Special Problem

Heart Surgery Simulation: Constant length Curves

Piyush Soni under the guidance of Prof. Jarek Rossignac
Georgia Institute of Technology, Graphics, Visualization and Usability Center
Atlanta, GA 30332
piyush_soni@gatech.edu, jarek@cc.gatech.edu

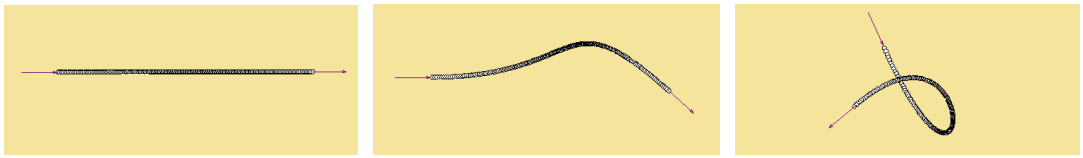


Figure 1: Constant length bending of a curve

Abstract

This project is an effort to simulate in real time the deformations of a curve which is constant in length. It also touches upon a novel approach of smoothing a constant length polyloop using the approach of minimizing the 'turn angles' which is explained in this report. This should be helpful in Heart Surgery and other applications where bending and joining of any tubular structures is required. In heart surgery, arteries which separate to arterioles and capillaries can be all simulated by using it.

Keywords: Constant length, Polyloop, bending, smoothing, turn angle, bi-arc, Bezier curves

1. Introduction

Various approaches were tried for the same, some of which were unsuccessful too. The basic idea was to control the curve by its end points and tangent vectors at those, such that changing the above enables us to produce any desired and possible shape of the constant length curve easily. The most feasible curve which could be inserted in the control points and tangent vectors needed to be studied, so that the length at any point of time can be calculated and manipulated in real time. The aim was to find out and setup a direct relation between the tangent vectors and the length of the curve. Though, it was found out that none of the approaches tried could establish such a simple relation. So, iteration had to be used finally. The same approaches are discussed in this report. In this report we assume that the reader has the knowledge about Bezier curves.

2. Circular Arcs

The first approach we thought of was that of inserting two circular arcs between the tangent vectors, such that the two meet at the same slope. As we can see from the figure below, the length of tangents from point Q to P and M are both 'a' (Two tangents from a common point to a circle) and similarly for 'b'.

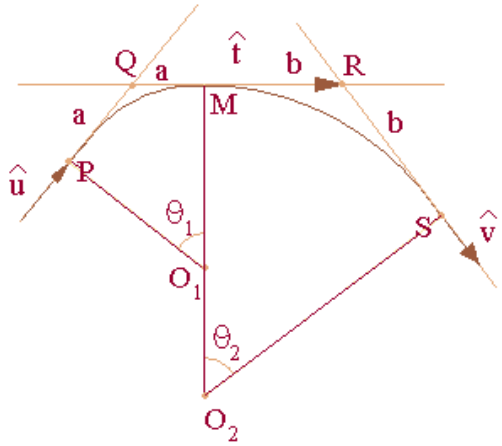


Figure 2: Insertion of two circular arcs in control points and tangents.

Now, there are three unknown variables in the figure above, a and b and t, and the equation which connects the variables are:

$$\vec{P} + a\hat{u} + (a+b)\hat{t} + b\hat{v} = \vec{S}$$

Where P and S are control points of the curve and u and v are unit vectors in the tangent directions. T is the vector which connects the other two tangents.

Here, we note that this equation is not sufficient to solve for both a and b.

If we fix a = b, things become easy and we can solve for them, but not otherwise.

The advantage of this approach was that a circle is a more intuitive a shape than others and if we know the arc angle, we can determine the length of the arc by multiplying the angle with the radius.
The total length of the bi-arc is given by:
$L = L_1 + L_2$
$= r_1.\theta_1 + r_2.\theta_2$
Now, to determine θ and r we need the values of a and b, and thus we come to know that the total equations are not enough to determine the values unless we do some tweak like fixing a = b above.

There are a few disadvantages too in inserting circular arcs, first of all the calculations of θ and r above involve complex trigonometric calculations, which would hamper the performance of the application and secondly, a two arcs look too circular to be real bending. To demonstrate this, we see the following figure:

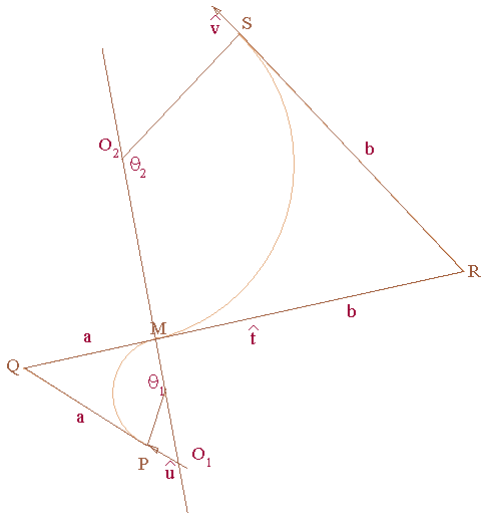


Figure 3: Circular arcs. Note that this bending is not very much like that of a real life tubular structure.

3. Bezier Curves

This was by far the most successful method implemented, which produced a simulation which is both realistic, easy to implement and which gives a pretty satisfactory real time response to the user deformations.

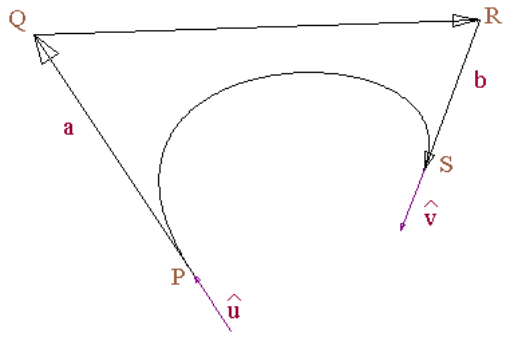


Figure 4: A Bezier curve given by four control points

In this, we start by fixing particular values of P, Q, R and S and let the user change the control points, **P** & **S**; and the control tangents **u** and **v**.

Again, calculating the length of the Bezier curve in real time was the challenge. Some papers had content about calculating its length, but they all involved complex calculations which are hard to carry out on the move.
For example, one example of calculating the length of Bezier curve is to do it by using its parametric form which is:

$$x = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \quad .....(1)$$

$$y = b_0 + b_1 t + b_2 t^2 + b_3 t^3 \quad .....(2)$$

This form of curve can be integrated by using Elliptic Integrals. (James FitzSimons, 2004)

But solving of Elliptic Integrals is in itself a tedious and calculation extensive part. According to the method given by James FitzSimons, it involves finding complex roots of four degree equations, then looking into big table of values for those. To prevent all those complexities – when they can not serve the purpose of improving the efficiency, I thought of implementing the simplest way of doing this, which was through numeric integrals in the given period. The same is given here.

Equations (1) and (2) can also be represented in the parametric form, as:

$$x = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

$$y = b_0 + b_1 y + b_2 y^2 + b_3 y^3$$

$$dx = (a_1 + 2.a_2 t + 3.a_3 t^2)dt$$

$$dy = (b_1 + 2.b_2 t + 3.b_3 t^2)dt$$

Now, length of a curve can be given by:

$$L = \int \sqrt{dx^2 + dy^2}$$

$$= \int \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\, dt$$

$$= \int_0^1 \sqrt{(a_1 + 2.a_2 t + 3.a_3 t^2)^2 + (b_1 + 2.b_2 t + 3.b_3 t^2)^2}\, dt$$

Hence, this can be represented in parametric form written above and easily integrated from t = 0 to 1 with a properly chosen close to zero value of 'dt', according to the precision required. Each time in the loop, the value of $(dx/dt)_t$ and $(dy/dt)_t$ are calculated, just squared and taken square root of to take the product with the small value, 'dt'. This is summed up for t = 0 to 1. To increase the speed of Bezier length calculation, we pre compute all those values which can be, and each time in the loop we just calculate how much dx and dy should be incremented. This converts the square calculations to just simple addition and linear multiplication. According to the testing done, it has proved to improve the performance of the application by at least 15%.

Now, when we have the length of Bezier function ready, we can calculate the initial length of the Bezier, and the instantaneous changed length by the user. Hence, we know whether the length has to be increased or decreased in order to bring it back to the original value.

When we have to change the length to a particular value, there are more than one ways of doing that due to the inherent indeterminism involved of 'a' and 'b' values. Here, we just choose to set one pattern to solve this problem. It follows:
Let a factor, f denote the measure of how much we want to increase or decrease the length. Then,
f = (Original Length – Current Length)/ K
Where K = a suitably chosen constant. Here, the value of K is also changed based on the difference between the two lengths to converge it faster. For example, when the length is decreasing from 150 to 100, it will take bigger steps to decrease than what it would take from 15 to 10.

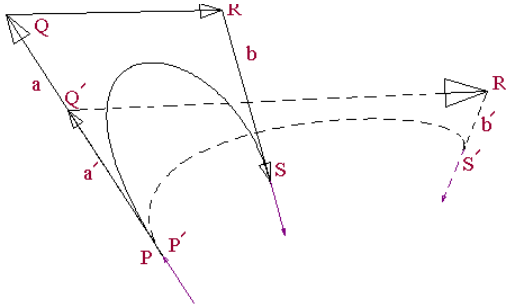


Figure 5: Conservation of length when end point and tangent vectors are moved

Now, if we want to decrease the length since the user has tried to increase it by pulling one end apart, we do that by decreasing the distance between points P and Q to PQ' and S and R to S'R' such that both of these edges decrease by the same factor f explained above. Though there are many other solutions and ways of doing it, the reason of choosing this is the smooth and intuitive results it produces. Similarly, we do for increasing the length of the curve.
When the control points are dragged, the tangent directions remain the same. They are changed only when the user deliberately holds the two tangent vectors (we will call them Handles)

***Observation**: By increasing the edge lengths, the length of the Bezier increases and vice versa. A proof/disproof of this is still worked upon.*

After matching length again, the new Bezier curve is drawn by joining sampled points by line segments.

Peformance
After some code improvements, the code works pretty good, and the response is real time.

4. Polyloop Smoothing

Another approach which was thought of was to divide the whole curve in small segments of straight lines, let the user deform it using one of the ends, and then move all the other segments with the end points and finally smoothen them which might produce good results.

Firstly, the smoothing approach has been explained: Let the curve be a polyloop of a certain number of vertices, all of which are connected by edges. More the number of vertices in the polyloop for a given length, smoother the curve will look like.

The approach used for smoothing here involves minimizing the angle each edge turns when it goes to the other.

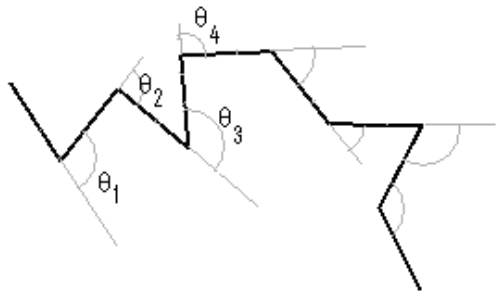


Figure 6: Example of an open polyloop with random turn angles

Here, we take the tuples of five edges from the polyloop and find out a configuration in which the set of those edges have the minimum "max turn", where max turn is the maximum value of the turn the edges take, shown in the above example as θ_1, θ_2, θ_3 and θ_4 lead to a global minimum for the curve. The argument is that if the turn angles have to be minimum, at least two of them should be the same. Because otherwise, if we try to change one of the angles, one will have a tendency to become more one of the turns will have the tendency to go on the acute side and the other one the obtuse side – and they can be in equilibrium only when they are equal.

The sub steps for doing this include:

1). *Mid edge turn minimization*

In this, we try to make the inner two turn angles equal, keeping the outer two edges as they are and find out the configuration of the others, and also whether this is the minimum 'max turn' configuration.

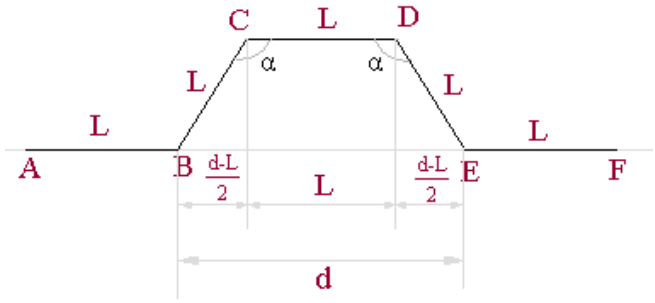


Figure 7: Making inner angles equal in the five edge tuple

As we can see from the figure above, let d be the distance between points B and E which is known to us. We can easily see that in this configuration CD is parallel to BE, as all edges are of length L and Angle BCD = Angle CDE.

It's not hard to see that C can be given by:

$$\vec{C} = \vec{B} + \left(\frac{d-L}{2}\right)B\hat{E} + \vec{H}$$

Where vector(**BE**) = Unit vector in the direction of BE and **H** is a vector in the direction perpendicular to it with a magnitude given by:

$$|\vec{H}| = \sqrt{L^2 - \left(\frac{d-L}{2}\right)^2}$$

D can be calculated simply by:

$$\vec{D} = \vec{C} + (L)B\hat{E}$$

Now, the maximum turn in this configuration can be calculated by just noting the maximum angles these edge vectors have to take. It comes handy to make a maxTurn() function.

One important thing to note in this is that the minimum 'max turn' configuration can be it's just opposite also, i.e. C & D can be the mirror image along BE of what has been shown in the figure 7. The program takes care of both of them.

2). Exterior edge turn minimization

Here we try to make the exterior two angles equal. The purpose is same, to find whether this is the minimum 'max turn' configuration.

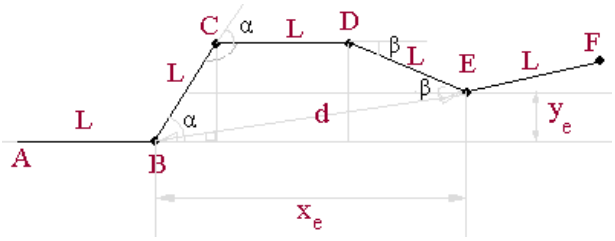


Figure 8: Making exterior angles equal. Note that this time CD is parallel to AB, not to BE.

This analysis is slightly longer than the previous one.
From the figure, it can be written that:

$$L(1+\cos\alpha)+L\cos\beta = x_e$$
$$L\sin\alpha - L\sin\beta = y_e$$

Manipulating, squaring and adding,

$$[L(1+\cos\alpha)-x_e]^2+[L\sin\alpha - y_e]^2 = L^2$$

Simplifying and manipulating again,

$$L^2 + {x_e}^2 + {y_e}^2 - 2x_eL$$
$$= 2L[(x_e - L)\cos\alpha + y_e \sin\alpha]$$

Let's say

$$\frac{L^2 + {x_e}^2 + {y_e}^2 - 2x_eL}{2L} = \text{K} \quad .....(4.1)$$

Then the above equation becomes

$$\text{K} = [(x_e - L)\cos\alpha + y_e \sin\alpha] \quad .....(4.2)$$

Where Equation (4.2) is a standard equation from which we can calculate the value of α by representing it in the form of
K/C = sin A cos B + cos B sin A

Dividing both sides by $\sqrt{(x_e - L)^2 + {y_e}^2}$ we get

$$\frac{K}{\sqrt{(x_e - L)^2 + {y_e}^2}} =$$
$$\frac{(x_e - L)}{\sqrt{(x_e - L)^2 + {y_e}^2}}\cos\alpha + \frac{y_e}{\sqrt{(x_e - L)^2 + {y_e}^2}}\sin\alpha$$

From (4.1) and the above equation,

$$\sqrt{\frac{K}{2L}} = \sin\theta\cos\alpha + \cos\theta\sin\alpha$$

Or, $\sin(\theta + \alpha) = \sqrt{\frac{K}{2L}}$

Or, $\alpha = \sin^{-1}\sqrt{\frac{K}{2L}} - \theta$

Where $\theta = \sin^{-1}\frac{(x_e - L)}{\sqrt{(x_e - L)^2 + {y_e}^2}}$

$$= \sin^{-1}\left(\frac{x_e - L}{\sqrt{2LK}}\right) + 2\pi n$$

Where n = 0, 1, 2...

Hence, using this value of α we can calculate the point C and D as we know the values of x_e and y_e which are the components of distance BE in the direction of AB and perpendicular to the direction of AB.

Note:
This was shown from one side, in which the turn of AB and BC are taken into consideration. Similar to this, the program performs calculations from the other side also, i.e. the turns for EF and DE.

Also, for both since it involves taking sin inverse, more than one values of angles are possible. All have been taken into consideration to determine the configuration which minimizes the max turn.

Shape Manipulation in Constant length polyloop

Having done till now, the user should be able to modify the curve, still maintaining the constant length of the curve, as in the case of Bezier curves. For this, I have implemented two techniques which enable the user to manipulate the curve.

Localized Manipulation

In this type of manipulation, the user generally wants to change the shape minimally keeping the length constant.
For this, the point closest to the mouse is dragged in the side of the polyloop which contains the mouse click. For this, the perpendicular bisector of the vertices which are two indices far from the nearest point is taken, and then they are straightened up to meet on this bisector. It is to be noted that the mouse click is just use to determine the closest point and the side on which the dragging has to be done.

It does not decide the actual direction of the dragging, which is just the perpendicular bi-sector of the points two far away from the closest one.

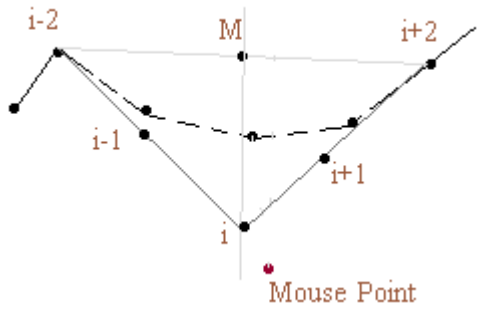


Figure 9: The old points are shown in dashed line, which are replaced by the new ones in the side of mouse click.

Global Manipulation

In this type of constant length curve manipulation, the closest point is actually dragged to match to the mouse point, and all the subsequent points on the curve follow the dragged point.

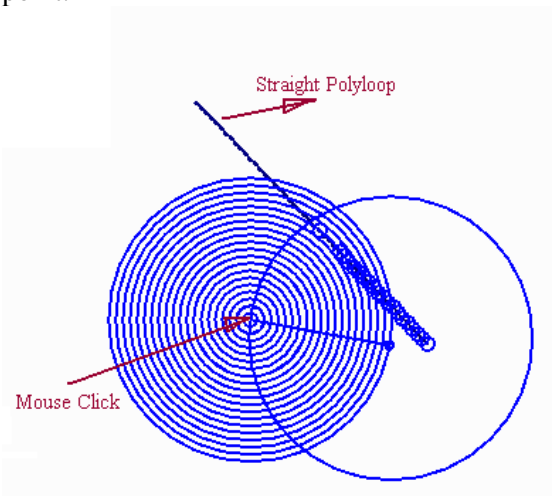


Figure 10: Circle Intersection testing to correctly pull all the possible points towards the mouse click.

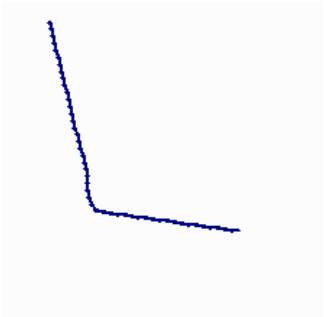


Figure 11: Result of the above operation – changed curve maintaining its constant length

In the above two figures, we see that after the closest point to mouse moves towards it, we just test whether after pulling each edge more towards the mouse point the curve can be merged into the older one. If not, the whole curve is moved along with the mouse pointer.

5. Conclusion

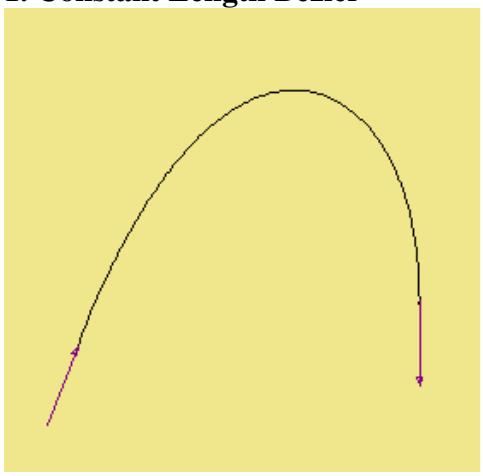
From all of the approaches we have seen till now, Bezier Curve was found to be best simulating a real bending tube or artery which can be converted to 3D to utilize in Heat Surgery. The code produced would be submitted to the Project guide, Prof. Jarek Rossignac for analysis and future development of the same.

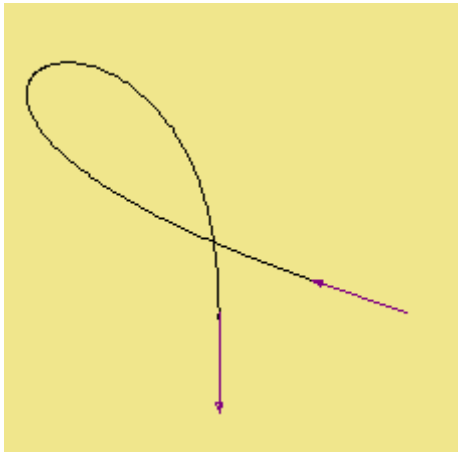
6. Future Work

A lot of work still needs to be done in this area. The segment approach of producing constant length open polyloops which work in real time don't seem to exist today. Other than that, converting the Bezier Curve deformations into 3D is also a development which needs to be implemented. The 3D deformations would include one more feature which doesn't exist in 2D, and that is torsional strain.

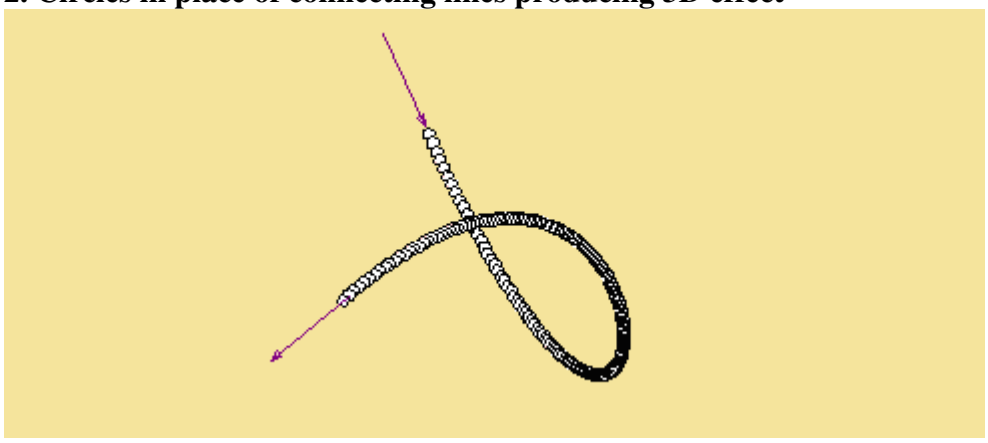
7. Results

1. Constant Length Bezier

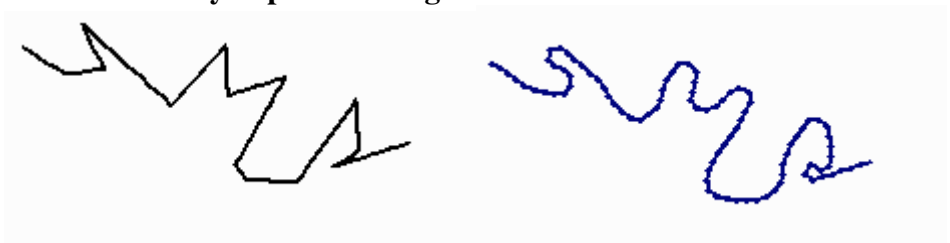




2. Circles in place of connecting lines producing 3D effect



3. Result of Polyloop Smoothing



8. References

1). Length of a Bezier Curve by James FitzSimons, June 24, 2004

2). Solid and Physical Modeling, Jarek Rossignac

3). Adaptive subdivision and the length and energy of Bézier curves - Jens Gravesen(Department of Mathematics, Technical University of Denmark)

4). Jarek Rossignac's class lectures

5). Cubic Curves - Steve Rotenberg, UCSD, Fall 2006