Yarn Animation

An Experiment with Curves and Physics

April M. Grow, Julie Rej
University of California: Santa Cruz
1156 High Street
Santa Cruz, California
agrow@soe.ucsc.edu, jrej@ucsc.edu

ABSTRACT

As computers and technology become more powerful and prevalent, we find them expanding into all corners of the real, that is to say non-digital, world. Many objects have been simulated, to varying degrees of success, but usually toward a specific purpose, a means to an end. This project abstracted away from creating yarn for dangling an object, or creating yarn for decoration, and moved toward modeling yarn for the sake of the yarn itself. In the following paper, we examine three different curves, their pros and cons for modeling yarn, physics on those curves, and procedural animation of the curves for the purpose of creating yarn that simply exists to be played with.

1. INTRODUCTION

Our project animates a yarn object made of appended curve fragments using a mix of procedural animation and physics animation. We were unable to meet our initial goals of collision detection and yarn-yarn or yarn-hook interaction. However, in lieu of those goals, we created other yarn-based animations using customizeable sine-wave based functions on a curve that can be drawn using Bezier, B-spline, or Catmull-Rom curve blending using matrix multiplication. We focused our efforts on the animation portion of the project and are pleased with the progress we were able to accomplish, even if it was not directly the same as our initial proposal.

2. RELATED WORKS

The inspiration for this project came from the senior Game Design Sequence (CMPS 170-172) that both April and Julie are participating in. Both authors are part of the same team, U.S.E.D., which is responsible for creating Pattern[**?**], a crochet simulator meant to teach the player about the craft of crochet without the frustrating learning curve in handling a crochet hook and yarn. While both authors planned to use canned animations for the game project, this course offered a unique opportunity to explore more thorough curve research and animations for the game instead. See the References Section (Section 7) for more information about Pattern.

3. TECHNICAL DETAIL

Our project's design combines a number of elements that were introduced in CMPS 161 this quarter, including but not limited to curves and their matrices, physics-based animation, and procedural animation. There are also elements of CMPS 160 that can be seen, such as camera control, primitive-drawing, and texture mapping.

3.1 Curves

In contrast to most projects and assignments required for CMPS 161, we built our project in XNA 3.1 using C# rather than using C++ and OpenGL. This decision was made based on our related work, Pattern, being in XNA 3.1 before the beginning of this quarter. As such, we were not able to use many of OpenGL's curve functions and had to make them ourselves.

All of the following cubic curves (that is, all of them except the nth degree bezier) are computed using the following formula:

$$TMG$$

where T is a row matrix representing our cubic curves:

$$T_{Cubic} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}$$

And G is a column matrix representing the four consecutive control points starting at P:

$$G_{Splines} = \begin{bmatrix} P_i \\ P_{i+1} \\ P_{i+2} \\ P_{i+3} \end{bmatrix}$$

The M for each curve is different and can be found in their respective sections. The two curves in our project that currently implement the above setup (B-Spline and Catmull-Rom) correctly function because their G's are based off of the four points. Hermite and Bezier have further restrictions on their points to make the curves properly, and were left out of our final project. Still, we researched them, so they will be included in this paper's discussions.

In our project, the yarn object is representing as a pair of lists: a list of control points and a list of curve points. Control points are altered by the physics, procedural animation, and directly by the user, and are the only means with which the user can alter the curve. The curve itself is drawn using the curve points, which are calculated using the TMG matrix multiplication above. At their core, curve points are simply a list of points, but with cylinders drawn over them they give our curve its yarn properties.

3.1.1 Bezier

The original design that we demonstrated mid-quarter was based on a single nth-order bezier curve, where n was the number of control points minus one. However, as Alex Pang warned, dealing with Bezier curves in an environment where we wanted to add line segments and animate their control points was extraordinarily messy. We were able to add three control points per segment in order to maintain curve continuity between the segments. Each segment used the following matrix as a blending function[**?**]:

$$M_{Bezier} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

However, we were not able to easily ensure the continuity between adjacent Bezier segments upon application of physics and procedural animation. Also, because of the following two curves (Section 3.1.2, Section 3.1.3), generating one segment per newly introduced control point, where Bezier generates one segment per three control points, the curve would not be as long nor fit properly into our data structures. Using Bezier segments was not intuitive or flexible enough for our purposes. In a similar experiment, we also tested Hermite curves, but the miss-match between number of control points also made it inconveneint to use. For posterity's sake, we kept the segmented Bezier algorithm and Hermite curve in our program even though our project does not make use of them. Here is the blending matrix for the Hermite curve[**?**]:

$$M_{Hermite} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Ultimately, we ended up keeping the single nth-order bezier curve to compare/contrast with the other curves we introduced into the system (Section 3.1.2, Section 3.1.3). The bezier algorithm that handles nth-order curves was derived from the Wikipedia website on Bezier curves[**?**]. The algorithm uses a combination of factorials and combinatorials in order to handle the varying degrees it can be given.

3.1.2 B-Spline

Our first Bezier-replacement was the uniform cubic B-spline[**?**]. We were primarily attracted to it as a result of its approximating its control points rather than interpolating them, as well as its ability to maintain C2 continuity in its segments after only adding a single control point[**?**]. Our algorithm applies the uniform cublic B-spline blending function to a set of four consecutive control points[**?**]:

$$M_{B-Spline} = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

In order to draw the whole curve, we iterate through the blending function using n control points (P) as follows:

$$B - Spline\_Blend(P_i, P_{i+1}, P_{i+2}, P_{i+3})$$

where $n \leq 4$ and $0 < i \leq n$-3.

3.1.3 Catmull-Rom

As an experiment, we also wanted to test out what our curve would look like when applied to an algorithm with C1 continuity[**?**] that interpolates some of its control points rather than approximating them all. Professor Pang described the Catmull-Rom Spline curve in class, and because of its similar structure to uniform cubic B-splines, we decided to implement it. It uses the same blending method was the uniform cubic B-spline:

$$Catmull - Rom\_Blend(P_i, P_{i+1}, P_{i+2}, P_{i+3})$$

where $n \leq 4$ and $0 < i \leq n$-3

However, the blend function accesses a different matrix, and thus a different parametric equation[**?**]:

$$M_{Catmull-Rom} = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

3.1.4 Controlling the Individual Control Point

Each individual control point can also be moved manually. The control point can be moved in the x, y and z direction. This functionality gives the user freedom to shape the yarn however they want.

3.2 Cylinders and Camera

The authors were able to make use of CMPS 160 knowledge for this project, and those contributions are worth noting here. First, we built a cylinder primitive using triangle lists and wrapped a single yarn texture around the cylinder. We aligned the texture on the cylinder such that when the cylinders lay end-to-end, the texture moves smoothly from one cylinder to the next. Secondly, we had to orient the cylinder in 3D space using matrix multiplications in the correct order. We laid the cylinders on the curve from curve point to curve point (where those points are defined by the curve blend functions (Sections 3.1.1 - 3.1.3)), and retained the angular information of that line segment for use in orienting the cylinder. Finally, we have a camera that the user can zoom, pan, and change between a front view and a side view of the yarn also using matrix multiplications.

3.3 Physics

The purpose of physics is to arrange the position of the control points based on user input. Physics is composed of two important functions: calculating the net force and determining the curvature. Each time physics is called the

net force is calculated. Curvature then uses the net force to determine the amount of curvature.

3.3.1 Finding the Net Force

The net force is equal to the constant minus the force of user input.

$$Ftotal = |Fconstant||Finput|$$

The user input is received either through the left joystick on the controller or through keys on the keyboard. If the left stick is not being moved in any way, the force from user input is equal to zero. As pressure is applied to the left stick, the force due to user input increases. Input from the keyboard works a little different. Four different keys are used to control the amount of force applied in the positive and negative direction along the x and y axis. If the user stops giving input, the yarn will freeze at that position rather than falling back to zero like in the case of the controller input.

Once the amount of force applied in the x and y diretion is recorded, the total *Finput* is calculated using the Pythagorean Theorem.

$$Finput = \sqrt{FXinput^2 + FYinput^2}$$

The *Finput* can never exceed the *Fconstant*. If the input force is greater than the constant force, the input force is reassigned to equal the constant force.

3.3.2 Finding the Curvature

Before Curvature redistributes the control points, a call will be made to the method that calculates the net force. The net force will help determine the placement of the control points for accurate curvature based on input. As shown in Figure 1, if $|Fconstant| > |Finput|$ the yarn will start to curve. The less the input force is then the constant force, the greater the curve in the yarn. When the forces are in equilibrium, $|Fconstant| = |Finput|$, the yarn is completely straight. Figure 2 shows the appearance of the yarn when the forces are in equilibrium.

Before the control points can be distributed, the pivot point needs to be determined. The pivot point and all control points that come before are not affected by the physics. Each control point between the pivot point and the end point are evaluated and redistributed in the x and y direction. Input from the user should cause the yarn to move in a radial like motion where the pivot point is the center of the circular path.

When the yarn is at rest, the affected control points all have the same x position. As the user applies a force the, the points start to spread out. Regardless of the amount of user force, as long as itŠs not zero, the control points will be evenly space between the pivot point and the end point. The more force, the more spread apart the control points get. The amount of spreading is directly affected by the net force.

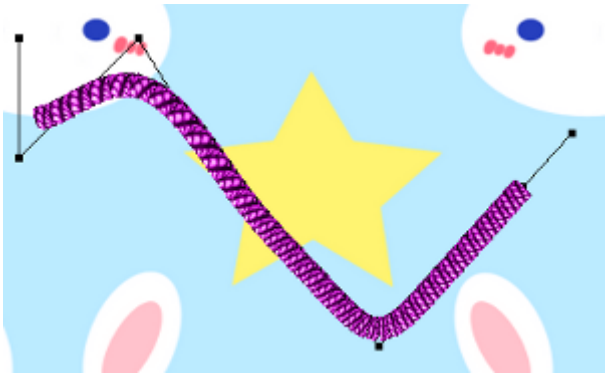

Figure 1: Image of the yarn when $|Fconstant| > |Finput|$

Figure 2: Image of the yarn when $|Fconstant| = |Finput|$

At rest, the affected control points are evenly spaced apart in the y position. Once the user applies a force, the control points form a v shape. The pivot and end point are the two high points in the v. As the user applies more force, the points will flatten their v shape and become linear. The sharpness of the drop in the v shape is directly affected by the net force.

3.4 Procedural Animation

In lieu of collision and inter-yarn animation, we decided to experiment with procedural animation. A simple algorithm that is string- and wave-related, the sine wave, seemed like a clear-cut choice. After CMPS 161 Program 4[**?**], the authors realized how interesting it was to set up a system that the user could simply play with, and that is what the procedural animation section is based off of.

3.4.1 Sine Wave

The sine wave follows a straight-forward equation that many students learn in trigonometry[**?**]:

$$y(t) = A \cdot sin(\omega t + \varphi)$$

We applied this formula to our control points, along with the ability for the user to control the amplitude (A), wavelength (ω), and speed or rate (φ). Using this equation, you can simulate a number of animations that can be done with yarn, such as playing jump-rope with the yarn, swinging the yarn between your fingers, or shaking one end of the yarn. The goal of this portion of the project was to interpolate control points in a predictable pattern and see how the yarn reacted. Each of the three curves (Sections 3.1.1 - 3.1.3) show vastly different animations with the same control point animations.

4. RESULTS

In terms of displaying yarn, we found the uniform cubic B-splines to be the best choice of the methods we tried. It offers the smoothest use of control points because of its approximating them, offering the highest amount of continuity, and is ultimately the simplest to use (in only having to add a single control point per segment, as opposed to Bezier). As a note specific to our CMPS 161 class, real-time collision detection between animated oddly shaped and oriented objects is far too difficult to attempt to accomplish in the span of a single final project. We are pleased with what we were able to accomplish, both in exploring curve algorithms and in animating those curves. We are glad that we scrapped our collision-based animations in favor of more free-form ones.

5. CONCLUSION

Our project has explored at length the pros and cons of various curve drawing methods for yarn and other string-like objects, as well as movement of those curves through space. In our project, physics forces and animation forces control the curve as a puppet master would: using the control points as marionette handles and allowing the curve blending functions to draw the resulting object. We did not use canned spring forces for our yarn, and instead experimented with new ways to orient the control points in order to achieve the result of a nicely draping curve. Finally, we took advantage of the yarnŠs curve structure and animated it based on the oscillating sinusoidal wave function to simulate a piece of yarn with different kinds of forces acting upon it. In future work, we hope to successfully integrate collision detection and resolution in order to allow yarn-yarn and yarn-hook interactions to also take place.

6. ACKNOWLEDGEMENTS

The authors would like to acknowledge William Tuttle for his help with the nth degree Bezier design and primitive drawing in XNA 3.1.

7. REFERENCES

[1] Alex Pang. Ucsc cmps 161 winter 2011 program 4. Online, 2011. http://users.soe.ucsc.edu/ pang/161/w11/prog4/.

[2] Rick Parent. *Computer Animation: Algorithms and Techniques*. Morgan Kaufmann Publishers, Burlington, MA, 2nd edition, 2008.

[3] Wikipedia. B-spline — wikipedia, the free encyclopedia. Online, 2011. http://en.wikipedia.org/wiki/B-spline.

[4] Wikipedia. Bezier — wikipedia, the free encyclopedia. Online, 2011. http://en.wikipedia.org/wiki/B%C3%A9zier_curve.

[5] Wikipedia. Sine wave — wikipedia, the free encyclopedia. Online, 2011. http://en.wikipedia.org/wiki/Sine_wave.

[6] Wikipedia. Smooth function — wikipedia, the free encyclopedia. Online, 2011. http://en.wikipedia.org/wiki/Smooth_function.

[7] Katarina Yang. Pattern. Online, 2011. http://people.ucsc.edu/ agrow/pattern/.