Drawing Parametric Curves and Splines

Parametric curves are defined as vector functions of some scalar parameter t running from 0 to 1. Mathematically, a point (x, y) on a curve is computed using two functions f_1 and f_2 of the parameter t.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f_1(t) \\ f_2(t) \end{bmatrix}$$

The shape of the curve depends on the form of the functions f_1 and f_2. For example if the two functions are linear in t, the curve is a line; if they are of 2nd order degree, the curve is a quadratic curve and so on.

Parametric lines:

The general form of parametric line equation is:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \alpha_1 t + \beta_1 \\ \alpha_2 t + \beta_2 \end{bmatrix}$$

To get the coefficients $(\alpha_1, \beta_1, \alpha_2, \beta_2)$, we may constrain the line to pass through two end points: $P_0 = (x_0, y_0)$ when t=0 and $P_1 = (x_1, y_1)$ when t=1. Substituting in the above equation yields:

$$\begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} \alpha_1(0) + \beta_1 \\ \alpha_2(0) + \beta_2 \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \alpha_1(1) + \beta_1 \\ \alpha_2(1) + \beta_2 \end{bmatrix} = \begin{bmatrix} \alpha_1 + \beta_1 \\ \alpha_2 + \beta_2 \end{bmatrix}$$

From which we have four equations in four unknowns. Note that we can solve for α_1, β_1 using only the two constrains involving x which are:

$$x_0 = \beta_1$$

$$x_1 = \alpha_1 + \beta_1$$

Getting:

$$\beta_1 = x_0 \; and \; \alpha_1 = x_1 - \beta_1 = x_1 - x_0$$

So:

$$x = (x_1 - x_0)t + x_0$$

In the same way we get:

$$y = (y_1 - y_0)t + y_0$$

Note also that we don't have to work on y; after all, the formula of y is the same as that of x except that every x is replaced with y. This always happens when the two functions f_1 and f_2 have the same form (of course with different coefficients in general). The equations can also be written as follows:

$$x = (1 - t)x_0 + tx_1$$

$$y = (1-t)y_0 + ty_1$$

Here we notice that each of the end points is multiplied by a function of t that weighs its contribution to the point at t. This function is called 'basis function' or 'blending function'. Basis functions for the line case are:

$$b_0(t) = 1 - t$$

$$b_1(t) = t$$

Cubic Hermite Curves

A third order (cubic) parametric curve is given by:

$$x(t) = \alpha_1 t^3 + \beta_1 t^2 + \gamma_1 t + \delta_1$$

$$y(t) = \alpha_2 t^3 + \beta_2 t^2 + \gamma_2 t + \delta_2$$

The derivatives with respect to t are:

$$x'(t) = 3\alpha_1 t^2 + 2\beta_1 t + \gamma_1$$

$$y'(t) = 3\alpha_2 t^3 + 2\beta_2 t^2 + \gamma_2$$

Hermite curves use four constraints to compute the 4 coefficients of each function. We'll derive the coefficients of x(t) only because those of y(t) will follow a similar procedure. The constraints on x(t) are:

$$x(0) = x_0$$

$$x'(0) = s_0$$

$$x(1) = x_1$$

$$x'(1) = s_1$$

Substituting in x(t) and x'(t):

$$\delta_1 = x_0$$

$$\gamma_1 = s_0$$

$$\alpha_1 + \beta_1 + \gamma_1 + \delta_1 = x_1$$

$$3\alpha_1 + 2\beta_1 + \gamma_1 = s_1$$

In matrix form:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \beta_1 \\ \gamma_1 \\ \delta_1 \end{bmatrix} = \begin{bmatrix} x_0 \\ s_0 \\ x_1 \\ s_1 \end{bmatrix}$$

Solving for the coefficients:

$$\begin{bmatrix}\alpha_1\\\beta_1\\\gamma_1\\\delta_1\end{bmatrix}=\begin{bmatrix}0&0&0&1\\0&0&1&0\\1&1&1&1\\3&2&1&0\end{bmatrix}^{-1}\begin{bmatrix}x_0\\s_0\\x_1\\s_1\end{bmatrix}=\begin{bmatrix}2&1&-2&1\\-3&-2&3&-1\\0&1&0&0\\1&0&0&0\end{bmatrix}\begin{bmatrix}x_0\\s_0\\x_1\\s_1\end{bmatrix}$$

Now we can write x(t) in a matrix form as follows:

$$x(t)=\alpha_1t^3+\beta_1t^2+\gamma_1t+\delta_1=[t^3\quad t^2\quad t\quad 1]\begin{bmatrix}\alpha_1\\\beta_1\\\gamma_1\\\delta_1\end{bmatrix}$$

So:

$$x(t)=[t^3\quad t^2\quad t\quad 1]\begin{bmatrix}2&1&-2&1\\-3&-2&3&-1\\0&1&0&0\\1&0&0&0\end{bmatrix}\begin{bmatrix}x_0\\s_0\\x_1\\s_1\end{bmatrix}$$

This is equivalent to:

$$x(t)=(2t^3-3t^2+1)x_0+(t^3-2t^2+t)s_0+(-2t^3+3t^2)x_1+(t^3-t^2)s_1$$

The matrix $\begin{bmatrix}2&1&-2&1\\-3&-2&3&-1\\0&1&0&0\\1&0&0&0\end{bmatrix}$ is called 'basis matrix'. If it is left-multiplied by the raw vector $[t^3\quad t^2\quad t\quad 1]$ the result is a raw vector with Hermite's basis (blending) functions:

$$[2t^3-3t^2+1,t^3-2t^2+t,-2t^3+3t^2,t^3-t^2]$$

There's many ways to compute x(t). The efficient way is to use the matrix form and start by right-multiplying the basis matrix by the input vector before the loop.

In Visual C++, we may build matrix and vector classes or structures to use them in the implementation (we'll need them also when we talk about transformation in next lectures).

```
struct Vector2
{
	double x,y;
	Vector2(double a=0,double b=0)
	{
		x=a;	y=b;
	}
};
class Vector4
{
	double v[4];
public:
	Vector4(double a=0,double b=0,double c=0,double d=0)
	{
		v[0]=a;		v[1]=b;		v[2]=c;		v[3]=d;
	}
	Vector4(double a[])
	{
		memcpy(v,a,4*sizeof(double));
	}
	double& operator[](int i)
	{
		return v[i];
	}
};
class Matrix4
{
	Vector4	M[4];
public:
	Matrix4(double A[])
	{
		memcpy(M,A,16*sizeof(double));
	}
	Vector4& operator[](int i)
	{
		return M[i];
	}
};
```

The following Matrix and Vector utility functions are also useful in our implementation

```
Vector4 operator*(Matrix4 M,Vector4& b) // right multiplication of M by b
{
	Vector4 res;
	for(int i=0;i<4;i++)
		for(int j=0;j<4;j++)
			res[i]+=M[i][j]*b[j];
	return res;
}

double DotProduct(Vector4& a,Vector4& b) //multiplying a raw vector by a column vector
{
	return a[0]*b[0]+a[1]*b[1]+a[2]*b[2]+a[3]*b[3];
}
```

The implementation of the Hermite curve uses the following utility function to compute the coefficients and store them in a 4D vector.

```
Vector4 GetHermiteCoeff(double x0,double s0,double x1,double s1)
{
	static double H[16]={2,1,-2,1,-3,-2,3,-1,0,1,0,0,1,0,0,0};
	static Matrix4 basis(H);
	Vector4 v(x0,s0,x1,s1);
	return basis*v;
}
```

The main Hermite curve drawing is thus implemented as:

```
void DrawHermiteCurve(HDC hdc,Vector2& P0,Vector2& T0,Vector2& P1,Vector2& T1 ,int
numpoints, COLORREF color)
{
	Vector4 xcoeff=GetHermiteCoeff(P0.x,T0.x,P1.x,T1.x);
	Vector4 ycoeff=GetHermiteCoeff(P0.y,T0.y,P1.y,T1.y);
	if(numpoints<2)return;
	double dt=1.0/(numpoints-1);
	for(double t=0;t<=1;t+=dt)
	{
		Vector4 vt;
		vt[3]=1;
```

```
            for(int i=2;i>=0;i--)vt[i]=vt[i+1]*t;
            int x=round(DotProduct(xcoeff,vt));
            int y=round(DotProduct(xcoeff,vt));
            SetPixel(hdc,x,y,color);
        }
}
```

The algorithm uses the parameter 'numpoints', that means number of points sampled from the curve function, to compute the step by which t increases in the loop. Note that it is better to use the line drawing function instead of SetPixel in order to guarantee the connectivity of the curve. Here is the implementation using the Windows API functions of line drawing: MoveToEx and LineTo:

```
void DrawHermiteCurve (HDC hdc,Vector2& P0,Vector2& T0,Vector2& P1,Vector2& T1 ,int
numpoints)
{
        Vector4 xcoeff=GetHermiteCoeff(P0.x,T0.x,P1.x,T1.x);
        Vector4 ycoeff=GetHermiteCoeff(P0.y,T0.y,P1.y,T1.y);
        if(numpoints<2)return;
        double dt=1.0/(numpoints-1);
        for(double t=0;t<=1;t+=dt)
        {
            Vector4 vt;
            vt[3]=1;
            for(int i=2;i>=0;i--)vt[i]=vt[i+1]*t;
            int x=round(DotProduct(xcoeff,vt));
            int y=round(DotProduct(ycoeff,vt));
            if(t==0)MoveToEx(hdc,x,y,NULL);else LineTo(hdc,x,y);
        }
}
```

The problem with the Hermite curve algorithm is the interpretation of the tangent (derivative) vectors T0 and T1. In computer animation it may be used to represent the velocity of the moving object at t=0 and t=1 respectively while P0 and P1 represent the position of the moving object at t=0 and t=1. The algorithm in this case is used to interpolate the motion of the object at values of t between 0 and 1. This means that t is a scaled version of the time during which the animation takes place. In curve drawing applications however, it is difficult for the designer to decide the correct values of T0 and T1 for some specific curve.

Bezier Curves

Bezier curves are special cases of Hermit curves in which T_0 and T_1 are computed from four points P_0, P_1, P_2, P_3 as follows:

$$T_0 = 3(P_1 - P_0)$$

$$T_1 = 3(P_3 - P_2)$$

The curve starts at P_0 when t=0 and ends at P_3 when t=1. This has a geometrical interpretation if we consider the individual components of the four points for example the x components i.e. x_0, x_1, x_2, x_3. Bezier assumes that $x=x_0$ when t=0, $x=x_1$ when t=1/3, $x=x_2$ when t=2/3 and $x=x_3$ when t=1. So we have four points in the T-X plane: (0, x_0), (1/3, x_1), (2/3, x_2) and (1, x_3). The line connecting the point (0, x_0) to (1/3, x_1) is tangent to the curve at t=0 so $x'(0)$equals the slope of this line; i.e.:

$$x'(0) = \frac{x_1 - x_0}{\frac{1}{3} - 0} = 3(x_1 - x_0)$$

Likewise, the line connecting (2/3, x_2) to (1, x_3) is tangent to the curve at t=1 so $x'(1)$equals the slope of the line at t=1; i.e.

$$x'(1) = \frac{x_3 - x_2}{\frac{2}{3} - \frac{1}{3}} = 3(x_3 - x_2)$$

The curve will not pass through P_1 and P_2. It lies in the 'convex hull' of the polygon P_0-P_1-P_2-P_3. The following is an implementation of Bezier curve that calls the Hermite curve algorithm:

```
void DrawBezierCurve(HDC hdc,Vector2& P0,Vector2& P1,Vector2& P2,Vector2& P3,int
numpoints)
{
        Vector2 T0(3*(P1.x-P0.x),3*(P1.y-P0.y));
        Vector2 T1(3*(P3.x-P2.x),3*(P3.y-P2.y));
        DrawHermiteCurve(hdc,P0,T0,P3,T1,numpoints);
}
```

Cardinal splines

If we are given a set of points P_0, P_1, P_2, …, P_n, we can draw a curve passing through P_1, P_2,…,P_{n-1} by calling the Hermite curve drawing algorithm for every interval P_i-P_{i+1} , i=1,2,..,n-2. The slope at P_i is given by:

$$T_i = (1-c)(P_{i+1} - P_{i-1})$$

c is called the 'tension' of the curve that takes values from 0 to 1. The following function shows the implementation of this algorithm. Note that the algorithm does not draw the first and last interval because it cannot compute the tangents at these points.

```
void DrawCardinalSpline(HDC hdc,Vector2 P[],int n,double c,int numpix)
{
        double c1=1-c;
        Vector2 T0(c1*(P[2].x-P[0].x),c1*(P[2].y-P[0].y));
        for(int i=2;i<n-1;i++)
        {
        Vector2 T1(c1*(P[i+1].x-P[i-1].x),c1*(P[i+1].y-P[i-1].y));
        DrawHermiteCurve(hdc,P[i-1],T0,P[i],T1,numpix);
        T0=T1;
        }
}
```