The transformation from control values to coefficients is exactly what the characteristic matrix does.

We can rewrite equation 3.1 as

\[
\begin{bmatrix}
\begin{bmatrix}
  t^3 & t^2 & t & 1 \\
\end{bmatrix}
\end{bmatrix}
\begin{bmatrix}
  a \\
  b \\
  c \\
  d \\
\end{bmatrix}
\] (3.2)

\(^1\)There are types of curve, called “rational curves” that do not correspond to equation 3.1, the most famous being NURBS. They are not strictly cubic parametric curves at all, although they are constructed from them.

\(^2\)Non-uniform curves don’t typically use this interval, and many curves that are used as poly-curve splines (such as the Catmull-Rom curve) use intervals of unit size but at successive starting points: \([0, 1], [1, 2], \ldots, [n, n+1]\) for successive curves in the spline.)
\[
\begin{bmatrix}
a \\
b \\
c \\
d
\end{bmatrix}
= 
\begin{bmatrix}
-1 & 3 & -3 & 1 \\
3 & -6 & 3 & 0 \\
-3 & 3 & 1 & 0 \\
1 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3
\end{bmatrix}
\]  
(3.3)

for a Bézier curve. And substituting equation 3.3 into equation 3.2 we get the matrix form of the Bézier curve from equation 2.9.

If the characteristic matrix is a transformation from control values to coefficients of the cubic, then the inverse of that matrix would transform the coefficients back into control values. We’ll see that in practice in section 9.3.

Understanding the characteristic matrices of parametric curves as transformations in this way will help us throughout this paper, but particularly in chapter 9 when we look at implementing curves in software.

One final note on notation here, there will be four coefficients \((a, b, c\), and \(d)\) for the cubic curve, but more or fewer for other orders of curve. In general I will write \(\vec{a}\) to indicate the vector of curve coefficients, regardless of the order of the curve. We could then say:

\[
x(t) = \vec{t}\vec{a}
\]  
(3.4)

and denote the coefficient transform as:

\[
\vec{a} = M_{\vec{a}}\vec{x}
\]  
(3.5)

### 3.1 Curves in Two and Three Dimensions

In the previous explanation I have assumed that the cubic curve has a single real value \(x(t) \in \mathbb{R}\) at each parameter\(^3\). I have therefore also assumed that the coefficients \(a, b, c\), and \(d\) in equation 3.1 are also real values \(a, b, c, d \in \mathbb{R}\) and similarly for the control values for each curve type.

This is somewhat limiting. While there are cases when we want to use a parametric curve to represent a single varying quantity, there are many more applications where the curve needs to represent a two or three dimensional figure. In other words we might want \(x(t) \in \mathbb{R}^2, x(t) \in \mathbb{R}^3\) or \(x(t) \in \mathbb{R}^4\) for homogenous coordinates in three dimensions\(^4\).

Moving from a single value to two and three dimensional vectors does not require a change in the mathematics presented in this paper, but does require a change in the type of control values we use. To get \(x(t) \in \mathbb{R}^2\), for example, we need \(a, b, c, d \in \mathbb{R}^2\) and therefore each element in our control value vector will also be \(X_i \in \mathbb{R}^2\). But the characteristic matrices (and the conversion matrices we’ll meet later) remain entirely unchanged.

We could write the equations to generate vector outputs, for example:

\[
\begin{bmatrix}
x(t) \\
y(t) \\
z(t)
\end{bmatrix}
= \begin{bmatrix}
t^3 & t^2 & t & 1
\end{bmatrix}
\begin{bmatrix}
-1 & 3 & -3 & 1 \\
3 & -6 & 3 & 0 \\
-3 & 3 & 1 & 0 \\
1 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x_0 \\
y_0 \\
z_0 \\
x_1 \\
y_1 \\
z_1 \\
x_2 \\
y_2 \\
z_2 \\
x_3 \\
y_3 \\
z_3
\end{bmatrix}
\]  
(3.6)

\(^3\)The parameter itself is real valued, \(t \in \mathbb{R}\), in all cases. I have never seen an application that required otherwise, and the mathematics in this paper would not apply.

\(^4\)There are applications where it is useful to have \(x(t) \in \mathbb{C}^n\) as well. In those cases elements of the mathematics I present in this paper does change (because matrix inverses involve conjugates). I am primarily interested in these curves as used in computer graphics, so I will omit the discussion of complex valued curves entirely.
but this is entirely equivalent to three separate curves, one for each dimension:

\[
x(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix}
-1 & 3 & -3 & 1 \\
3 & -6 & 3 & 1 \\
-3 & 3 & -3 & 1 \\
1 & & & 
\end{bmatrix} \begin{bmatrix} x_0 \\
x_1 \\
x_2 \\
x_3 
\end{bmatrix}
\]

\[
y(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix}
-1 & 3 & -3 & 1 \\
3 & -6 & 3 & 1 \\
-3 & 3 & -3 & 1 \\
1 & & & 
\end{bmatrix} \begin{bmatrix} y_0 \\
y_1 \\
y_2 \\
y_3 
\end{bmatrix}
\]

\[
z(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix}
-1 & 3 & -3 & 1 \\
3 & -6 & 3 & 1 \\
-3 & 3 & -3 & 1 \\
1 & & & 
\end{bmatrix} \begin{bmatrix} z_0 \\
z_1 \\
z_2 \\
z_3 
\end{bmatrix}
\]

For most of the rest of this paper I will continue to discuss curves without specifying what type of value they are generating. Some of the curves in this paper are almost always used with vectors in practical applications. To reflect this I will sometimes call the control values ‘points’ when the discussion seems clearer as a result.

### 3.2 Transformed Curves

One issue that is more pressing with two and three dimensional curves is transformations. We typically want to be able to take a curve and transform it via a combination of translation, rotation and scaling, and possibly skewing. We would like to be able to calculate the curve resulting from such a transformation.

Assuming that our transformation is in matrix form:

\[
\vec{v}' = \vec{v}T = \begin{bmatrix} x & y & z & 1 \end{bmatrix} T
\]

where \(\vec{v}\) is a position row-vector using homogenous coordinates (so we can support translation), and the transform matrix \(T\) is structured for pre-multiplication\(^5\). We can transform the curve:

\[
\begin{bmatrix} x'(t) & y'(t) & z'(t) & 1 \end{bmatrix} = \begin{bmatrix} x(t) & y(t) & z(t) & 1 \end{bmatrix} T = \vec{t}M_\alpha \begin{bmatrix} x_0 & y_0 & z_0 & 1 \\
x_1 & y_1 & z_1 & 1 \\
x_2 & y_2 & z_2 & 1 \\
x_3 & y_3 & z_3 & 1 
\end{bmatrix} T
\]

Multiplying the control value matrix by the transform matrix gives us a new set of control values for the translated curve. In the same way, if we are using coefficients of the curve rather than control values (i.e. we’ve already multiplied out by \(M_\alpha\)), we can multiply the coefficients by the transform matrix to get a new set of coefficients.

This means that to transform our curve, we just transform its control values or its coefficients. The calculation does not depend on the type of curve used.

---

\(^5\)Many graphics systems use column-vectors and post-multiplication instead, but because our control values are placed in a column-vector, row-vectors are simpler to combine with the rest of the math in this paper. The implementation in each case would be about the same, however. It is just a notational benefit.
Chapter 4

Control Point Conversions

With these definitions we can calculate the conversions between curve types. To move from a curve with characteristic matrix $M_\beta$ to a curve with $M_\alpha$, we use the fact that the resulting curves should be the same for all values of the parameter. I.e.:

$$x_\alpha(t) = x_\beta(t)$$
$$\vec{t} M_\alpha \vec{x}_\alpha = \vec{t} M_\beta \vec{x}_\beta$$
$$M_\alpha \vec{x}_\alpha = M_\beta \vec{x}_\beta$$
$$\vec{x}_\alpha = M_\alpha^{-1} M_\beta \vec{x}_\beta$$  \hspace{1cm} (4.1)

The product of the two matrices $M_\alpha^{-1} M_\beta$, written as $M_{\beta\to\alpha}$, is a transformation from curve $\beta$’s control values into curve $\alpha$’s control values. If you consider equation 2.1, you see that the control points are multiplied by the characteristic matrix. Putting these together we can see the process clearly:

$$x(t) = \vec{t} M_\alpha \vec{x}_\alpha$$
$$= \vec{t} M_\alpha M_\alpha^{-1} M_\beta \vec{x}_\beta$$
$$= \vec{t} M_\beta \vec{x}_\beta$$

These conversions require the inverse of the characteristic matrices, so from this point on I will routinely give the inverses along as I introduce a characteristic matrix. For the three curve types we’ve seen so far, the inverse matrices are:

$$M_\beta^{-1} = \frac{1}{3} \begin{bmatrix}
3 & 1 & 3 \\
1 & 2 & 3 \\
3 & 3 & 3 \\
\end{bmatrix}$$  \hspace{1cm} (4.2)

$$M_{CR}^{-1} = \begin{bmatrix}
1 & 1 & -1 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 \\
6 & 4 & 2 & 1 \\
\end{bmatrix}$$  \hspace{1cm} (4.3)

and

$$M_H^{-1} = \begin{bmatrix}
0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 \\
3 & 2 & 1 & 0 \\
\end{bmatrix}$$  \hspace{1cm} (4.4)
4.1 Catmull-Rom to Bézier

This conversion from Catmull-Rom control values to Bézier control values is:

\[ \vec{x}_B = M_{\text{CR} \rightarrow \text{B}} \vec{x}_{\text{CR}} \]  \hspace{1cm} (4.5)

where

\[ M_{\text{CR} \rightarrow \text{B}} = M_{\text{B}}^{-1} M_{\text{CR}} = \frac{1}{6} \begin{bmatrix} 0 & 6 & 0 & 0 \\ -1 & 6 & 1 & 0 \\ 0 & 1 & 6 & -1 \\ 0 & 0 & 6 & 0 \end{bmatrix} \]  \hspace{1cm} (4.6)

4.2 Bézier to Catmull-Rom

This can be calculated as the inverse of equation 4.6, or as:

\[ M_{\text{B} \rightarrow \text{CR}} = M_{\text{CR}}^{-1} M_{\text{B}} = \frac{1}{3} \begin{bmatrix} 6 & -6 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & -6 & 6 \end{bmatrix} \]  \hspace{1cm} (4.7)

4.3 Hermite to Bézier

The Hermite to Bézier control value conversion is the transformation:

\[ M_{\text{H} \rightarrow \text{B}} = M_{\text{B}}^{-1} M_{\text{H}} = \frac{1}{3} \begin{bmatrix} 3 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 0 & 3 & 0 & -1 \\ 0 & 3 & 0 & 0 \end{bmatrix} \]  \hspace{1cm} (4.8)

4.4 Bézier to Hermite

The reverse transformation is:

\[ M_{\text{B} \rightarrow \text{H}} = M_{\text{H}}^{-1} M_{\text{B}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \]  \hspace{1cm} (4.9)

4.5 Hermite to Catmull-Rom

This is an unusual conversion to have to perform, but is just as trivial as the previous examples:

\[ M_{\text{H} \rightarrow \text{CR}} = M_{\text{CR}}^{-1} M_{\text{H}} = \begin{bmatrix} 0 & 1 & -2 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \end{bmatrix} \]  \hspace{1cm} (4.10)
4.6 Catmull-Rom to Hermite

The inverse likewise is:

\[ M_{CR \rightarrow H} = M_{H}^{-1}M_{CR} = \frac{1}{2} \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \] (4.11)

Obviously as the number of curves we consider increases the number of possible inter-conversions grows quickly. In the remainder of this paper I will only give the data for conversions to and from Bézier curves (which are more commonly needed in my experience). The other conversions can be simply calculated if required.

Chapter 5

B-Splines

B-Splines are another basic type of curve. In their cubic version, they are normally given control points $x_{-1}$, $x_0$, $x_1$, and $x_2$, as for Catmull-Rom. Again this is indicative of their use in piece-wise poly-curves. Unlike the other curves in this paper, B-Splines don’t interpolate any of their control points (see figure 5.1 for an example), but they do have higher order continuity when used in a poly-curve.

![Figure 5.1: A B-Spline curve.](image)

B-splines are defined with the characteristic matrix:

$$M_{BS} \overset{def}{=} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \quad (5.1)$$

which has an inverse of:

$$M_{BS}^{-1} = \frac{1}{3} \begin{bmatrix} 0 & 2 & -3 & 3 \\ 0 & -1 & 0 & 3 \\ 0 & 2 & 3 & 3 \\ 18 & 11 & 6 & 3 \end{bmatrix} \quad (5.2)$$

5.1 B-Spline to Bézier

$$M_{BS\rightarrow B} = M_{B}^{-1}M_{BS} = \frac{1}{6} \begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix} \quad (5.3)$$
5.2 Bézier to B-Spline

\[
M_{\text{B} \rightarrow \text{BS}} = M_{\text{BS}}^{-1}M_{\text{B}} = \begin{bmatrix}
6 & -7 & 2 & 0 \\
0 & 2 & -1 & 0 \\
0 & -1 & 2 & 0 \\
0 & 2 & -7 & 6
\end{bmatrix}
\] (5.4)
Chapter 6

Curves with Additional Parameters

A sequence of Catmull-Rom curves can be used to interpolate any number of points smoothly. There is one Catmull-Rom curve between each consecutive pair of points. The start and end points of that curve are the two points themselves. The derivative of the curve at the start and end is the difference between the previous and next control points in the sequence. So the derivative at $x_0$ is given by:

$$\dot{x}_0 = \frac{1}{2}(x_1 - x_{-1})$$

(6.1)

or more generally:

$$\dot{x}_i = \frac{1}{2}(x_{i+1} - x_{i-1})$$

(6.2)

If you look at equation 4.11 you can see this clearly, the bottom two rows of the conversion matrix show that the contribution to the derivative is half of the difference between the neighboring two points. This is part of the definition of the Catmull-Rom curve.

We can specify other families of curves, however, by calculating the derivative in a different way. Like the Catmull-Rom curve these can be used to smoothly interpolate a sequence of control values, with individual curves between each pair of points.

6.1 Cardinal Curves

The Cardinal curves are a super-set of the Catmull-Rom curve, they have:

$$\dot{x}_i = (1 - \tau)\frac{1}{2}(x_{i+1} - x_{i-1})$$

(6.3)

where $\tau$ is called the ‘tension’ parameter, and controls how smoothly the curve turns through its control points. For a Catmull-Rom curve the tension is zero, giving equation 6.2. A set of Cardinal curves are shown in figure 6.1.

Note that the effect of $\tau$ is relatively modest, so I’ve had to increase the $x$-range of this diagram compared to those previous. Super-imposed on figure 2.2, the other Cardinal curves would be within the line width of the Catmull-Rom curve. The tension parameter is better visualised with a series of Cardinal curves in a spline, as shown in the figure.
6.1 Cardinal to Hermite

Because we are calculating the derivatives, we are effectively converting from the Cardinal curve to a Hermite curve. The start and end points are the same in both cases, and the derivatives of the Hermite curve are given by equation 6.3. In matrix form we can express this as:

\[
\vec{x}_H = M_{C(\tau) \to H} \vec{x}_{C(\tau)} = \left(\begin{array}{cccc}
0 & 2 & 0 & 0 \\
0 & 0 & 2 & 0 \\
\tau - 1 & 0 & 1 - \tau & 0 \\
0 & \tau - 1 & 0 & 1 - \tau
\end{array}\right) \vec{x}_{C(\tau)}
\] (6.4)

where I’ve calculated the elements in this matrix directly, rather than from the conversion formula \( M_H^{-1} M_{C(\tau)} \).

6.1.2 Characteristic Matrix

We can therefore calculate the characteristic matrix \( M_{C(\tau)} \) by:

\[
M_{C(\tau)} = M_H M_{H}^{-1} M_{C(\tau)} = M_H M_{C(\tau) \to H}
\] (6.5)

where \( M_{C(\tau) \to H} \) is given by equation 6.4. We get:

\[
M_{C(\tau)} = \frac{1}{2} \left[ \begin{array}{cccc}
\tau - 1 & 3 + \tau & -3 - \tau & 1 - \tau \\
2 - 2\tau & -5 - \tau & 4 + 2\tau & \tau - 1 \\
\tau - 1 & 0 & 1 - \tau & 0 \\
0 & 2 & 0 & 0
\end{array} \right]
\] (6.6)

The inverse of the Cardinal characteristic matrix is:

\[
M_{C(\tau)}^{-1} = \left[ \begin{array}{cccc}
1 & 1 & 1 - \lambda & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 \\
3\lambda & 2\lambda & 1\lambda & 1
\end{array} \right]
\] (6.7)

where

\[
\lambda = \frac{2}{1 - \tau}
\]

Note that, if \( \tau = 0 \), we get the Catmull-Rom characteristic matrix (from equation 2.13) and its inverse (equation 4.3). So, as expected, the Catmull-Rom curve is the Cardinal curve with zero tension.
6.1.3 Cardinal to Bézier

The control point conversion from Cardinal curve to Bézier is:

\[ M_{C(\tau)} \rightarrow B = M_B^{-1} M_{C(\tau)} = \frac{1}{6} \begin{bmatrix} 0 & 6 & 0 & 0 \\ \tau - 1 & 6 & 1 - \tau & 0 \\ 0 & 1 - \tau & 6 & \tau - 1 \\ 0 & 0 & 6 & 0 \end{bmatrix} \] (6.8)

6.1.4 Bézier to Cardinal

The conversion from Bézier to Cardinal is:

\[ M_B \rightarrow C(\tau) = M_{C(\tau)}^{-1} M_B = \begin{bmatrix} \mu & -\mu & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & -\mu & \mu \end{bmatrix} \] (6.9)

where \( \mu \) is given by:

\[ \mu = 3\lambda = \frac{6}{1 - \tau} \]

6.2 Kochanek-Bartels Curve (TCB Curves)

We can generate other families of curve by replacing equation 6.2 with other expressions. The Kochanek-Bartels curves are produced with the tangent calculations:

\[ \dot{x}_0 = \frac{1}{2}(1 - \tau) [(1 + b)(1 + c)(x_0 - x_{-1}) + (1 - b)(1 - c)(x_1 - x_0)] \] (6.10)

\[ \dot{x}_1 = \frac{1}{2}(1 - \tau) [(1 + b)(1 - c)(x_0 - x_{-1}) + (1 - b)(1 + c)(x_1 - x_0)] \] (6.11)

Notice that there are slightly different forms for the start and end tangent (differing in the sign of \( c \)), and that the equations use the point itself \( x_0 \) as well as its neighbors.

The Kochanek-Bartels curve has three parameters: \( \tau \) is called the tension and functions similarly to the tension in the Cardinal curve; \( b \) is called the bias and controls whether the tangent is more biased towards the following point or the preceding point; and \( c \) is the continuity, which controls whether a sequence of curves will be joined smoothly or with cusps. The names of these parameters gives this curve type its alternative name: the TCB curve.

The Kochanek-Bartels curve is a super-set of the Cardinal curve. Cardinal curves are obtained when \( b = 0 \) and \( c = 0 \). The Catmull-Rom curve is therefore obtained when all three parameters are set to zero.

Figures 6.2 and 6.3 show the Kochanek-Bartels curves for varying values of its parameters. Note that a grid for the \( \tau = 1 \) case is not shown. As can be seen from equations 6.10 and 6.11, if \( \tau = 1 \) then the derivates are zero. The shape of the curve in that case is shown in the third part of figure 6.1.
Figure 6.2: A set of Kochanek-Bartels curves with $c \in \{-1, -\frac{1}{2}, 0, \frac{1}{2}, 1\}$, $b \in \{-1, 0, 1\}$ and $\tau = -1$. 
Figure 6.3: A set of Kochanek-Bartels curves with $c \in \{-1, -\frac{1}{2}, 0, \frac{1}{2}, 1\}$, $b \in \{-1, 0, 1\}$ and $\tau = 0$. 
6.2.1 Kochanek-Bartels to Hermite

Just as we did for the Cardinal curve, we can use what we know about the Kochanek-Bartels curve to construct the conversion matrix that converts Kochanek-Bartels control values into Hermite control values. We get:

\[
\vec{x}_H = M_{\text{KB}(\tau,c,b) \rightarrow H} \vec{x}_{\text{KB}(\tau,c,b)}
\]

\[
= \frac{1}{2} \begin{bmatrix}
0 & 2 & 0 & 0 \\
0 & 0 & 2 & 0 \\
-A & A - B & B & 0 \\
0 & -C & C - D & D
\end{bmatrix} \vec{x}_{\text{KB}(\tau,c,b)}
\] (6.12)

where:

\[
A = (1 - \tau)(1 + b)(1 + c) \quad (6.13)
\]
\[
B = (1 - \tau)(1 - b)(1 - c) \quad (6.14)
\]
\[
C = (1 - \tau)(1 + b)(1 - c) \quad (6.15)
\]
\[
D = (1 - \tau)(1 - b)(1 + c) \quad (6.16)
\]

6.2.2 Characteristic Matrix

In the same way as for Cardinal curves, we calculate the characteristic matrix by pre-multiplying the conversion matrix by the Hermite characteristic matrix:

\[
M_{\text{KB}(\tau,c,b)} = M_H M_{\text{KB}(\tau,c,b) \rightarrow H}
\]

\[
= \frac{1}{2} \begin{bmatrix}
-A & 4 + A - B - C & -4 + B + C - D & D \\
2A & -6 + C - 2A + 2B & 6 + D - C - 2B & -D \\
0 & A - B & B & 0 \\
0 & 2 & 0 & 0
\end{bmatrix}
\] (6.17)

using the same \(A, B, C\) and \(D\) as before. And its inverse is:

\[
M_{\text{KB}(\tau,c,b)}^{-1} = \begin{bmatrix}
B & B & B - 2 & 1 \\
\frac{A}{B} & \frac{A}{B} & \frac{B - 2}{A} & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1
\end{bmatrix}
\]

(6.18)

6.2.3 Kochanek-Bartels to Bézier

\[
M_{\text{KB}(\tau,c,b) \rightarrow B} = M_B^{-1} M_{\text{KB}(\tau,c,b)} = \frac{1}{6} \begin{bmatrix}
0 & 6 & 0 & 0 \\
-A & 6 + A - B & B & 0 \\
0 & C & 6 + D - C & -D \\
0 & 0 & 6 & 0
\end{bmatrix}
\] (6.19)
6.2.4 Bézier to Kochanek-Bartels

\[ M_{B\to KB(\tau,c,b)} = M_{KB(\tau,c,b)}^{-1} M_B = \begin{bmatrix}
\frac{6 - B + A}{A} & -\frac{6}{A} & 0 & \frac{B}{A} \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\frac{C}{D} & 0 & -\frac{6}{D} & \frac{6 - C + D}{D}
\end{bmatrix} \] (6.20)

Once again, setting \( \tau = 0, b = 0 \) and \( c = 0 \) we get \( A = 1, B = 1, C = 1, D = 1 \) and the matrices reduce to the Catmull-Rom case.
Chapter 7

Lower Order Bézier Curves

In this section we’ll depart from considering cubic curves to look at the quadratic form of the Bézier. This is used in some applications because it is faster and simpler than the cubic form. It is still a practical primitive for some modeling tasks where performance is more crucial than modeling flexibility. Figure 7.1 shows a quadratic bezier.

Figure 7.1: A quadratic Bézier curve is defined by its three control values.

The quadratic form of the single-parameter parametric curve is:

\[
x(t) = \vec{t}^2 M_\alpha \vec{x} = \begin{bmatrix} t^2 & t & 1 \end{bmatrix} M_\alpha \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}
\]

(7.1)

where \(M_\alpha\) and \(\vec{x}\) are the characteristic matrix and vector of control points as before, and \(\vec{t}\) is the row vector of powers of the parameter \(t\). Unlike for the cubic case, the two vectors now have three elements each, and the matrix has three columns and rows.

As we saw in equation 2.11, the characteristic matrix of the quadratic Bézier curve is:

\[
M_{B_2} \overset{\text{def}}{=} \begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 2 \\ 1 & 2 & 2 \end{bmatrix}
\]

Its inverse is:

\[
M_{B_2}^{-1} = \frac{1}{2} \begin{bmatrix} 1 & 2 \\ 1 & 2 \\ 2 & 2 \end{bmatrix}
\]

(7.2)

\(^1\text{It is possible to generate versions of some of the other curves in this paper for the quadratic case. I have never seen them used in practice, however, so I won’t go through the machinations of using them here.}\)
7.1 Conversion from Quadratic to Cubic Bézier

To convert from quadratic to cubic Bézier, we can’t use equation 4.1 directly. The two characteristic matrices have different dimensions. To calculate the conversion we first note that:

\[ x(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ -2 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \]  

(7.3)

In other words, a quadratic is just a cubic with zero coefficients for the \( t^3 \) term. Using \( M_B' \) to denote this extended version of the characteristic matrix for the quadratic Bézier, and using \( M_B_3 \) to denote the regular characteristic matrix for the cubic Bézier, we perform the calculation:

\[ \vec{x}_{B_3} = M^{-1}_{B_3} M_{B_2} \vec{x}_{B_2} \]  

(7.4)

This gives the conversion:

\[ \vec{x}_{B_3} = \frac{1}{3} \begin{bmatrix} 3 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 3 \end{bmatrix} \vec{x}_{B_2} \]  

(7.5)

So the cubic Bézier control points are two thirds of the way along the line from the end-points to the quadratic control point.

7.2 Conversion from Cubic to Quadratic Bézier

It is impossible to approximate a cubic curve with a quadratic, so there is no direct conversion between cubic and quadratic Bézier.

There are a number of approaches for approximation. Each involves splitting the cubic into a number of quadratic sections that approximate the original curve to within some application-defined tolerance. These solutions are algorithmic, rather than algebraic, however, and so are not covered in this paper.

7.3 Linear Elements

Straight lines can be seen as a first-order curve, and can be written in the same way as the other curves.

\[ x_L(t) = \vec{t}_1 M_L \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \]

where \( x_0 \) and \( x_1 \) are the two points on the line at parameters 0 and 1 respectively. The characteristic matrix of the line is:

\[ M_L \overset{\text{def}}{=} \begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix} \]

and its inverse is:

\[ M_L^{-1} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \]  

(7.6)
This is an eccentric way of representing lines, but it is useful when it comes to performing conversions. For example, we can follow the method in section 7.1 and extend the characteristic matrix to four rows to find the conversion matrix from a line to a cubic Bézier:

\[
M_{L \rightarrow B_3} = M_{B_3}^{-1} M'_L = M_{B_3}^{-1} \begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
-1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0
\end{bmatrix} = \frac{1}{3} \begin{bmatrix}
1 & 2 & 1 \\
1 & 2 & 0 \\
1 & 0 & 1 \\
1 & 0 & 0
\end{bmatrix}
\]

(7.7)

Once again, it should be obvious that we cannot do the opposite conversion: in general we cannot accurately convert a curve into a straight line. There are various approximation methods, however, for splitting the curve into a large number of straight line segments. As before, we will not consider them here.
Chapter 8

Subdivision

Curve subdivision is the splitting of one curve into two new curves, so that the two curves are coincident with the original. In almost all cases the two new curves are of the same type and order as the original. So we might split a cubic Bézier into two new cubic Béziers.

Although it has been suggested as a way of drawing curves\(^1\), it is terribly inefficient for that purpose. Most commonly it is used in interactive applications when a user can’t get the shape they need and want extra control over their curve. A curve is split into two, giving them higher resolution.

8.1 Changing Parameters

Subdivision can be seen as a change in the \( t \) parameter. Imagine we’re subdividing a curve at \( t = \frac{1}{2} \), and for now imagine we’re only interested in the first of the two subdivided curves. We are effectively taking a curve defined for \( 0 \leq t \leq 1 \) and mapping its parameter onto a range \( 0 \leq t' \leq 2 \). So now, the portion of the new curve from \( 0 \leq t' \leq 1 \) corresponds to the first half of the original curve.

In general starting with a curve from \( 0 \leq t \leq 1 \), we can generate a sub-curve from \( t_0 \leq t \leq t_1 \) with the change of parameter:

\[
t' = \frac{t - t_0}{t_1 - t_0}
\] (8.1)

so for a subdivision at \( t = 0.5 \) we get the change of parameters:

\[
\begin{align*}
t'_{\text{first}} &= 2t \\
t'_{\text{second}} &= 2t - 1
\end{align*}
\]

The term ‘subdivision’ refers to splitting a curve into two at some internal point (often \( t = \frac{1}{2} \)), as I’ve done above. But note that equation 8.1 is more general: it allows us to generate any sub-curve. There is no requirement to set \( t_0 = 0 \) or \( t_1 = 1 \). This more general approach is known as ‘curve trimming’. In the remains of this section we’ll consider the curve trimming problem, of which subdivision is just a special choice of \( t_0 \) and \( t_1 \).

\(^1\)The basic idea being to keep subdividing a curve until a section is flat enough to be drawn with a straight line. The logic being that areas of the curve that are almost flat require few subdivisions and areas of high curvature require more. Unfortunately ‘flat-enough’ cannot be calculated easily and the time taken in that calculation outweighs any speed-up as a result of drawing fewer lines. Because drawing extra line segments is typically very cheap and often hardware optimized, we would be better off drawing more segments than we need and saving the subdivision calculations. The most efficient way of drawing a curve is forward differencing, which we’ll meet in chapter 9. It effectively uses a fixed sampling along the curve, rather than dynamically subdividing.
In practice we want the opposite conversion to that in equation 8.1. We have the characteristic matrix and the control points of the original, untrimmed, curve. We want to map the new parameter into the old parameter so we can use equation 2.1 to evaluate the curve (we will consider how to calculate new control points for the subdivided curve below).

So the transform we’re interested in is:

\[ t = t'(t_1 - t_0) + t_0 \] (8.2)

To use this result with equation 2.1, we need to calculate the \( \vec{t} \) vector. Substituting equation 8.2 into \( \vec{t} \) we get:

\[
\vec{t} = \begin{bmatrix}
(t_1 - t_0)^3 t'^3 + 3t_0(t_1 - t_0)^2 t'^2 + 3t_0^2(t_1 - t_0)t' + t_0^3 \\
(t_1 - t_0)^2 t'^2 + 2t_0(t_1 - t_0)t' + t_0^2 \\
(t_1 - t_0)t' + t_0
\end{bmatrix}^T
\] (8.3)

This can be expressed in matrix terms as:

\[
\vec{t} = \vec{\vec{t}} \mathbf{S} = \vec{\vec{t}} \begin{bmatrix}
(t_1 - t_0)^3 & 3t_0(t_1 - t_0)^2 & (t_1 - t_0)^2 \\
3t_0^2(t_1 - t_0) & 2t_0(t_1 - t_0) & t_1 - t_0 \\
t_0^3 & t_0^2 & t_0 & 1
\end{bmatrix}
\] (8.4)

where \( \mathbf{S} \) is called the trimming matrix, for two parameters \( t_0 \) and \( t_1 \).

We can use the trimming matrix by substituting equation 8.4 into equation 2.1:

\[
x(t') = \vec{\vec{t}} \mathbf{M}_\alpha \vec{x}
\] (8.5)

The function of the trimming matrix should be clear here, it takes the \( \vec{\vec{t}} \) vector (i.e. made up of the new parameters) on its left and converts them into the old parameters so we can use the same characteristic matrix and control points as before.

The inverse of the trimming matrix \( \mathbf{S}^{-1} \) is less commonly used, it transforms the old parameters into new parameter and has the form:

\[
\mathbf{S}^{-1} = \begin{bmatrix}
d^3 & d^2 \\
-3d^3 t_0 & -2d^2 t_0 & d \\
3d^3 t_0^2 & 2d^2 t_0 & -d t_0 & 1
\end{bmatrix}
\] (8.6)

where

\[
d = \frac{1}{t_1 - t_0}
\]
The trimming matrices for other orders of curve can be calculated in the same way. The quadratic and linear matrices are sub-matrices of the cubic:

\[
S_2 = \begin{bmatrix}
(t_1 - t_0)^2 & t_1 - t_0 & t_1 - t_0 & 1 \\
2t_0(t_1 - t_0) & t_0 & t_0 & 1 \\
t_0^2 & t_0 & t_0 & 1
\end{bmatrix}
\]

and:

\[
S_1 = \begin{bmatrix}
t_1 - t_0 \\
t_0 \\
1
\end{bmatrix}
\]

with inverses:

\[
S_2^{-1} = \begin{bmatrix}
d^2 & d \\
-2d^2t_0 & -d \\
d^2t_0^2 & -dt_0 & 1
\end{bmatrix}
\]

and:

\[
S_1^{-1} = \begin{bmatrix}
d \\
-dt_0 \\
1
\end{bmatrix}
\]

with \(d\) as before.

### 8.2 Generating Subdivided Curve Coefficients

Recall from section 3 that the characteristic matrix is a conversion from some control points to the coefficients of the polynomial curve. Substituting equation 8.2 into equation 3.4, then, we get the general form:

\[
x(t') = \vec{t}' S \vec{a}
\]

So pre-multiplying the coefficients of the polynomial curve by \(S\) gives us the coefficients for the subdivided curves:

\[
\vec{a}' = S \vec{a}
\]

### 8.3 Generating Subdivided Control Points

The final thing to note is that we can easily calculate the new control points of a subdivided curve using the trimming matrix. If:

\[
\vec{a} = M_0 \vec{x}
\]

then

\[
\vec{x}' = (M_0^{-1} S M_0) \vec{x}
\]

Dedicated code for subdividing a particular curve type can store the value of the parenthesized expression as a dedicated transform for generating the control points for the subdivided curve from...
the master-curve. In the most common case of a cubic Bézier curve divided at some point \( t_s \) we have:

\[
\vec{x}_{\text{first}}' = \begin{bmatrix}
1 - t_s & t_s \\
(1 - t_s)^2 & 2t_s(1 - t_s) \\
(1 - t_s)^3 & 3t_s(1 - t_s)^2 & 3t_s^2(1 - t_s) & t_s^3
\end{bmatrix} \vec{x}
\]

\[
(8.14)
\]

\[
\vec{x}_{\text{second}}' = \begin{bmatrix}
(1 - t_s)^3 & 3t_s(1 - t_s)^2 & 3t_s^2(1 - t_s) & t_s^3 \\
(1 - t_s)^2 & 2t_s(1 - t_s) & t_s^2 \\
1 - t_s & t_s & 1
\end{bmatrix} \vec{x}
\]

\[
(8.15)
\]

And for a subdivision where \( t_s = \frac{1}{2} \) we have:

\[
\vec{x}_{\text{first}}' = \frac{1}{8} \begin{bmatrix}
8 & 4 & 4 \\
2 & 4 & 2 \\
1 & 3 & 3 & 1
\end{bmatrix} \vec{x}
\]

\[
(8.16)
\]

\[
\vec{x}_{\text{second}}' = \frac{1}{8} \begin{bmatrix}
1 & 3 & 3 & 1 \\
2 & 4 & 2 \\
4 & 4 \\
8
\end{bmatrix} \vec{x}
\]

\[
(8.17)
\]
Lots of implementations of curves are badly written and waste calculations. The most common mistake is to perform a calculation based on the control points to evaluate a curve. So we might write, for a cubic Bézier, code something like:

```c
function evaluate_bezier(x0, x1, x2, x3, t)
{
    omt = 1 - t;
    return (omt*omt*omt)*x0 + (3*omt*omt*t)*x1 + 3*(omt*t*t)*x2 + (t*t*t)*x3;
}
```

The calculation is taken directly from equation 2.4 and avoids the matrix form entirely. For libraries supporting a range of different curve types you often see APIs like:

```c
class Bezier3
{
    float x0, x1, x2, x3;
    function evaluate(t)
    {
        omt = 1 - t;
        return (omt*omt*omt)*x0 + (3*omt*omt*t)*x1 + 3*(omt*t*t)*x2 + (t*t*t)*x3;
    }
}

class CatmullRom3
{
    float x_1, x0, x1, x2;
    function evaluate(t)
    {
        return 0.5 * ( (-x_1 +3*x0 -3*x1 +x2)*t*t*t +
                       (2*x_1 -5*x0 +4*x1 -x2)*t*t +
                       (-x_1+x1)*t +
                       (2*x0));
    }
}
```

and so on for the other curve types. A commercial code-optimization tool is capable of removing the common calculations from this code and assigning them to temporary variables, but even so the code will be inefficient.
9.1 Evaluating Curves

For a better way to implement parametric curves, notice that the coefficients of the powers of \( t \) are constant for every call to \texttt{CatmullRom3::evaluate}. A much better solution to both cases is to store the coefficients for the underlying cubic curve. In terms of our matrix this means storing:

\[
\vec{a} = \mathbf{M}_\alpha \vec{x}_\alpha \tag{9.1}
\]

(where \( \vec{a} \) is the vector of coefficients, as described in section 3) With these coefficients stored we evaluate the curve with:

\[
x(t) = \vec{t} \vec{a} \tag{9.2}
\]

This allows us to write the more efficient code:

```cpp
class CatmullRom3 {
    float a, b, c, d;
    function CatmullRom3(x_1, x0, x1, x2) {
        a = 0.5 * (-x_1 + 3*x0 - 3*x1 + x2);
        b = 0.5 * (2*x_1 - 5*x0 + 4*x1 - x2);
        c = 0.5 * (-x_1 + x1);
        d = x0;
    }
    function evaluate(t) {
        return ((a*t + b)*t + c)*t + d;
    }
}
```

The factorization in the \texttt{evaluate} method uses Horner’s rule, minimizing the number of multiplications needed. This approach makes it easier and clearer to add a method such as:

```cpp
class CatmullRom3 {
    // ... as before ...
    function evaluateFirstDerivative(t) {
        return (3*a*t + 2*b)*t + c;
    }
}
```

In fact we can combine the implementations for all our cubic curves into a single cubic curve class:

```cpp
class Curve3 {
    float a, b, c, d;
    function setBezier(x0, x1, x2, x3) {
        a = -x0 + 3*x1 - 3*x2 + x3;
        b = 3*x0 - 6*x1 + 3*x2;
        c = -3*x0 + 3*x1;
        d = x0;
    }
}
```
function setCatmullRom(x_1, x0, x1, x2)
{
  a = 0.5 * (-x_1 +3*x0 -3*x1 +x2);
  b = 0.5 * (2*x_1 -5*x0 +4*x1 -x2);
  c = 0.5 * (-x_1+x1);
  d = x0;
}

function setCardinal(x_1, x0, x1, x2, tension)
{
  omt = 1 - tension;
  a = 0.5 * (-omt*x_1 +(3+tension)*x0 -(3+tension)*x1 +omt*x2);
  b = 0.5 * (2*omt*x_1 -(5+tension)*x0 +(4+2*tension)*x1 -omt*x2);
  c = 0.5 * (-omt*x_1 + omt*x1);
  d = x0;
}

// ... similarly for the other curve types ...

function evaluate(t)
{
  return ((a*t + b)*t + c)*t + d;
}

function evaluateFirstDerivative(t)
{
  return (3*a*t + 2*b)*t + c;
}

Again we could use a code-optimizer to reduce the number of calculations needed, particularly in calculating the coefficients for the setCardinal method (and if we’d implemented Kochanek-Bartels curves there would be even more potential for factorization). But in this case the calculations are only been performed when the curve is created, not at every evaluation, so there is little reason to worry too much\(^1\).

Having a single implementation for each order of curve makes it much simpler to optimize algorithms for tasks such as: evaluating many points along a curve, splitting a curve into sub-curves, calculating tangents and normals, calculating a bounding box, collision or proximity detection and efficient implementation on a GPU. All of these tasks depend only on the coefficients of the curve, not the control points, and so can benefit from a single authoritative implementation. We will look at some of these issues in the following sections.

### 9.2 Forward Differencing

To draw a parametric curve, it is typically necessary to split it into small straight line segments. These should be small enough so that the resulting curve appears to be smooth, but not so small that there are too many to draw efficiently.

There are various approaches to splitting the curve into lines, but by far the most common is the most naïve. We simply split the parameter into equal intervals, sample the curve at those

\(^1\)Of course, if your application has a curve who’s control points are altered about as often as a point on the curve is evaluated, then this approach doesn’t speed things up at all and the original approach is just as good (in fact better in the Bézier case because of the factorization used there). But in all cases I’ve come across evaluation is much more common than initialization, even for interactive graphics applications when the control points are being dragged: only one of many curves is being edited at a time, and we typically want to draw the whole curve (involving lots of evaluations) for each alteration of a control point.
points and draw lines between adjacent points. So we might want 100 lines per curve, we therefore sample at \( t = \{0, 0.01, 0.02, \ldots, 0.98, 0.99, 1.0\} \).

This approach is naïve in that it ignores the curvature. A curve that is almost a straight line will get just as many elements as a tight loop. Other approaches seek to alleviate this waste by adaptively sampling the \( t \) values to put more line segments in high curvature areas. Unfortunately performing the calculations to determine how many line segments to use is often slower than drawing a wasteful number of segments, and the naïve algorithm usually wins out.

We can optimize its efficiency, however. Rather than evaluating the curve at each point (which involves calculating the cubic equation in \( t \)), we can use a technique called forward differencing to determine the value of \( x(t + \delta) \) from \( x(t) \), \( \dot{x}(t) \) and \( \ddot{x}(t) \), where \( \delta \) is the interval we’re sampling at (0.01 in the previous example).

A forward difference is defined as:

\[
\Delta x(t) = x(t + \delta) - x(t) \tag{9.3}
\]

It is the difference in the value of the curve from one point to the next point one interval away. We calculate this iteratively, so

\[
x(t + \delta) = \Delta x(t) + x(t) \tag{9.4}
\]

So forward differencing starts with the initial value of the curve \( x(0) \), and then uses the forward difference \( \Delta x(t) \) to generate successive values of the curve.

In the following discussion I will use a subscript to indicate the order of the curve I am discussing.

### 9.2.1 Linear Forward Differences

Take the linear ‘curve’:

\[
x_1(t) = at + b
\]

The initial value of this curve is

\[
x_1(0) = b \tag{9.5}
\]

The forward difference \( \Delta x_1(t) \) is given by:

\[
\Delta x_1(t) = (a(t + \delta) + b) - (at + b)
= a\delta \tag{9.6}
\]

This forward difference is independent of the parameter \( t \), so the same forward difference is added at each iteration. In code this would look like this:

```plaintext
function evaluateInSteps(steps, action)
{
    delta = 1 / steps;
    x = b;
    Dx = delta*a;

    for (i = 0; i <= steps; i++) {
        action(x);
        x += Dx;
    }
}
```
One would not want to use this approach to draw a linear element, of course, but it still could be used to generate points along a line. In production code the action to take (such as drawing the next segment of the line) would probably form part of this method. I have represented it above as a callback function called action for generality.

9.2.2 Quadratic Forward Differences

With curves of higher order than linear the utility of the approach is clearer. For example in the quadratic curve

\[ x_2(t) = at^2 + bt + c \]

the forward difference is given by:

\[
\Delta x_2(t) = (a(t + \delta)^2 + b(t + \delta) + c) - (at^2 + bt + c) \\
= 2a\delta t + a\delta^2 + b\delta
\] (9.7)

This is no longer an equation independent of \( t \), it is linear in \( t \). For each forward difference we could evaluate equation 9.7 and use the result to increment the current value of \( x_2(t) \), as before. But that would defeat the point, since we are trying to avoid evaluating a polynomial for every sample.

Instead we use the same technique again, this time to calculate the forward difference of the forward difference. Since the forward difference is linear in \( t \), we’ve just seen that its forward difference will be constant. We write this forward difference of a forward difference as \( \Delta^2 x_2(t) \) and call it the second forward difference. Calculating this for the quadratic case we get:

\[ \Delta^2 x_2(t) = 2a\delta \]

So at each iteration we update the value of \( x_2(t) \) using the forward difference \( \Delta x_2(t) \), and update it using the second forward difference:

\[
x_2(t + \delta) = x_2(t) + \Delta x_2(t) \\
\Delta x_2(t + \delta) = \Delta x_2(t) + \Delta^2 x_2(t)
\]

In summary, the forward differences for the quadratic case are:

\[
\Delta x_2(t) = 2a\delta t + (a\delta^2 + b\delta) \\
\Delta^2 x_2(t) = 2a\delta
\] (9.8) (9.9)

and the initial values are found by substituting \( t = 0 \):

\[
x_2(0) = c \\
\Delta x_2(0) = a\delta^2 + b\delta
\] (9.10) (9.11)

With an implementation of:

```java
class Curve2 {
    function evaluateInSteps(steps, action) {
        delta = 1 / steps;
```
delta2 = delta * delta;

x = c;
Dx = delta2*a + delta*b;
D2x = 2*delta2*a;

for (i = 0; i <= steps; i++) {
    action(x);
    x += Dx;
    Dx += D2x;
}

This process can be repeated for any order of curve. For a curve of order \( n \), the \( n \)th derivative will be constant, so we’ll need \( n \) forward differences to update the curve. Each forward update has the form:

\[
\Delta^i x_n(t + \delta) = \Delta^{i-1} x_n(t) + \Delta^i x_n(t)
\]  

(9.12)

for \( i = \{1, 2, \ldots, n\} \) and \( \Delta^0 x_n(t) \equiv x_n(t) \).

9.2.3 Cubic Forward Differences

For the cubic curve:

\[
x_3(t) = at^3 + bt^2 + ct + d
\]

the forward differences are:

\[
\Delta x_3(t) = 3a\delta t^2 + (3a\delta^2 + 2b\delta) t + (a\delta^3 + b\delta^2 + c\delta)
\]  

(9.13)

\[
\Delta^2 x_3(t) = 6a\delta^2 t + (6a\delta^3 + 2b\delta^2)
\]  

(9.14)

\[
\Delta^3 x_3(t) = 6a\delta^3
\]  

(9.15)

and the initial values are therefore:

\[
x_3(0) = d
\]  

(9.16)

\[
\Delta x_3(0) = a\delta^3 + b\delta^2 + c\delta
\]  

(9.17)

\[
\Delta^2 x_3(0) = 6a\delta^3 + 2b\delta^2
\]  

(9.18)

\[
\Delta^3 x_3(0) = 6a\delta^3
\]  

(9.19)

To implement this for the cubic curve, we have:

class Curve3
{
    // ... previous code as before ...

    function evaluateInSteps(steps, action)
    {
        delta = 1 / steps;
        delta2 = delta * delta;
        delta3 = delta2 * delta;
        
        // ... code for evaluating the cubic curve ...
    }
}
x = d;
Dx = delta3*a + delta2*b + delta*c;
D2x = 6*delta3*a + 2*delta2*b;
D3x = 6*delta3*a;

for (i = 0; i <= steps; i++) {
    action(x);
    x += Dx;
    Dx += D2x;
    D2x += D3x;
}

9.3 Retrieving Control Points

In some applications, such as graphics packages, you need to have access to the control points after having specified the curve. It isn’t enough just to convert the control points into coefficients and throw them away. In this case we can store the control points along with the coefficients, possibly in a curve-specific subclass:

class Bezier3 extends Curve3
{
    float x0, x1, x2, x3;

    private function updateCoefficients()
    {
        a = -x0 +3*x1 -3*x2 +x3;
        b = 3*x0 -6*x1 +3*x2;
        c = -3*x0 +3*x1;
        d = x0;
    }
    function set x0 (x) { x0 = x; updateCoefficients(); }
    function set x1 (x) { x1 = x; updateCoefficients(); }
    function set x2 (x) { x2 = x; updateCoefficients(); }
    function set x3 (x) { x3 = x; updateCoefficients(); }
}

with similar subclasses for the other curve types.

9.3.1 Recalculating Control Points from Coefficients

Recall that the characteristic matrix of a curve transforms the control values into coefficients. Therefore the inverse of that matrix transforms the coefficients back into control values. If memory use is critical, we could avoid storing the control values and reconstruct them when needed from the coefficient. In the Bézier case, for example:

class Bezier3 extends Curve3
{
    function setControlPoints(x0, x1, x2, x3)
    {
        a = -x0 +3*x1 -3*x2 +x3;
    }
b = 3*x0 -6*x1 +3*x2;
c = -3*x0 +3*x1;
d = x0;

function getControlPoints()
{
    third = 1 / 3;
x0 = d;
x1 = third * (c + 3*d);
x2 = third * (b + 2*c + 3*d);
x3 = a+b+c+d;
    return x0, x1, x2, x3;
}

(the pseudo-code assumes some kind of multiple return mechanism is available). Round-tripping from control points to coefficients and back many times is likely to introduce obvious numerical errors, so this approach is not ideal.

The situation is worse for curve types that require additional parameters, such as the Cardinal curve or the Kochanek-Bartels curve. The additional parameters can't be reconstructed from the coefficients, and without them the control points can't be reconstructed. They would, therefore, need to be passed in as arguments to the getControlPoints method. In practice, therefore, the first approach of storing the control points alongside the coefficients is most commonly used.

9.4 Implementing Subdivision and Curve Trimming

Chapter 8 showed that trimming a curve is a matter of pre-multiplying the coefficients of the curve by a particular trimming matrix. We can therefore implement this on our Curve3 class with a method that performs the coefficient transform.

class Curve3
{
    // ... previous code as before ...

    function trimmed(t0, t1)
    {
        D = (t1 - t0);
        D2 = D * D;
        t02 = t0 * t0;

        new_a = (D2*D) * a;
        new_b = (3 * t0 * D2) * a + (D2) * b;
        new_c = (3 * t02 * D) * a + (2 * t0 * D) * b + (D) * c;
        new_d = (t02*t0) * a + (t02) * b + (t0) * c + d;

        return Curve3(new_a, new_b, new_c, new_d);
    }
}

where Curve3(a, b, c, d) is a constructor that simply sets the coefficients to the given values. Once again this can be optimized somewhat to remove common sub-expressions.

The quadratic form of this is:
class Curve2
{
    // ... previous code as before ...

    function trimmed(t0, t1)
    {
        D = (t1 - t0);
        new_a = (D*D) * a;
        new_b = (2 * t0 * D) * a + (D) * b;
        new_c = (t0*t0) * a + (t0) * b + c;
        return Curve3(new_a, new_b, new_c);
    }
}

which is the cubic form with \( a = 0 \) and the names of the other coefficients changed.

9.5 Implementing Vector Curves

So far the curve implementations in this section have been for one dimensional curves. As described in section 3.1, two or three dimensional curves are just a group of one-dimensional curves, with one curve per dimension. In this section we’ll look at some approaches to implementing curve-code that works with in multiple dimensions at once.

9.5.1 Sequential Calculation

The simplest way of implementing curves would be to calculate each dimension in turn. We could, therefore have an implementation of a two dimensional cubic curve as:

```javascript
class 2dCurve3
{
    Curve3 xAxis;
    Curve3 yAxis;

    function evaluate(t)
    {
        return xAxis.evaluate(t), yAxis.evaluate(t);
    }
}
```

with similar convenience methods for trimming the curve or extracting control points. This is a perfectly acceptable and general approach for implementing curves when the calculations are being done one operation at a time. In fact it has some important benefits for modularity and maintainability too. An approach where the calculations for the two or three axes are mixed into one method leads to a bloat in variable names and can be harder to debug and understand.

It is computationally wasteful only in that it recalculates \( t^2 \) and \( t^3 \) for the x and y axis. The same approach using forward differencing is even more marginal: it is wasteful only in the calculation of \( \delta^2 \) and \( \delta^3 \), which happens only once before differencing begins. Clearly these are small inefficiencies that may be worth the benefits in modularity.
9.5.2 Polymorphic or Templated Implementation

In many languages we don’t have to worry whether our code was using scalar values or vector values as control points, because the language provides support for polymorphic or templated algorithms. In particular, many languages allow us to defined arithmetical operators for custom types. We could therefore defined a vector type with something like:

```cpp
class Vector3d {
    float x, y, z;

    function operator-() {
        return Vector3d(-x, -y, -z);
    }
    function operator+(other) {
        return Vector3d(x+other.x, y+other.y, z+other.z);
    }
    function operator*(scalar) {
        return Vector3d(x*scalar, y*scalar, z*scalar);
    }
    // ... and so on for other operations ...
}
```

And create our `Curve3` implementation so it can accept either a scalar or a vector instance for control points. In languages such as Python or Ruby that support duck-typing, this is trivial. In languages such as C++ we might need to create a templated version of the curve class. In either case the curve code presented above will work unmodified.

There is something we can do to improve such an implementation. The first is to limit the number of operations we expect of the control points passed in. We can efficiently limit ourselves to just the three operations above: unary negation, addition between two values of the same type, and post-multiplication by a scalar. This provides a simpler interface and is easily documented. In C++ it also has the benefit of allowing the operator overloads to all be methods of the type-class (although I am aware many developers prefer to have external friend operator functions, I prefer to keep code in one place).

In order to make this work, however, we'll need to restructure the code we've written to make sure we only use these operations. For example, the `setBeziers` function from our `Curve3` implementation would look like this:

```cpp
function setBeziers(x0, x1, x2, x3) {
    a = -x0 + x1*3.0 + x2*(-3.0) + x3;
    b = x0*3.0 + x1*(-6.0) + x2*3.0;
    c = x0*(-3.0) + x1*3.0;
    d = x0;
}
```

We could further reduce the number of required operators by using `x * (-1.0)` for `-x`, but this will typically be less efficient than having all three operators available.

All our other code-snippets can be similarly refactored.
9.5.3 Vectorization and GPU Implementation

In many situations neither approach above will be good enough: we want to perform calculations as quickly as possible, and we want to take advantage of hardware that can perform more than one calculation at a time. That means we’ll be doing our calculations either on the SIMD units of a regular CPU (MMX, SSE1 or SSE2 for the Intel naming scheme, other processors have similar facilities), or on the GPU (as part of a shader program or a GPGPU program). In each of these cases we can perform multiple calculations simultaneously. In particular they are often used to perform four floating point calculations in parallel.

To evaluate:

\[
\begin{align*}
  x &= a_xt^3 + b_xt^2 + c_xt + d_x \\
  y &= a_yt^3 + b_yt^2 + c_yt + d_y \\
  z &= a_zt^3 + b_zt^2 + c_zt + d_z
\end{align*}
\]

we structure the calculation so that we have SIMD variables containing:

\[
\begin{align*}
  a_v &= [a_x, a_y, a_z, 0]; \\
  b_v &= [b_x, b_y, b_z, 0]; \\
  c_v &= [c_x, c_y, c_z, 0]; \\
  d_v &= [d_x, d_y, d_z, 0];
\end{align*}
\]

where the \([e, f, g, h]\) notation indicates placing the four given values into the successive floating point slots of a 4-slot SIMD variable.\(^2\) We can then treat these vectorized coefficients as single coefficients and perform the evaluation by doing:

\[
\begin{align*}
  \text{function evaluate}(t) \\
  \{ \\
  &tv = [t, t, t, t]; \\
  &t2v = tv \times tv; \\
  &t3v = t2v \times tv; \\
  &result = av*t3v + bv*t2v + cv*tv + dv; \\
  &\text{return result[0], result[1], result[2];}
\}
\]

where the intended multiplication, \(\times\), is component-wise and the square bracket notation indicates extracting values from the vectorized variable. I’ve compiled a SIMD vector of \(t\) values and used them to generate a vector of \(t^2\) and \(t^3\) vectors. This is normally faster than the converse:

\[
\begin{align*}
  t2 &= t \times t; \\
  t3 &= t2 \times t; \\
  tv &= [t, t, t, t]; \\
  t2v &= [t2, t2, t2, t2]; \\
  t3v &= [t3, t3, t3, t3];
\end{align*}
\]

because the compilation of a vector is costly compared to the difference between vector and scalar multiplication.

\(^2\)Note that because there are four slots, and only three dimensions, I have filled the remaining value with 0, any other value can also be used. In particular many 3d systems use 1 for the last value, for reasons we won’t consider here. For systems in two dimensions it is possible to use the same approach, or to use the two extra slots to perform a whole other curve evaluation at the same time. This is particularly useful when evaluating curved surfaces when pairs of curves are often evaluated in parallel.
Many well implemented graphics systems treat vectors as a single opaque type, consisting of three coordinates and a padding value (the 0 in our example above, but more commonly set to 1). This structure makes it very easy to load the vector into a single SIMD-variable without further manipulation. In this case we need do nothing special to vectorize our coefficients, \(a-d\), or to de-vectorize our result by extracting the \(x, y,\) and \(z\) components of \(\text{result}\). We will, however, still need to create the vector form of the parameter \(t\), and its powers, as that is always a single scalar value.

Forward differencing works in a vectorized implementation in very much the same way as curve evaluation:

```plaintext
function evaluateInSteps(steps, action)
{
    delta = 1 / steps;
    deltav = [delta, delta, delta, delta];
    delta2v = deltav * deltav;
    delta3v = delta2v * deltav; k6v = [6,6,6,6];

    v = dv;
    Dv = delta3v*av + delta2v*bv + delta*cv;
    D2v = k6v*delta3v*av + [2,2,2,2]*delta2v*bv;
    D3v = k6v*delta3v*av;

    for (i = 0; i <= steps; i++) {
        action(v);
        v += Dv;
        Dv += D2v;
        D2v += D3v;
    }
}
```

again requiring some vectorizing before using the same algorithm as before.

Finally trimming once again works similarly:

```plaintext
function trimmed(t0, t1)
{
    D = 1.0 / (t1 - t0);
    Dv = [D, D, D, D];
    D2v = Dv * Dv;
    D3v = D2v * Dv;
    t0v = [t0, t0, t0, t0];
    t02v = t0v * t0v;
    k3v = [3, 3, 3, 3];

    new_av = (D3v) * av;
    new_bv = (-k3v * t0v * D3v) * av + (D2v) * bv;
    new_cv = (k3v * t02v * D3v) * av - ([2,2,2,2] * t0v * D2v) * bv + (Dv) * cv;
    new_dv = (-t02v*t0v * D3v) * av + (D2v * t02v) * bv - (Dv*t0v) * cv + dv;

    return Curve3(new_av, new_bv, new_cv, new_dv);
}
```
Chapter 10

Epilogue

I am conscious of the many, many topics that I haven’t touched in this paper. The most prominent, I think, are non-uniform splines; and rational curves. Put together with B-Splines, described in chapter 5, these two additional features produce the mighty NURBS, which are very widely used in graphics applications. My sense is, however, that the extreme flexibility of NURBS is often overkill, and their use is waning in favor of other techniques such as Catmull-Clark subdivision.

Because my implementation interests are mostly for real-time applications, I have not needed a reference for the mathematics of these curve types in the same way. I may get round to adding them to this paper, but for now I apologize for their absence.

A more serious deficiency in this paper is the lack of discussion of curved surfaces. I have needed curved surfaces more often than linear curves, in real time applications. Fortunately the mathematics for curved surfaces derives directly from the mathematics in this paper, using the same set of characteristic, trimming and conversion matrices. I hope to return to the topic of surfaces in another paper, in particular to illustrate caveats in their implementation. The mathematical data in such a paper, however, will be a rather modest extension to that here.

---

1. A non-uniform spline is made up of individual curves that use parameter intervals other than $t \in [0,1]$ and that are themselves controllable. This is a different approach to control the curvature, in contrast with curves such as the Cardinal and Kochanek-Bartels curves.

2. A rational curve is one that has the form

$$z(t) = \frac{at^3 + bt^2 + ct + d}{et^3 + ft^2 + gt + h}$$

(contrast that with equation 3.1) In other words it is a ratio of two regular cubic curves. Once more this addition gives more control over the curve shape.