TechNote TN-06-003

Curve-Constrained Scrolling Via Script

Jim Armstrong
Singularity
March 2006

This is the seventh in a series of TechNotes on the subject of applied curve mathematics in Adobe Flash^{TM}. Each TechNote provides the mathematical foundation for a set of Actionscript examples.

Application of Curve Mathematics to an Actual Problem

This TechNote illustrates how to apply methods introduced in previous discussions to create specific curves to solve a specific problem. This TechNote discusses the issue of scrolling along a curved path.

The problem of interest arose from an actual application. Time constraints forced a primitive solution, however, the solution allowed an artist to make timeline-based adjustments. Sometimes, project constraints force a mix of scripted and timeline animation. In this TechNote, I wanted to revisit the problem from the standpoint of a purely mathematical and script-based approach.

Scrolling Along a Curve

In recent years, Flash interfaces have become more organic. Rectangular and trapezoidal buttons have been replaced by complex, curved interface elements. In some interfaces, this includes curved lines and arrows that represent scrollable controls. We have all seen examples of scrollbars that move in a strictly horizontal or vertical line. The example in this paper deals with constraining a scrollable UI element along a curve.

Timeline Approach

The screen shot, below, illustrates an example from an actual application. Time constraints and the need to provide a means for artists to make quick changes dictated a mixture of timeline animation and script.

The visual scrolling indicator was a circle that was constrained to move along a curved path, indicated by an arrow graphic. When the viewer clicked on one of the up/down arrow controls, content scrolled by a fixed number of pixels in the required direction. The circular graphic moved along the path in proportion to the remaining (invisible) content. When the viewer left-clicks on the circle and holds the mouse button down, content can be ‘speed-scrolled’ just like a browser. The difference is that the scrolling indicator (circle) is always constrained to move along the curved path.

A simple solution to this problem is to create a path in the Flash IDE that overlays the curved arrow path. The beginning and ending points of this path correspond to the beginning and ending levels of the scroll indicator. Use the path as a motion guide.

In the screen shot below, the path animation was converted to keyframe animation. Onion-skinning was turned on so that an artist could adjust the position of each keyframe in the event there was an issue with the motion guide.

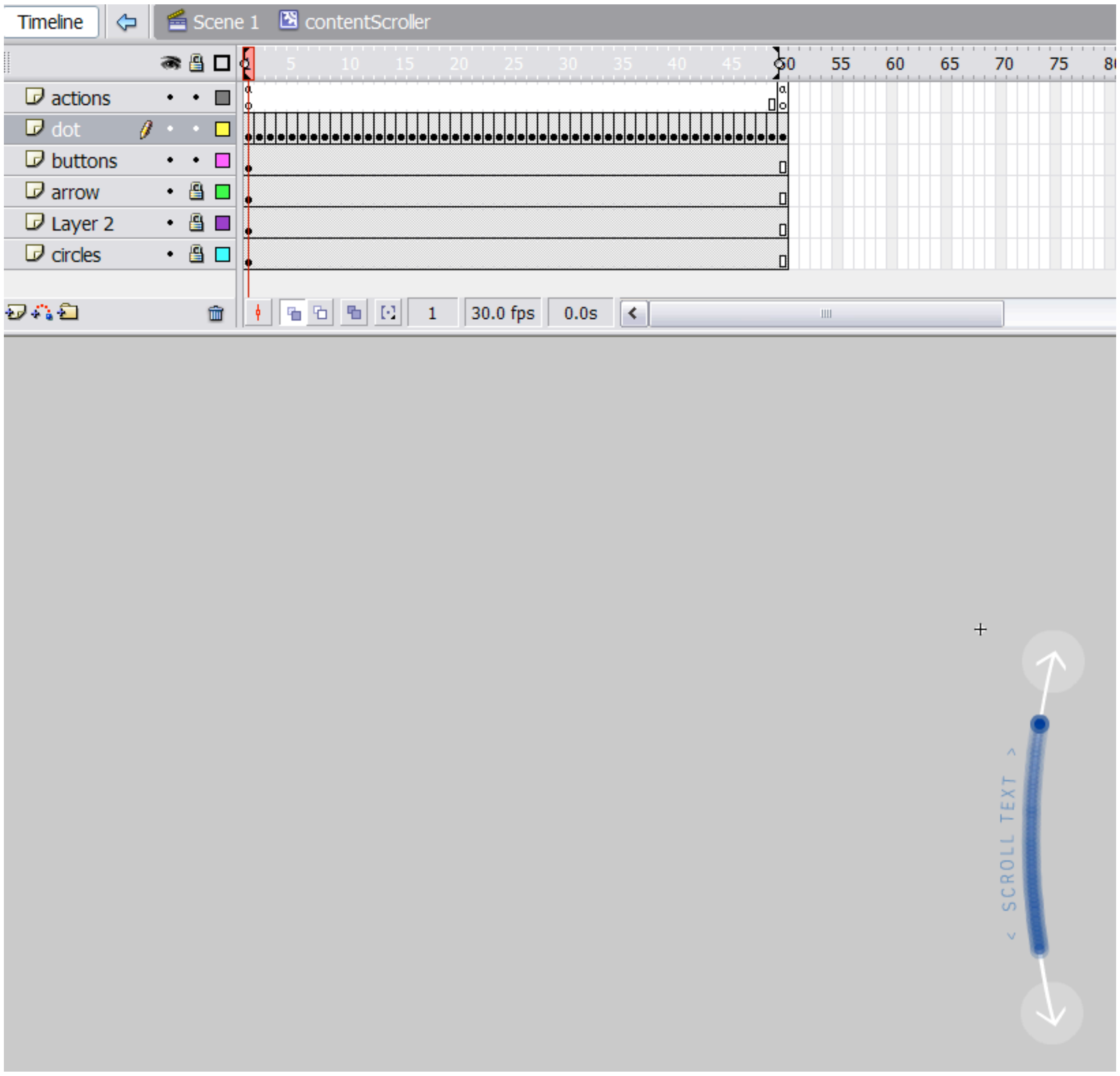


Diagram 1: Motion along a prescribed path.

When the scrolling clip is attached to a parent *MovieClip*, the extremes of the 'dot' indicator correspond to some coordinates y_1 and y_2 in the parent clip's coordinate space. When the indicator is pressed, the *onPress()* handler assigns an *onEnterFrame()* handler that maps the current y-coordinate to the interval [0,1]. A value of zero corresponds to $y = y_1$ and a value of one corresponds to $y = y_2$. Coordinates outside the scroll indicator's range are mapped to either zero or one so that the scroller never goes 'out of range.'

The problem now becomes one of mapping the current indicator in [0,1] into a frame of the timeline animation. An indicator value of zero is mapped to the first frame. An indicator value of 0.5 is mapped to the 'middle' frame of the animation. An indicator value of one is mapped to the final frame of the animation. As the mouse is moved up and down, the 'dot' appears to move along the curve. The indicator value can also be used to set the y-coordinate of the scrollable content.

This approach has the advantage of being easy to setup and script. Simplicity, however, comes with a cost. It is not easy to determine in advance how many frames are required for the motion guide to produce smooth movement along the curve. In the above example, there is not much curvature to the arrow, yet 50 frames of path animation were required.

Some modest amount of effort is required to determine the range of y-coordinates spanned by the limits of the scroll indicator in the parent's coordinate space. If the interface ever changes, these coordinates must be re-computed. It is possible to place invisible 'marker' clips at the appropriate places and then use *localToGlobal()* or John Grden's *localToLocal()* function [1] to compute the required range.

Even worse, if the scroll arrow ever changes, the path animation must be recreated from scratch. The timeline method is well suited for quickly developing a demo with fixed assets, but is not friendly to changes.

Representing A Path With Parametric Curves

A more general-purpose approach that is very change-friendly is to represent the path with a parametric curve. As before, y-coordinate values are mapped to [0,1]. This parameter is used to compute the x-coordinate of a parametric curve representing the scroll path.

The simplest curve is a parametric quadratic of the form

$$P(t) = c_0 + c_1 t + c_2 t^2 \qquad [1]$$

Three conditions are required in order to compute the coefficients in [1]. The easiest conditions to enforce are for the curve to pass through three specific points. Two of the points correspond to the endpoints of the scroll path. The third point represents the 'midpoint' of the path, as illustrated in the following diagram.

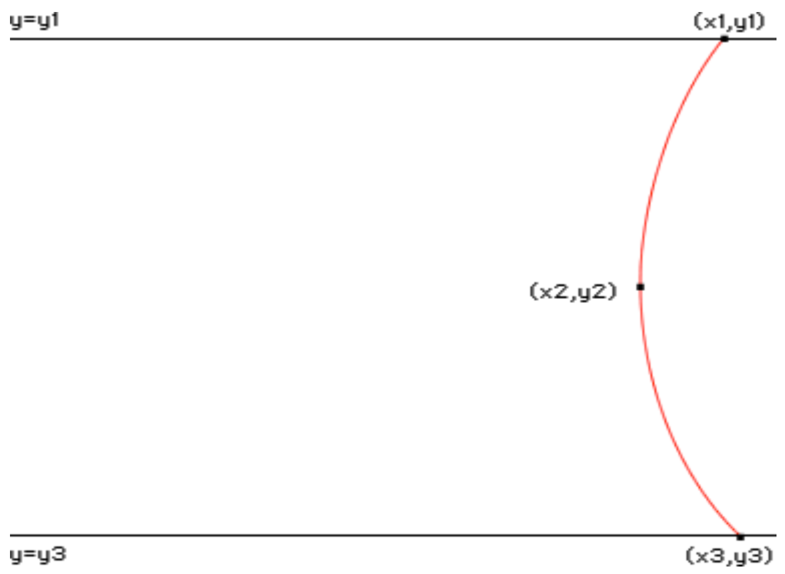


Diagram 2: Parametric quadratic curve passing through three points.

Let $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, and $P_3 = (x_3, y_3)$. The parametric curve passes through each of these points at pre-specified parameter values. Since the second point is intended to be a 'midpoint' of the curve, let P_1 correspond to $t = 0$, P_2 correspond to $t = 0.5$, and P_3 correspond to $t = 1$. Substituting these values into [1] yields the system of equations,

$$\begin{aligned} c_0 &= P_1 \\ c_0 + c_1/2 + c_2/4 &= P_2 \qquad [2] \\ c_0 + c_1 + c_2 &= P_3 \end{aligned}$$

which quickly reduces to

$$\begin{aligned} 4P_1 + 2c_1 + c_2 &= 4P_2 \\ P_1 + c_1 + c_2 &= P_3 \end{aligned} \qquad [3]$$

Solve the first equation in [3] for c_2 and substitute in the second equation,

$$c_2 = 4P_2 - 2c_1 - 4P_1 \quad \Rightarrow \quad P_1 + c_1 + (4P_2 - 2c_1 - 4P_1) = P_3 \qquad [4]$$

Solving [4] for c_1 and then substituting back into [3] yields

$$\begin{aligned} c_0 &= P_1 \\ c_1 &= 4P_2 - 3P_1 - P_3 \qquad [5] \\ c_2 &= 2(P_1 - 2P_2 + P_3) \end{aligned}$$

Parametric Quadratic Class and Demo

Equations [5] can be easily incorporated into a parametric quadratic class, which is included as part of the code distribution for this TechNote. The *PQuad* class accepts three interpolation points. Equations [5] are used to compute coefficients of the quadratic polynomial in t, passing through each point.

The following screen shot illustrates a choice of three points that form a close fit to the curved arrow.

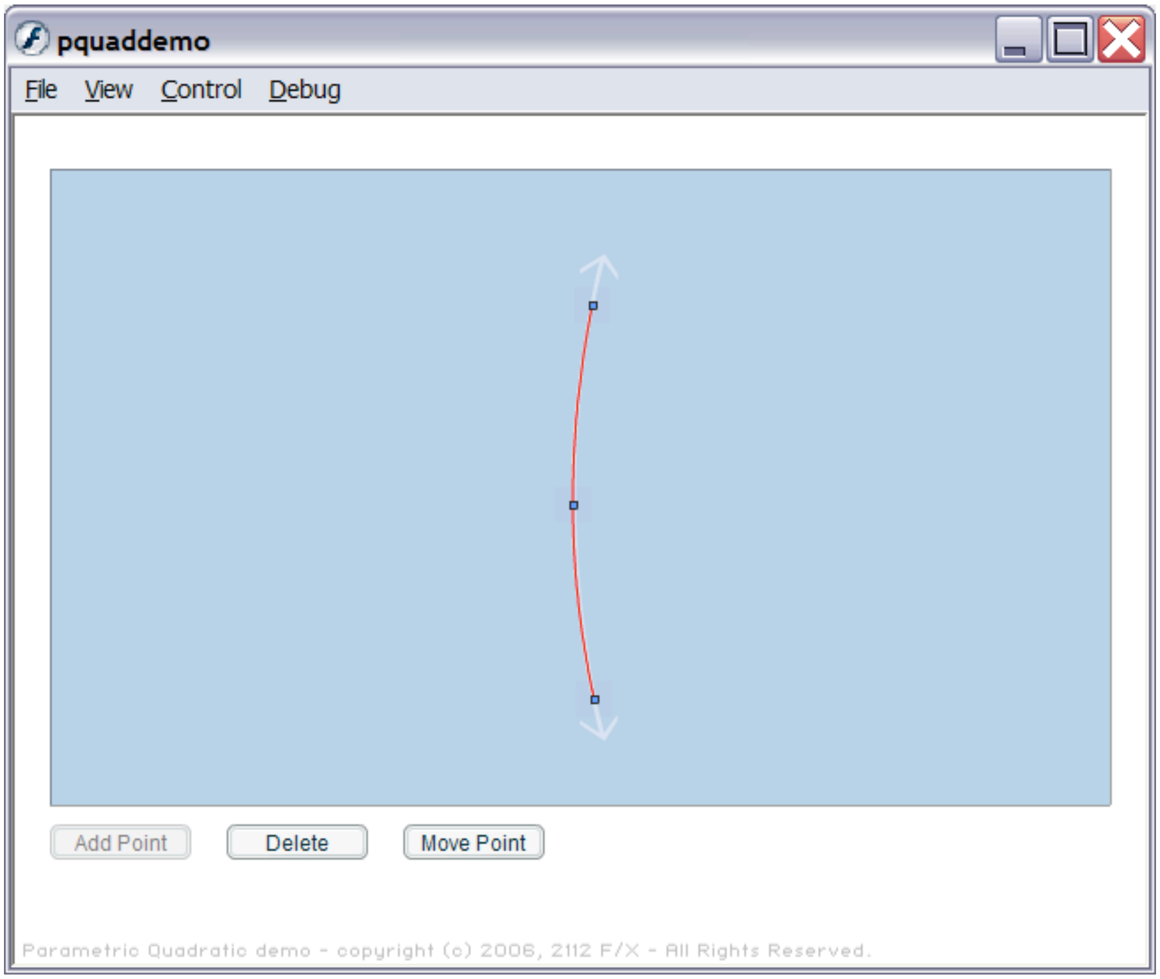

pquaddemo
File
View
Control
Debug
Add Point
Delete
Move Point
Parametric Quadratic demo - copyright (c) 2006, 2112 F/X - All Rights Reserved.

Diagram 3: Sample usage of the PQuad class.

For many curved navigational graphics, *PQuad* should be adequate to produce an approximation to a curved segment. Experiment with the demo, especially moving the middle point further away from the two endpoints. What you should notice is that this approach provides no *shape control* over the curve. Although it is possible to modify the algorithm to specify an arbitrary parameter value for the middle point, this tends to skew the curve towards one of the two endpoints. It is very difficult to create a user interface to fit a particular curved segment using such an approach.

A quadratic curve only has three degrees of freedom. By specifying that the quadratic curve pass through three points, no additional shape control is available. The fit is good for curved segments like the above example, where the three points are nearly collinear. For more general curved segments, more shape control may be required.

What would be highly desirable is a curve that interpolates three points and provides tangent control at each of the three points.

Piecewise Hermite Curve

A cubic Hermite curve interpolates two control points, with tangent controls at each point. The tangents provide shape control for the curve. Suppose two Hermite curves were constructed, the first of which interpolates (x_1, y_1) and (x_2, y_2). The second curve interpolates (x_2, y_2) and (x_3, y_3). Tangent controls at (x_1, y_1) and (x_3, y_3) are automatically provided with C^1 continuity at the join point, (x_2, y_2). This construction produces a piecewise cubic curve that interpolates all three points, with tangent controls at each point. The tangent handle at the join point is symmetric, i.e. the magnitude of the in-tangent equals the magnitude of the out-tangent. One endpoint of the tangent is a reflection of the other end.

The TechNote on Hermite curves introduced the basic equations for describing a single curve segment. Collecting terms to allow coordinates to be computed via nested multiplication was left as an exercise. The necessary equations are listed below as the coefficients are stored (segment by segment) in instances of the *Cubic* class.

$$H(t) = [2(P_0 - P_1) + (R_0 + R_1)]t^3 + [3(P_1 - P_0) - 2R_0 - R_1]t^2 + R_0 t + P_0 \qquad [6]$$

Application of this formula is dependent on selecting the correct tangents for each segment. For purposes of this application, there are two curves with three control points. Each curve shares a common control point, with C^1continuity. The first segment has an out-tangent at the first control point and an in-tangent at the midpoint. The second segment has an out-tangent at the midpoint and an in-tangent at the third control point.

If this process was extended to encompass n knots, the complete curve would have $n-1$ segments and $2((n-2)+1)$ tangent vectors. It is simplest to store the tangents in linear order for each segment, i.e. out-tangent, in-tangent, out-tangent, in-tangent, etc. When computing the cubic coefficients for each segment, some additional index computation is required to select the correct tangent. This is seen in the *__computeCoef()* method of the *Hermite* class.

Two programs are required in order to use the piecewise Hermite curve in a scroller. One program allows an artist to 'design' the appropriate piecewise curve. When finished, information needed to construct the curve is written to a file. The actual application reads the information, constructs the curve dynamically, and then uses the curve to control the motion of a scroll indicator.

The 'curve design' program should at least allow a background graphic to be attached from the library. A really nice implementation would allow a background graphic to be externally loaded. The background graphic contains the curved area of an interface for which the piecewise Hermite curve must 'match.' The user should be allowed to enter three control points with tangents initially set by some prescribed algorithm. After the curve is plotted, the user adjusts control point placement and tangents to optimize the fit.

The ideal production implementation employs a projector so that coordinate information to reconstruct control points and tangents could be conveniently written to a file.

Tangent Handles

In many applications, it is necessary to auto-generate tangent handles or control cages. Sometimes, the specification of tangents is well defined. In other cases, the choice is more arbitrary. The goal is to provide a suitable initial curve that is to be further modified to suit the artist's goals. This section discusses the topic of auto-tangent generation in the case of no *a priori* specification. There is no 'best' approach, so you are encouraged to experiment with and modify the following algorithm.

Tangent handles at the outermost control points are directed from control point to the nearest tangent handle at the midpoint. Their length is half the distance from outer control point to tangent handle. The in- and out-tangents at the join point are parallel to the chord from P_0 to P_2. To avoid 'kinks', both tangent lengths are set to half the smallest distance from the midpoint to one of the outer control points. The general case is illustrated in the following diagram.

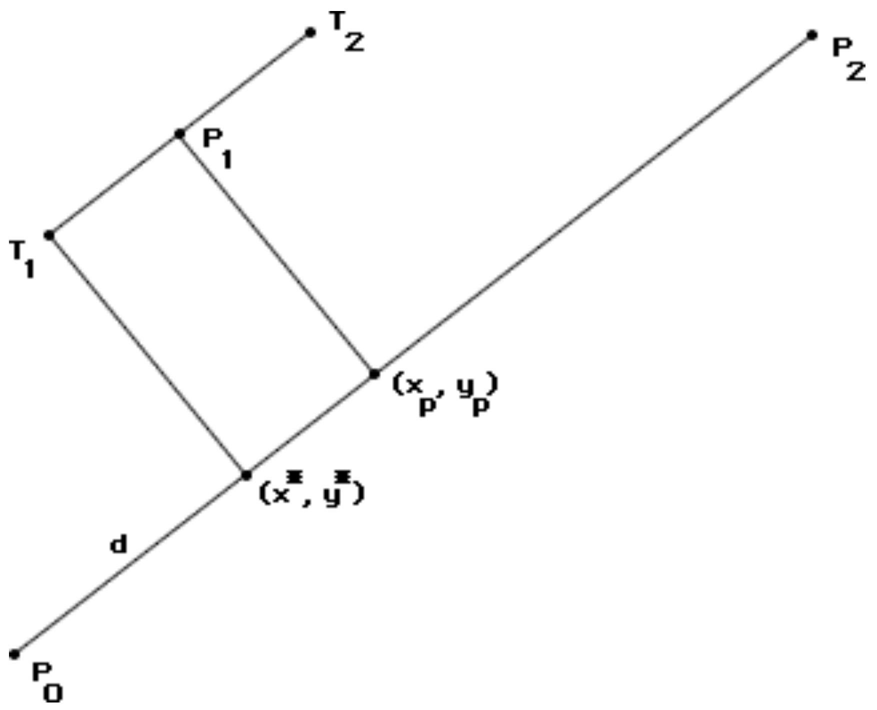


Diagram 4: Computation of in- and out-tangent handle coordinates.

Essentially, the problem is to compute the coordinates of tangent handles T_1 and T_2. Once coordinates of those tangent handles are computed, the tangents at the outer control points are trivial.

If T_1 is projected onto the line segment P_0P_2, then the projection point, (x^*, y^*) is at a prescribed distance, d, from P_0.

The equation of the line passing through P_0 and P_2 in parametric form is $P_0 + t(P_2 - P_0)$, where t varies from 0 to 1. Instead of distance, it is easier to think of assigning a parameter value to compute (x^*, y^*). A value of t = 0.5 is a convenient choice; however, some additional information is required.

Let P describe the projection point with coordinates (x_p, y_p). Choose the segment P_0P or PP_2 having minimum distance. If P^* is the point with coordinates (x^*, y^*), then set

$P^* = P_0 + 0.5(P - P_0)$ if P_0P is the minimum-distance segment or $P^* = P + 0.5(P_2 - P)$ otherwise.

Once P^* is chosen, T_1 can be computed and once T_1 is computed, T_2 is simply a reflection about P_1. This approach requires solving two problems.

1) Compute the projection point, P, to determine the segment of minimum distance. Use this information to compute (x_p, y_p) and P^*.

2) Use P^* to compute T_1 or T_2 (depending on the minimum segment). Assign coordinates of the other tangent handle as a reflection about P_1.

<u>Problem 1</u>

This is a classic problem in computational geometry, requiring the projection of a point, P_1 onto a line passing through P_0 and P_2. The parametric equation of the line segment is $P_0 + t(P_2 - P_0)$. Since the line segment from P_1 to P is perpendicular to P_0P_2

$$[P_0 + t(P_2 - P_0)] \bullet [P_1 - P] = 0 \qquad [7]$$

This TechNote is already too long (and this is a classic problem whose solution is well documented both online and in the literature), so we will skip the algebra and get right to the solution (see [2] for example),

$$t = \frac{(x_1 - x_0)(x_2 - x_0) + (y_1 - y_0)(y_2 - y_0)}{\|P_2 - P_0\|^2}$$

yielding the coordinates

$$\begin{aligned} x_p &= x_0 + t(x_2 - x_0) \\ y_p &= y_0 + t(y_2 - y_0) \end{aligned} \qquad [8]$$

Given these coordinates, determine the minimum-distance segment and the coordinates (x^*, y^*).

Problem 2

Suppose that P_0P is the minimum-distance segment, requiring the computation of T_1. This point is at the intersection of the line segments P^*T_1 and P_1T_1. Suppose the slope of P_0P_2 is m. The following two equations apply to the intersection point (recall that slopes of perpendicular lines are negative reciprocals of one another)

$$\begin{aligned} P_{1y} &= T_{1y} + m(P_{1x} - T_{1x}) \\ T_{1y} &= y^* - (T_{1x} - x^*)/m \end{aligned} \qquad [9]$$

from which

$$\begin{aligned} &my^* - T_{1x} + x^* = mP_{1y} - m^2(P_{1x} - T_{1x}) \\ &\Rightarrow T_{1x} = \frac{x^* + m(y^* - P_{1y} + mP_{1x})}{m^2 + 1} \end{aligned} \qquad [10]$$

The value of T_{1x} from [10] can be substituted into the second equation in [9] to solve for T_{1y}. The coordinates for T_{2x} and T_{2y} are obtained directly via reflection about P_1. Equation [10] may be used for T_{2x} and T_{2y} in the event that PP_2 is the minimum-distance segment. The only difference is the algorithm for (x^*, y^*). This is illustrated in the demonstration code.

Details

The process would not be fun if there were no subtle details ☺

1) In the single-segment Hermite curve demo, a knot was interactively assigned and then its tangent handle was immediately set. Since this application requires a composite curve with two segments, three control points are assigned and then four tangent handles are set. The middle tangents are assigned first, followed by the outer tangent handles. This order is dictated by the fact that the tangent handles are the join point are required in order to set the tangents at the outer control points.

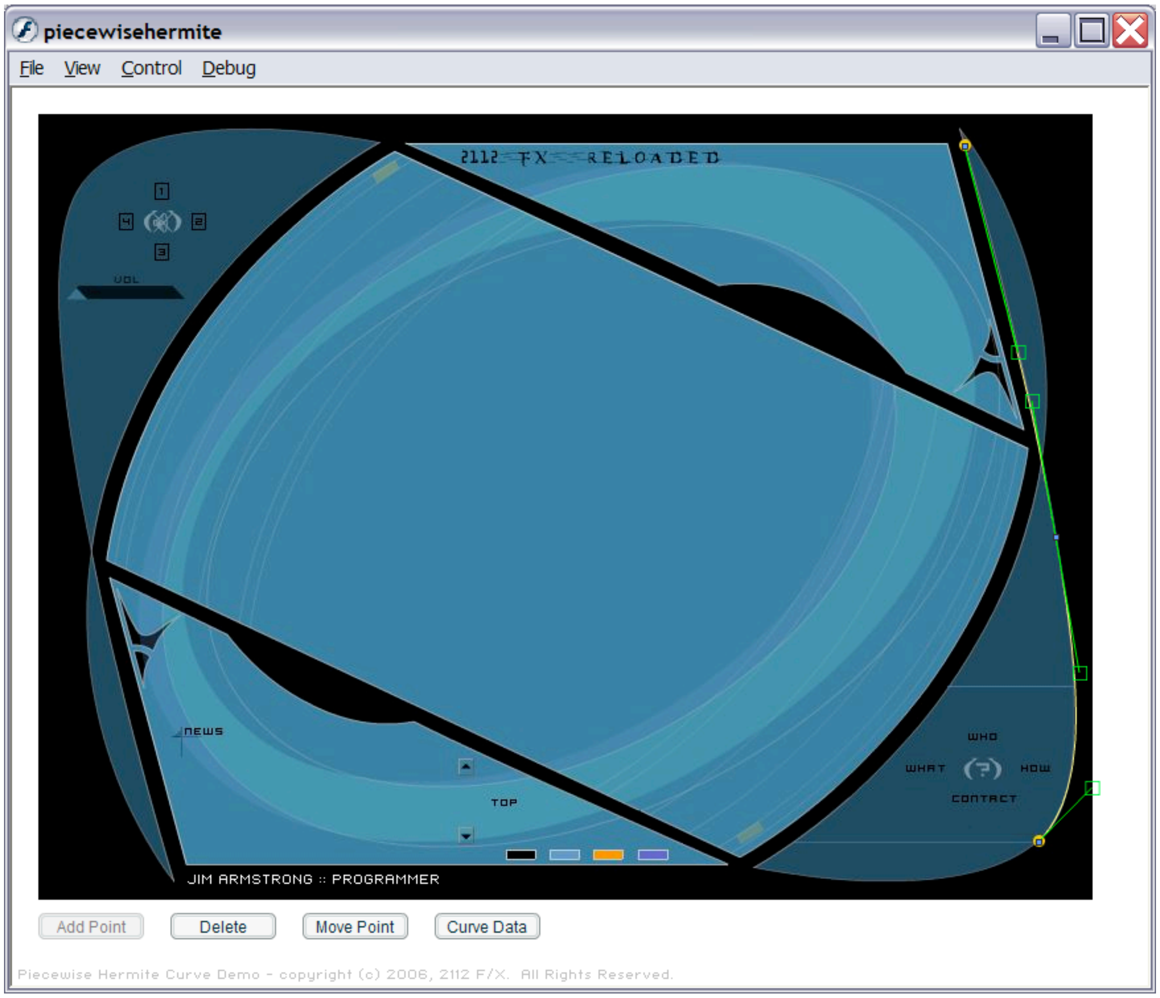
2) The tangent values that feed into the Hermite coefficient computations are set based on whether the tangent index is even or odd. This is because after the first control point, two tangents key off a common control point (at each join).

3) When control points are moved, the tangent handles are offset by the same amounts as the control point. Two tangent handles are moved at each join point. Internally to the class, the deltas between tangent handles and control points remain unchanged by translation. The driver is responsible for updating the handle coordinates in the display.

4) The Hermite curve uses a uniform parameterization just like the Catmull-Rom spline. Support for arc-length parameterization will be added in the future.

Curve Creation Demonstration Program

For illustrative purposes, I used a very old site interface with some organic elements. The two orange ‘dots’ in the diagram below determine the position of the initial and terminal scrolling points along the curve.



The line color and tangent handle colors were made variable to enable better contrast against a wide variety of background colors. Placement of control points and adjusted tangent handles are illustrated.

There is no ability to zoom and pan the interface (other than that provided by Flash itself), so making very small-scale changes is harder than it might be in a production code. Note that any zoom/pan capability would be limited only to the background graphic and would be masked inside some prescribed display area.

Once the curve is 'fit' to the interface graphic, the 'Curve Data' button displays a colon-delimited string of control point and tangent information. A production implementation in a projector would write this data to a file. The text field displayed in the interface is selectable, so copy and paste the data to a file named *curve.dat*. This file is read by the actual scrolling program using *LoadVars*.

Scrolling Example

The goal is to create a curved, vertical scroller. Users are accustomed to strictly horizontal or vertical scrollbars. In the case of a vertical scroller, when the scroll indicator is at the 'top' of the scrollable interface, it is expected that the content is also at its uppermost display limit. Conversely, when the scroll indicator is at the 'bottom' of the interface, the user expects content to be at its lowest display limit.

The above observation indicates a linear algorithm, even though scrolling is displayed along a curve. First, compute the range from P_{0_y} to P_{2_y}. The mouse y-coordinate is polled and mapped to the interval [0,1] based on its position in between the two y-coordinate limits (values outside the range are clipped at zero or one). The value inside [0,1] is used as a parameter to query the x- and y-coordinates of the curve. The scroll indicator is placed at those coordinates.

As long as the scroll indicator is pressed and held, the viewer may move the mouse up and down anywhere in the interface. The scroll indicator will appear to be 'dragged' along the curve. The indicator never exceeds the vertical boundaries established by P_{0_y} and P_{2_y}.
While this approach satisfies one user-interface consideration, there are some mathematical issues.

1) The curve is parameterized (uniformly) on t, not arc-length. Equal increments in t do not correspond to equal distances along the curve. Depending on curvature, this results in non-uniform velocity of the scroll indicator along the curve, even for constant velocity of vertical mouse movement.

2) In general, the computed parameter value will **not** produce the same y-coordinate along the curve as the mouse y-coordinate. A more exact algorithm returns the proper x-coordinate, given a y-coordinate. In the case of either time- or arc-length parameterizations, this approach requires the ability to interpolate parameter as a function of coordinate. The cubic spline is useful for such a purpose since the variance of arc length with natural parameter is smooth. The fraction of current parameter value at a given y-coordinate is used to set content position.

3) The scroll indicator responds to any horizontal mouse movement in the interface. It may be desirable to create a bounding box for the curve and only scroll when the mouse is inside the bounding box. This requires computing the maximum horizontal and vertical extents of the curve. For a vertical scroller, the vertical limits are already known. Finding the exact horizontal limits requires computing the maximum horizontal extent of the curve. This approach requires some extra calculus to compute the parameter value where the curve is maximized (or minimized) in a particular dimension. For purposes of this TechNote, the example program allows the mouse to move horizontally anywhere in the interface.

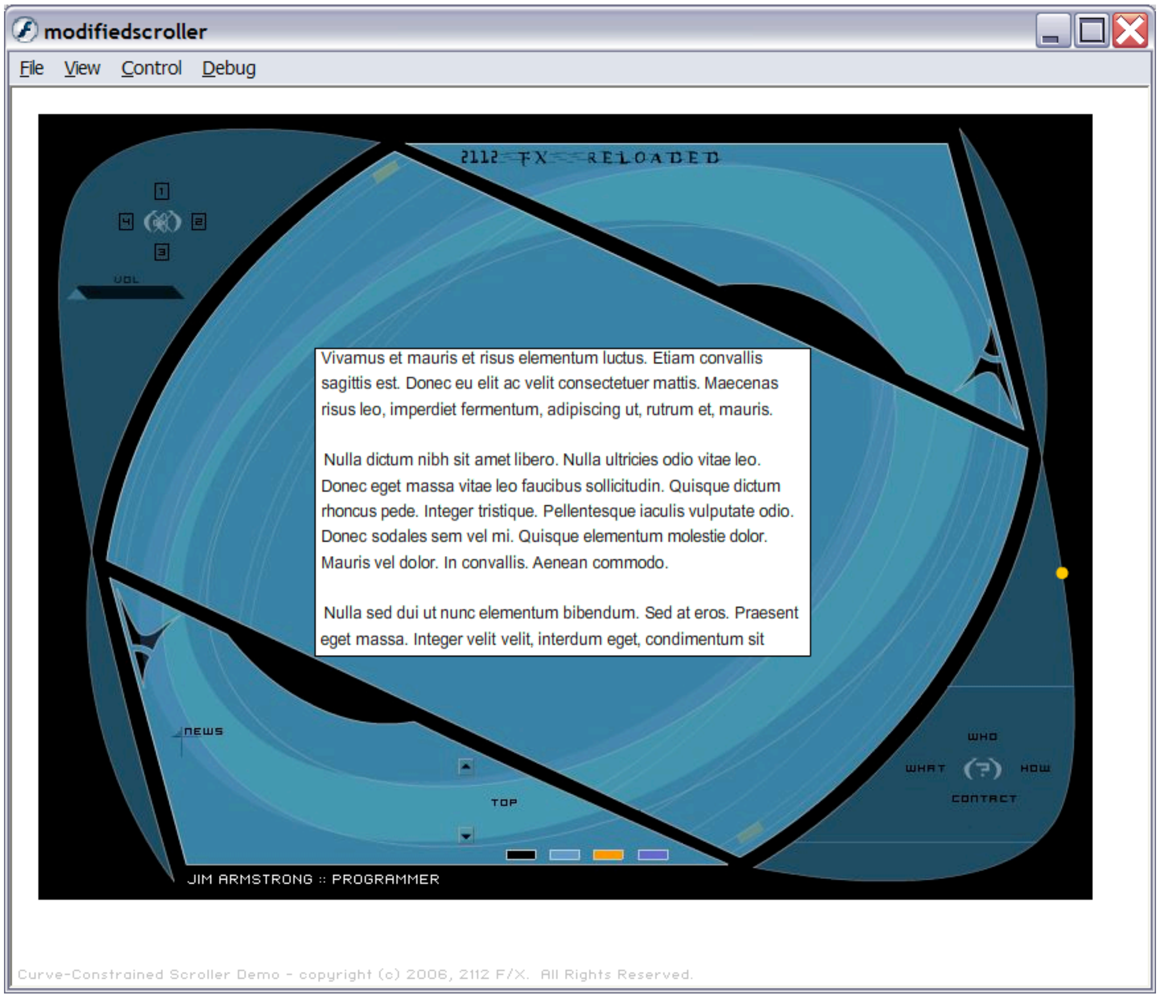
Two scrolling problems need to be addressed in a full production implementation. A typical user interface has scroll arrows that allow the interface to be scrolled small amounts when pressed. In addition to moving content by the appropriate amount, a parameter value needs to be computed to move the scroll indicator along the curve. The other problem is to adjust content position based on the current parameter value. Since the parameter from zero to one based on y-coordinate, the parameter can be used to control either the *maxscroll* property of a *TextField* or the position of a MovieClip, relative to its maximum scrollable limit.

The most accurate implementation of a curve-constrained vertical scroller requires arc-length parameterization and the ability to return x-coordinate along the curve based on current mouse y-coordinate.

Modified Scroller with Parameter Interpolation

Instead of converting the mouse y-coordinate to a parameter based on fraction of range, a somewhat better approach is to create a table of parameter values corresponding to y-coordinates on the curve. As part of initializing the scroller, create a table of values that can be used to 'lookup' a t-value given any y-coordinate. The cubic spline class is ideal for such a task. Once the parameter value is returned from the cubic spline, use the parameter both to scroll the *TextField* and to determine the exact coordinates on the curve to set the scroll indicator. With a reasonable interpolation, the y-coordinate on the curve will be very close the mouse y-coordinate, providing a reasonable approximation to a curved scroller.

A screenshot is provided below.


modifiedscroller
File View Control Debug
2112 FX RELOADED
VOL
NEWS
TOP
WHO
WHAT
HOW
CONTACT
JIM ARMSTRONG :: PROGRAMMER
Curve-Constrained Scroller Demo - copyright (c) 2006, 2112 F/X. All Rights Reserved.

Vivamus et mauris et risus elementum luctus. Etiam convallis sagittis est. Donec eu elit ac velit consectetuer mattis. Maecenas risus leo, imperdiet fermentum, adipiscing ut, rutrum et, mauris.

Nulla dictum nibh sit amet libero. Nulla ultricies odio vitae leo. Donec eget massa vitae leo faucibus sollicitudin. Quisque dictum rhoncus pede. Integer tristique. Pellentesque iaculis vulputate odio. Donec sodales sem vel mi. Quisque elementum molestie dolor. Mauris vel dolor. In convallis. Aenean commodo.

Nulla sed dui ut nunc elementum bibendum. Sed at eros. Praesent eget massa. Integer velit velit, interdum eget, condimentum sit

Summary

The best implementation of curve-constrained scrolling requires both arc-length parameterization and the ability to return one coordinate along the curve given the other. The latter problem was handled with cubic spline interpolation in this paper. With an arc-length parameterization, the fraction of total arc-length at a specific x- or y-coordinate is used to determine scroll position. This computation results in the most accurate curve-constrained scroller.

This TechNote illustrated the creation of two types of curves to ‘fit’ sections of organic shapes in a user interface, a parametric quadratic curve and a piecewise cubic Hermite curve. This problem also illustrated a practical application of the cubic spline class introduced in the first TechNote in this series.

A general-purpose curve design program is critical to producing a curve that exactly simulates the required motion. The program supplied in this TechNote is a start towards such an application, although there is substantial room for improvement. Even if you never require a curve-constrained scroller, the piecewise Hermite curve and auto-tangent algorithm could be useful in a future application.

I hope you find this a useful reference and starting point for applications requiring any type of motion along a curve inside a user interface.

References

[1] Grden, J., "localToGlobal and globalToLocal coversion made easy" -- www.acmewebworks.com/default.asp?ID=100&mc=Content

[2] Foley, J., Van Dam, A., Feiner, S., and Hughes, J., "Computer Graphics: Principles and Practice, 2nd. Ed., Addison Wesley, 1997.