Finding the Minimum Distance Between a Point and a Cubic Spline

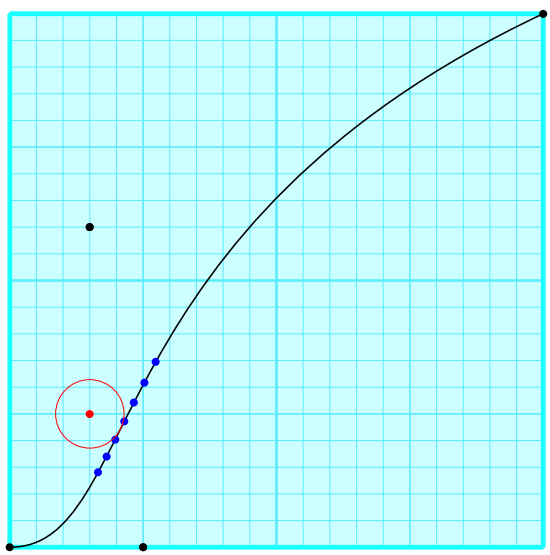
Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

http://www.tinaja.com
don@tinaja.com
(928) 428-4073

As we have seen **here**, **here**, **here**, **here** and **here**, **cubic splines** and their **Bezier Curve** variants are a very efficient, very sparse, and very powerful way to draw smooth continuous curves essential for typography, animation, and graphics.

An essential recurring question is how you quickly **find the minimum distance between a point in a plane and a given cubic spline**. This becomes important when **least squares fitting fuzzy data** or when modifying a spline to guarantee it eventually passes through a given point.

There is no known single pass deterministic solution to this problem, so repeated approximations have to be used. What follows is based on **this paper**. A typical minimum distance problem might look something like this....



We see a single cubic spline going from **x=0, y=0** to **x=20, y=20** as its **t** value ranges from **t=0** to **t=1**. The control points appear as black dots.

We also see a red point at **3, 5** whose nearest distance we seek. To keep things simple, we will assume that the point is **very near** the curve. And that it has only **one** minimum distance which lies somewhere between **t=0** and **t=1**. And that we are not yet excessively concerned about speed or algorithm efficiency.

We might simply guess **t=0.3** as a good minimum, as per the lowest blue dot. You would then find **x(t)** and **y(t)**. And then subtract them to find both the current **x** distance and the **y** distance to the point being measured. The vector distance will be the square root of the sum of the squares of the **x** and the **y** errors.

Doing so would give us a distance of **2.214**. Which is not half bad, but obviously not the minimum. We could then continue increasing **t** till we pass the minimum, reverse our path with smaller **t** increments, and keep going till we get near our correct distance value of **1.28305**. At **t=0.384982**.

Instead, we can work much smarter and greatly reduce the number of trips we will need. Minimizing the distance squared does the same thing as minimizing the distance and eliminates any nasty square root radicals. We can find the expression for the current distance as a function of **t** and then find its derivative slope. Setting this slope to zero would give us our needed minimum.

Unfortunately, the slope expression is a directly unsolvable fifth order polynomial. Fortunately, we have a sneaky trick called **Newton's Method** to deal with it.

Newton's Method works well whenever you are already **very near** to the solution of a **well behaved** function…

NEWTON'S METHOD OF SOLVING UGLY EQUATIONS —

Better guess = good guess - (function)/(function slope)

Which leads to this easy-to-use minimum distance guess improver…

TO IMPROVE A MINIMUM DISTANCE SPLINE GUESS —

Better t guess = present t guess -

[(deltax)*x'(t) + (deltay)*y'(t)] /

[(x'(t))^2 + (y'(t))^2 + (deltax)*x"(t) + (deltay)*y"(t)]

Where **deltax** and **deltay** are the present error distances. **x(t)** is **At^3 + Bt^2 + Ct + D**, **x'(t) is 3At2 + 2Bt + C**, and **x"(t) is 6At + 2B**.

Similarly, **y(t)** is **Et^3 + Ft^2 +Gt + H**, **y′(t) is 3Et2 + 2Ft + G**, and **y″(t)** is **6Et + 2F**.

Huh? Let's try to repeat this in English…

> **TO IMPROVE A MINIMUM DISTANCE SPLINE GUESS —**
>
> **To make a better guess of the t value needed for a minimum distance to a point, REDUCE t by a fraction...**
>
> **whose numerator is the present x error times the present x′(t) slope PLUS the present y error times the present y(′t) slope.**
>
> **and whose denominator consists of four summed terms of x′(t) slope squared PLUS y′(t) slope squared PLUS the x error times the present x″(t)slope-of-slope PLUS the y error times the present y″(t) slope-of-slope.**

BTW, equation **(5)** in the **original paper** is not nearly as scary and obtuse as it looks. This is simply the function divided by its slope. As to the mystery terms in the denominator, note that the derivative of **x*y** is **x(dy) + y(dx)**. And that **x′(t)** times the derivative of a function which coincidentally also is **x′(t)** produces an **x′(t)^2** term. And the same goes for **y**.

When you make this improved guess, you will find convergence to be very fast if you are already very close. in this case, an initial guess and repeated improved guesses give us results of…

t = 0.30000 s = 2.21409
t = 0.41878 s = 1.51473
t = 0.38769 s = 1.28461
t = 0.38498 s = 1.28305
t = 0.38496 s = 1.28305

Thus, a single pass is "good enough" if you are already really close, but five or more passes might be needed on a poorer first guess. And a really bad guess might not even converge at all.

I've added a red checking circle of radius **1.28305** to our test point. On extreme magnification, it seems to verify these equations by exactly hitting our spline curve precisely and tangentially at the expected **t** value.

Some Code

What we have seen so far can be calculated manually or in most any computer language. Naturally, I overwhelmingly prefer use of the exceptionally versatile **PostScript** general purpose computing language and my **Gonzo Utilities**.

Detailed code examples appear in figure1 (for gg80) of the **sourcecode** to this **GuruGram**. We will only excerpt some key code concepts here.

Here is the derivation of how you **relate spline power coefficients to control points**. The code to do so might go like this…

```
/findAH {
        /A x3 x2 3 mul sub x1 3 mul add x0 sub store
        /E y3 y2 3 mul sub y1 3 mul add y0 sub store
        /B x2 3 mul x1 6 mul sub x0 3 mul add  store
        /F y2 3 mul y1 6 mul sub y0 3 mul add  store
        /C x1 3 mul x0 3 mul sub store
        /G y1 3 mul y0 3 mul sub store
        /D x0 store    /H y0 store
             } store
```

And here is the "cubeless" method of finding x(t) and y(t) and their derivatives…

```
/findx {/ttt exch store A ttt mul B add ttt mul C add ttt
  mul D add /curx exch store} store

/findy {/ttt exch store E ttt mul F add ttt mul G add ttt
  mul H add /cury exch store} store

/finds {/curs curx xx sub dup mul cury yy sub dup mul add
  sqrt store} store

/xslope {tt dup mul 3 mul A mul tt 2 mul B mul add C
  add} store

/xslopeslope {tt 6 mul A mul B 2 mul add} store

/yslope {tt dup mul 3 mul E mul tt 2 mul F mul add G
 add} store

/yslopeslope  {tt 6 mul E mul F 2 mul add store}
```

Note that you do not have to recalculate intermediate distances if you know ahead of time that you are going to make five successive **t** improvements. You need only make the **finds** calculation **after** your best guess.

The code is reasonably fast and can be further improved. Some **alternative enhancements** are available if you need extreme speed for real time animation.

Finally, here is the **t guess improver** code…

```
/makebetterguess {
        xslope xslopeslope
        yslope yslopeslope
        /deltax curx xx sub store
        /deltay cury yy sub store

        deltax xslope mul
        deltay yslope mul add

        xslope dup mul
        yslope dup mul add
        deltax xslopeslope mul add
        deltay yslopeslope mul add   div

        /tadj exch store
        /tt tt tadj sub store
             } store
```

For More Help

Similar tutorials and additional support materials are found on our **Cubic Spline**, **PostScript** and our **GurGram** library pages. As always, **Custom Consulting** is available on a cash and carry or contract basis. As are seminars.

For details, you can email **don@tinaja.com**. Or call **(928) 428-4073**.