Kochanek-Bartels Cubic Splines (TCB Splines)

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: July 16, 1999
Last Modified: August 25, 2022

Contents

1 Introduction

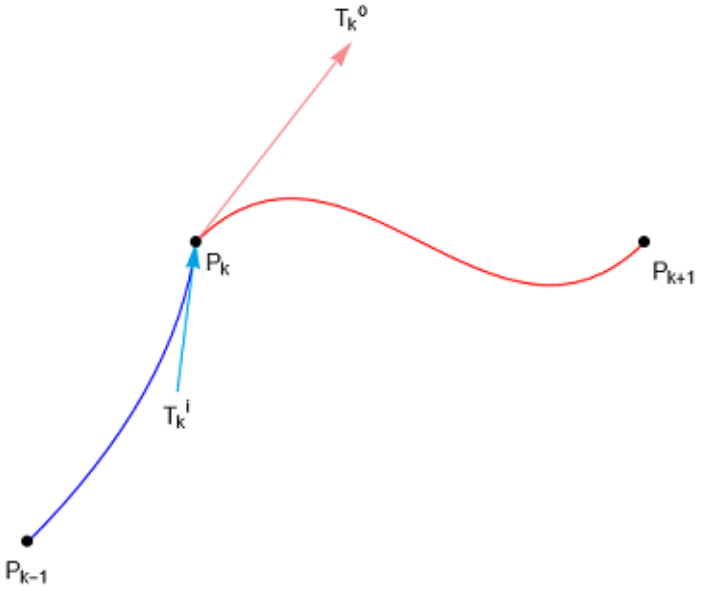
A sequence of positional key frames is

$$\left\{\left(s_k, \boldsymbol{P}_k, \boldsymbol{T}_k^i, \boldsymbol{T}_k^o\right)\right\}_{k=0}^{n-1} \tag{1}$$

where s_k is the *sample time*, $\boldsymbol{P}_k$ is the *sample position*, $\boldsymbol{T}_k^i$ is the *incoming tangent vector* and $\boldsymbol{T}_k^o$ is the *outgoing tangent vector*. The Kochanek–Bartels spline [1] specifies a cubic polynomial interpolation between each pair of key frames by choosing the incoming and outgoing tangents in a special way. The incoming and outgoing tangent vectors at $\boldsymbol{P}_k$ need not be the same vector, in which case the spline has a derivative discontinuity at that position. The tangents at $\boldsymbol{P}_k$ are chosen based on neighboring positions and on three parameters that have some visual appeal. The parameters are *tension* τ_k, which controls how sharply the curve bends at a control point; *continuity* γ_k, which controls the continuity or discontinuity at a position; and *bias* β_k, which controls the direction of the path at $\boldsymbol{P}_k$ by using weighted combination of one-sided derivatives at that position.

Figure 1 shows a typical position, tangent vectors and the curve segments passing through the position. All the figures in this document were drawn using Mathematica [2].

Figure 1. A cubic spline curve passing through point $\boldsymbol{P}_k$ with incoming tangent $\boldsymbol{T}_k^i$ and outgoing tangent $\boldsymbol{T}_k^o$. As you walk from $\boldsymbol{P}_{k-1}$ to $\boldsymbol{P}_k$, the direction you reach $\boldsymbol{P}_k$ is $\boldsymbol{T}_k^i$, the incoming direction to $\boldsymbol{P}_k$. As you walk from $\boldsymbol{P}_k$ to $\boldsymbol{P}_{k+1}$, the direction you leave $\boldsymbol{P}_k$ is $\boldsymbol{T}_k^o$, the outgoing direction from $\boldsymbol{P}_k$. Each of the direction vectors at $\boldsymbol{P}_k$ is tangent to the relevant curve.



The Kochanek–Bartels splines are sometimes called *TCB splines*, the acronoym TCB referring to tension, continuity and bias.

2 Cubic Polynomial Curves

In general, a cubic polynomial curve can be constructed for a pair of key frames $(s_k, \boldsymbol{P}_k, \boldsymbol{T}_k^i, \boldsymbol{T}_k^o)$ and $(s_{k+1}, \boldsymbol{P}_{k+1}, \boldsymbol{T}_{k+1}^i, \boldsymbol{T}_{k+1}^o)$. The curve is of the form

$$\boldsymbol{X}_k(s) = \boldsymbol{A}_k + \left(\frac{s-s_k}{\Delta_k}\right)\boldsymbol{B}_k + \left(\frac{s-s_k}{\Delta_k}\right)^2\boldsymbol{C}_k + \left(\frac{s-s_k}{\Delta_k}\right)^3\boldsymbol{D}_k \tag{2}$$

where $\Delta_k = s_{k+1} - s_k$ and where $s \in [s_k, s_{k+1}]$. The first-order derivative is

$$\boldsymbol{X}_k'(s) = \frac{1}{\Delta_k}\left(\boldsymbol{B}_k + 2\left(\frac{s-s_k}{\Delta_k}\right)\boldsymbol{C}_k + 3\left(\frac{s-s_k}{\Delta_k}\right)^2\boldsymbol{D}_k\right) \tag{3}$$

the second-order derivative is

$$\boldsymbol{X}_k''(s) = \frac{1}{\Delta_k^2}\left(2\boldsymbol{C}_k + 6\left(\frac{s-s_k}{\Delta_k}\right)\boldsymbol{D}_k\right) \tag{4}$$

and the third-order derivative is

$$\boldsymbol{X}_k'''(s) = \frac{1}{\Delta_k^3}(6\boldsymbol{D}_k) \tag{5}$$

The derivatives of order larger than 3 are zero-valued vectors. The vector-valued coefficients of the polynomial are determined by requiring

$$\boldsymbol{X}_k(s_k) = \boldsymbol{P}_k,\ \ \boldsymbol{X}_k(s_{k+1}) = \boldsymbol{P}_{k+1},\ \ \boldsymbol{X}_k'(s_k) = \boldsymbol{T}_k^o,\ \ \boldsymbol{X}_k'(s_{k+1}) = \boldsymbol{T}_{k+1}^i \tag{6}$$

The curve must pass through the endpoints and the curve derivatives at the endpoints must match the specified tangent vectors. The four linear equations implied by these conditions are

$$\begin{aligned}
&\boldsymbol{A}_k = \boldsymbol{P}_k\\
&\boldsymbol{A}_k + \boldsymbol{B}_k + \boldsymbol{C}_k + \boldsymbol{D}_k = \boldsymbol{P}_{k+1}\\
&\boldsymbol{B}_k = \Delta_k\boldsymbol{T}_k^o\\
&\boldsymbol{B}_k + 2\boldsymbol{C}_k + 3\boldsymbol{D}_k = \Delta_k\boldsymbol{T}_{k+1}^i
\end{aligned} \tag{7}$$

The solution is

$$\begin{aligned}
\boldsymbol{A}_k &= \boldsymbol{P}_k\\
\boldsymbol{B}_k &= \Delta_k\boldsymbol{T}_k^o\\
\boldsymbol{C}_k &= 3(\boldsymbol{P}_{k+1} - \boldsymbol{P}_k) - \Delta_k(2\boldsymbol{T}_k^o + \boldsymbol{T}_{k+1}^i)\\
\boldsymbol{D}_k &= -2(\boldsymbol{P}_{k+1} - \boldsymbol{P}_k) + \Delta_k(\boldsymbol{T}_k^o + \boldsymbol{T}_{k+1}^i)
\end{aligned} \tag{8}$$

The curve may be rewritten in terms of an Hermite interpolation basis $H_0(u) = 2u^3 - 3u^2 + 1$, $H_1(u) = -2u^3 + 3u^2$, $H_2(u) = u^3 - 2u^2 + u$ and $H_3(u) = u^3 - u^2$ for $u \in [0, 1]$, namely,

$$\boldsymbol{X}_k(s) = H_0\left(\frac{s-s_k}{\Delta_k}\right)\boldsymbol{P}_k + H_1\left(\frac{s-s_k}{\Delta_k}\right)\boldsymbol{P}_{k+1} + H_2\left(\frac{s-s_k}{\Delta_k}\right)\Delta_k\boldsymbol{T}_k^i + H_3\left(\frac{s-s_k}{\Delta_k}\right)\Delta_k\boldsymbol{T}_{k+1}^i \tag{9}$$

A spline is formed by the sequence of curves $\boldsymbol{X}_k(s)$ for $0 \le k \le n-2$. Each segment is continuous and differentiable. The spline is continuous over all segments because it passes through all the sample positions.

Whether or not the spline is differentiable at the sample positions depends on the choice of incoming and outgoing tangent vectors. The spline is G^1 continuous at position $\boldsymbol{P}_k$ when the incoming and outgoing tangents $\boldsymbol{T}_k^i$ and $\boldsymbol{T}_k^o$ are in the same direction but not necessarily of equal lengths. The spline is C^1 continuous at the sample position if the incoming and outgoing tangents are equal.

3 Choosing Tangent Vectors

Let us now look at the choices for tangent vectors based on the tension, continuity and bias parameters. The formulation in [1] implicitly assumes that $s_k = k$, which is uniform spacing of the sample times. Section 4 of the article discusses choosing tangent vectors when nonuniform sample times are used. The goal is to obtain continuity of speed at the sample positions but not necessarily continuity of position. The authors' derivation requires more attention than what is provided in the article; I will discuss this in the next section.

The default values for tension τ_k, continuity γ_k and bias β_k are all zero. In this case the incoming and outgoing tangent vectors are chosen to be the same vector,

$$\boldsymbol{T}_k^i = \boldsymbol{T}_k^o = \frac{1}{2}((\boldsymbol{P}_{k+1} - \boldsymbol{P}_k) + (\boldsymbol{P}_k - \boldsymbol{P}_{k-1})) = \frac{\boldsymbol{P}_{k+1} - \boldsymbol{P}_{k-1}}{2} \tag{10}$$

The units of the tangent vectors are position divided by time. The right-hand side of Equation (10) appears to have units of position. To make the left-hand sides and the right-hand side commensurate, think of the denominator 2 of the right-hand side having units of time, thus making the units match. This choice of tangents leads to the *Catmull–Rom spline*. The tangent vector at $\boldsymbol{P}_k$ is chosen to be parallel to the secant vector from $\boldsymbol{P}_{k-1}$ to $\boldsymbol{P}_{k+1}$ as shown in Figure 2,

Figure 2. The Catmull–Rom spline. The tangent vector at $\boldsymbol{P}_k$ (blue) is chosen to be parallel to the secant vector (red) connecting the neighboring positions.

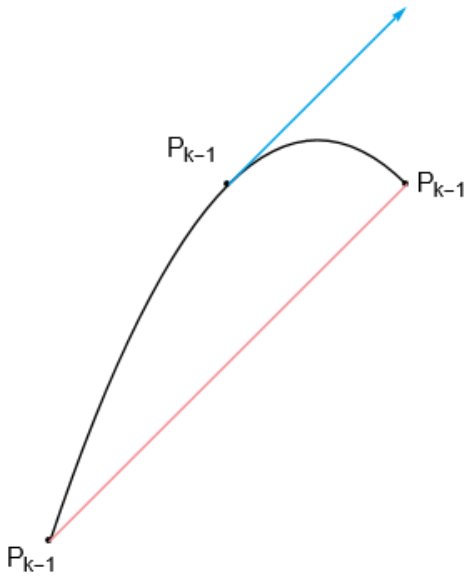
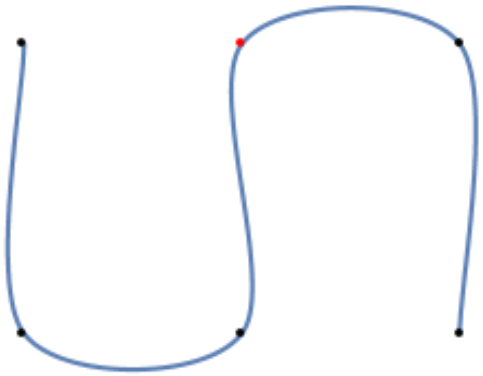


Figure 3 shows a Catmull–Rom spline for 6 sample positions. The spline has C^1 continuity everywhere.

Figure 3. The Catmull–Rom spline for 6 sample positions $(s_k, \boldsymbol{P}_k)$, namely, $(0, (14, 256))$, $(1, (14, 86))$, $(2, (142, 86))$, $(3, (142, 256))$, $(4, (270, 256))$ and $(5, (270, 86))$.



The remainder of the discussion examines the shape of the curve as tension, continuity and bias are modified at the red point in the figure.

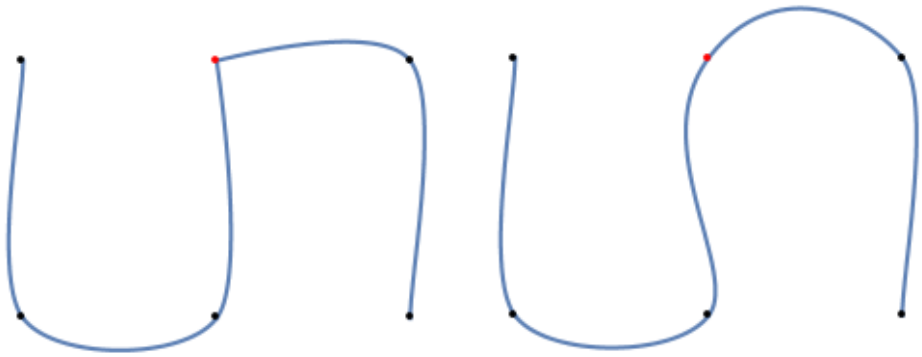
3.1 Tension

Tension $\tau_k \in [-1, 1]$ is introduced as a parameter in Equation (10) as shown next,

$$\boldsymbol{T}_k^i = \boldsymbol{T}_k^o = \frac{(1 - \tau_k)}{2}((\boldsymbol{P}_{k+1} - \boldsymbol{P}_k) + (\boldsymbol{P}_k - \boldsymbol{P}_{k-1})) \tag{11}$$

As noted previously, the units must match between the left-hand and right-hand sides. The tension is dimensionless, so the 2 in the denominator has units of time. If $\tau_k = 0$, then we have the Catmull–Rom spline. For τ_k near 1 the curve shows *tightness* at the control point whereas τ_k near -1 shows *slack* at the control point. Varying τ_k changes the length of the tangent at the control point, a smaller-length tangent causing tightening and a larger-length tangent causing slackening. Figure 4 shows nonzero tension values applied to the red sample position of Figure 3.

Figure 4. Left: Tension $\tau_3 = 1$ at $\boldsymbol{P}_3$. Right: Tension $\tau_3 = -1$ at $\boldsymbol{P}_3$.



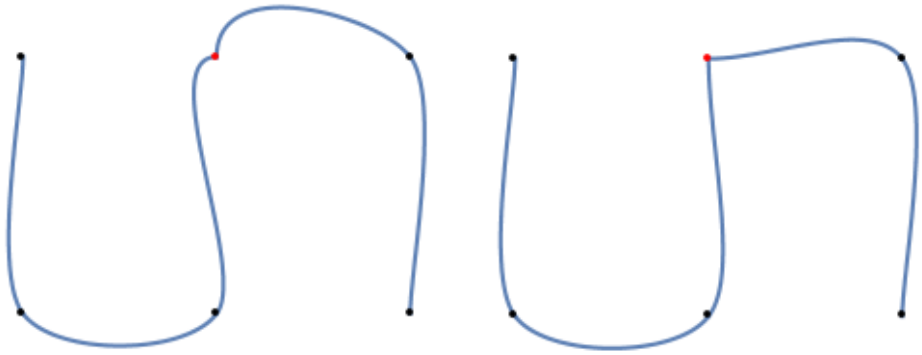
3.2 Continuity

Continuity $\gamma_k \in [-1, 1]$ is introduced as a parameter in Equation (10) as shown next. Notice that the incoming and outgoing tangents are generally different.

$$\begin{aligned} \boldsymbol{T}_k^i &= \tfrac{1+\gamma_k}{2}(\boldsymbol{P}_{k+1} - \boldsymbol{P}_k) + \tfrac{1-\gamma_k}{2}(\boldsymbol{P}_k - \boldsymbol{P}_{k-1}) \\ \boldsymbol{T}_k^o &= \tfrac{1-\gamma_k}{2}(\boldsymbol{P}_{k+1} - \boldsymbol{P}_k) + \tfrac{1+\gamma_k}{2}(\boldsymbol{P}_k - \boldsymbol{P}_{k-1}) \end{aligned} \tag{12}$$

Once again the 2 in the denominator has units of time so that the units match between the left-hand and right-hand sides. When $\gamma_k = 0$, the curve is the Catmull–Rom curve and has a continuous tangent vector at the control point. As $|\gamma_k|$ increases, the resulting curve has a *corner* at the control point, the direction of the corner depending on the sign of γ_k. Figure 5 shows nonzero continuity values applied to the red sample position of Figure 3.

Figure 5. Left: Continuity $\gamma_3 = 1$ at $\boldsymbol{P}_3$. Right: Continuity $\gamma_3 = -1$ at $\boldsymbol{P}_3$.



In both images of the figure it is apparent that the curve has neither C^1 nor G^1 continuity at the red sample position.

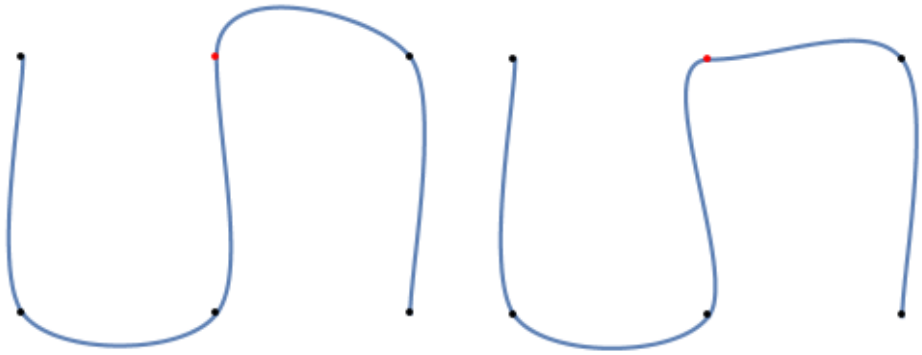
3.3 Bias

Bias $\beta_k \in [-1, 1]$ is introduced as a parameter in Equation (10) as shown next. As with tension, the incoming and outgoing tangents are chosen to be equal.

$$\boldsymbol{T}_k^i = \boldsymbol{T}_k^o = \frac{1-\beta_k}{2}(\boldsymbol{P}_{k+1} - \boldsymbol{P}_k) + \frac{1+\beta_k}{2}(\boldsymbol{P}_k - \boldsymbol{P}_{k-1}) \tag{13}$$

As in the other cases, the 2 in the denominator has units of time so that the units match between the left-hand and right-hand sides. When $\beta_k = 0$, the left and right one-sided tangents are equally weighted, producing the Catmull–Rom spline. For β_k near -1, the outgoing tangent dominates the direction of the path of the curve through the control point (*undershoot*). For β_k near 1, the incoming tangent dominates (*overshoot*). Figure 6 shows nonzero continuity values applied to the red sample position of Figure 3.

Figure 6. Left: Bias $\beta_3 = 1$ at $\boldsymbol{P}_3$. Right: Bias $\beta_3 = -1$ at $\boldsymbol{P}_3$.



The three effects may be combined into a pair of equations

$$\boldsymbol{T}_k^i = \frac{(1-\tau_k)(1+\gamma_k)(1-\beta_k)}{2}(\boldsymbol{P}_{k+1} - \boldsymbol{P}_k) + \frac{(1-\tau_k)(1-\gamma_k)(1+\beta_k)}{2}(\boldsymbol{P}_k - \boldsymbol{P}_{k-1}) \tag{14}$$

and

$$\boldsymbol{T}_k^o = \frac{(1-\tau_k)(1-\gamma_k)(1-\beta_k)}{2}(\boldsymbol{P}_{k+1} - \boldsymbol{P}_k) + \frac{(1-\tau_k)(1+\gamma_k)(1+\beta_k)}{2}(\boldsymbol{P}_k - \boldsymbol{P}_{k-1}) \tag{15}$$

4 Continuity of Speed

Section 4 of [1] discusses weighting of the tangent vectors in Equations (14) and (15) in order to account for nonuniform sample times. The following is a quote from that section.

> If we assume default continuity ($c = 0$) at key $\boldsymbol{P}_i$, the spline segment between $\boldsymbol{P}_i$ and $\boldsymbol{P}_{i+1}$ should join smoothly with the segment used between $\boldsymbol{P}_{i-1}$ and $\boldsymbol{P}_i$. While this is true when looking only at the path of the motion, it may not necessarily be true for the speed of motion. The forumlas given in Equation 8 and Equation 9 assume an equal number of inbetweens within each key interval. A problem can exist if the animator requests a different number of inbetweens for adjacent intervals. Consider the case where 10 inbetweens are supposed to be generated in the interval from $\boldsymbol{P}_{i-1}$ to $\boldsymbol{P}_i$, but only 5 inbetweens between $\boldsymbol{P}_i$ and $\boldsymbol{P}_{i+1}$ as was shown in Figure 2. In the first interval, the step size for the interpolation parameter s will be $\Delta_1 = 1/11$ whereas for the second interval the step size will be $\Delta_2 = 1/6$. If the same parametric derivative is used for both splines at $\boldsymbol{P}_i$, these different step sizes will cause a discontinuity in the speed of motion. What is required, if this discontinuity is not intentional, is a means of making a local adjustment to the interval separating successive frames before and after the key frame so that the speed of entry matches the speed of exit. This can be accomplished by adjusting the specification of the tangent vector at the key frame based on the number of inbetweens in the adjacent intervals. In practice this turns out to be very simple, because we have already made provisions for two distinct tangent vectors at each key position in order to accommodate the continuity control parameter. Once the tangent vectors have been found for an equal number of inbetweens in the adjacent intervals, the adjustments required for different numbers of inbetweens (N_{i-1} frames between $\boldsymbol{P}_{i-1}$ and $\boldsymbol{P}_i$ followed by N_i frames between $\boldsymbol{P}_i$ and $\boldsymbol{P}_{i+1}$) can be made by weighting the tangent vectors appropriately:

The c-value refers to the continuity parameter that I have called γ. The index i refers to the index that I have called k. Equations 8 and 9 in the article are my equations (14) and (15). The reference to Figure 2 of the article is irrelevant to the discussion here. Using my notation, the authors' proposed solution is to adjust the lengths of the tangent vectors to obtain new tangent vectors,

$$\widehat{\boldsymbol{T}}_k^i = \frac{2N_k}{N_{k-1}+N_k}\boldsymbol{T}_k^i, \;\; \widehat{\boldsymbol{T}}_k^o = \frac{2N_{k-1}}{N_{k-1}+N_k}\boldsymbol{T}_k^o \tag{16}$$

The authors are confounding the concepts of *continuous variable* and *discrete variable*. The premise of the article is to design an interpolation algorithm using the *continuous variable* s. The outcome is the TCB spline function. As a function of a continuous variable, the methods of calculus apply. The TCB spline function computes the position $\boldsymbol{P}(s)$ at time s. The derivative of the TCB spline function computes the velocity $\boldsymbol{T}(s)$ at time s, which is the tangent vector to the spline curve at position $\boldsymbol{P}(s)$. As a particle moves along the curve, the speed of the particle is the length of the velocity vector $|\boldsymbol{T}(s)|$. In terms of the continuous variable s, the continuity of the speed of motion at a sample position $\boldsymbol{P}_k$ is guaranteed when the lengths of the incoming and outgoing tangent vectors are the same; that is, $|\boldsymbol{T}_k^i| = |\boldsymbol{T}_k^o|$. The direction of motion is not necessarily continuous; even though the lengths of the tangent vectors are equal, the tangent vectors themselves are not guaranteed to be equal.

The adjustment coefficiencts proposed by the authors are an intuitive appeal to how an animator selects times at which the TCB spline function is evaluated. Let $\boldsymbol{P}(s)$ be the TCB spline function for $s \in [s_{\min}, s_{\max}]$. Effectively the animator chooses a finite set of M times $\tilde{s}_j \in [s_{\min}, s_{\max}]$ to produce a finite set of M positions $\widetilde{\boldsymbol{P}}_j = \boldsymbol{P}(\tilde{s}_j)$ for $0 \le j < M$. The sequence $\{(\tilde{s}_j, \widetilde{\boldsymbol{P}}_j)\}$ is indexed by the *discrete variable* j. The authors mention "these different step sizes will cause a discontinuity in the speed of motion," which is reference to a perceived visual anomaly in the speed which is based on number of samples (using a discrete variable j) and not on the length of the tangent vectors (using a continuous variable s). The anomaly is about the animators choices for $\tilde{s}_j$ and in fact about choices for the key frames.

Although in practice the adjustment based on number of inbetweens might produce acceptable results, I prefer to separate the choice of inbetweens from the interpolation function. If an animator is going to select inbetweens as an offline process for an application, the TCB spline function can be hard-coded to use the inbetween counts N_k. If the inbetweens are determined at runtime, the counts N_k must be known first in order to compute the incoming and outgoing tangents after which the TCB spline polynomial coefficients can be computed. This is problematic if the application does not know how to select N_k algorithmically, for example, if the samples are generated by a clock.

In my opinion, the correct approach to obtain continuity of speed requires two modifications. First, requiring that the denominator 2 have units of time is not natural. Instead, the time samples need to be involved in the computation of incoming and outgoing tangents; in fact, the tangents should be weighted averages of velocity vectors,

$$\begin{aligned}
\boldsymbol{T}_k^i &= \frac{(1-\tau_k)(1+\gamma_k)(1-\beta_k)}{2}\left(\frac{\boldsymbol{P}_{k+1}-\boldsymbol{P}_k}{\Delta_k}\right) + \frac{(1-\tau_k)(1-\gamma_k)(1+\beta_k)}{2}\left(\frac{\boldsymbol{P}_k-\boldsymbol{P}_{k-1}}{\Delta_{k-1}}\right) \\
\boldsymbol{T}_k^o &= \frac{(1-\tau_k)(1-\gamma_k)(1-\beta_k)}{2}\left(\frac{\boldsymbol{P}_{k+1}-\boldsymbol{P}_k}{\Delta_k}\right) + \frac{(1-\tau_k)(1+\gamma_k)(1+\beta_k)}{2}\left(\frac{\boldsymbol{P}_k-\boldsymbol{P}_{k-1}}{\Delta_{k-1}}\right)
\end{aligned} \tag{17}$$

When $\gamma_k = 0$ the incoming and outgoing tangents are equal, so the curve has C^1 continuity at $\boldsymbol{P}_k$ regardless of the choice of tension and bias. When $\tau_k = \gamma_k = \beta_k = 0$ and the sample times are $s_k = k$, we obtain the Catmull–Rom spline.

Second, when $\gamma_k \neq 0$, if you decide you want continuity of speed even though you do not have continuity of direction, adjustments must be made to obtain incoming and outgoing tangents that have the same length. Use the incoming and outgoing tangents

$$\widehat{\boldsymbol{T}}_k^i = \frac{2|\boldsymbol{T}_k^o|}{|\boldsymbol{T}_k^i| + |\boldsymbol{T}_k^o|}\,\boldsymbol{T}_k^i, \quad \widehat{\boldsymbol{T}}_k^o = \frac{2|\boldsymbol{T}_k^i|}{|\boldsymbol{T}_k^i| + |\boldsymbol{T}_k^o|}\,\boldsymbol{T}_k^o \tag{18}$$

where $\boldsymbol{T}_k^i$ and $\boldsymbol{T}_k^o$ are defined by Equation (17). I included the factor 2 so that Equation (18) has a form similar to that of Equation (16).

Using Equation (18), the speed at $\boldsymbol{P}_k$ is $2|\boldsymbol{T}_k^i||\boldsymbol{T}_k^o|/(|\boldsymbol{T}_k^i| + |\boldsymbol{T}_k^o|)$. An additional degree of freedom, say, $\lambda_k > 0$, can be provided at the sample position $\boldsymbol{P}_k$ that allows control of the speed at that position. Use the incoming and outgoing tangents

$$\widehat{\boldsymbol{T}}_k^i = \frac{2\lambda_k|\boldsymbol{T}_k^o|}{|\boldsymbol{T}_k^i| + |\boldsymbol{T}_k^o|}\,\boldsymbol{T}_k^i, \quad \widehat{\boldsymbol{T}}_k^o = \frac{2\lambda_k|\boldsymbol{T}_k^i|}{|\boldsymbol{T}_k^i| + |\boldsymbol{T}_k^o|}\,\boldsymbol{T}_k^o \tag{19}$$

where $\boldsymbol{T}_k^i$ and $\boldsymbol{T}_k^o$ are defined by Equation (17). The speed is $2\lambda_k|\boldsymbol{T}_k^i||\boldsymbol{T}_k^o|/(|\boldsymbol{T}_k^i| + |\boldsymbol{T}_k^o|)$. Equation (18) is a specialization of Equation (19) where $\lambda_k = 1$ for all k.

5 Boundary Conditions

The discussion has been presented as if $\boldsymbol{P}_k$ were an interior point of the sequence of samples. However, it is necessary to provide computations for the outgoing tangent at $\boldsymbol{P}_0$ and for the incoming tangent at $\boldsymbol{P}_{n-1}$. Equation (17) implies abstractly that $\boldsymbol{T}_0^o$ is a linear combination of $(\boldsymbol{P}_1 - \boldsymbol{P}_0)$ and $(\boldsymbol{P}_0 - \boldsymbol{P}_{-1})$. However, the index -1 is out of range. The outgoing tangent can be chosen as

$$\boldsymbol{T}_0^o = \frac{(1-\tau_0)(1-\gamma_0)(1-\beta_0)}{2}\left(\frac{\boldsymbol{P}_1 - \boldsymbol{P}_0}{\Delta_0}\right) \tag{20}$$

Similarly, $\boldsymbol{T}_{n-1}^i$ is a linear combination of $(\boldsymbol{P}_n - \boldsymbol{P}_{n-1})$ and $(\boldsymbol{P}_{n-1} - \boldsymbol{P}_{n-2})$. However, the index n is out of range. The incoming tangent can be chosen as

$$\boldsymbol{T}_{n-1}^i = \frac{(1-\tau_{n-1})(1-\gamma_{n-1})(1+\beta_{n-1})}{2}\left(\frac{\boldsymbol{P}_{n-1} - \boldsymbol{P}_{n-2}}{\Delta_{n-2}}\right) \tag{21}$$

I used this approach to generate the splines shown in Figures 3 through 6.

Alternatively, an implementation may allow the user to specify the boundary tangent vectors $\boldsymbol{T}_0^o$ and $\boldsymbol{T}_{n-1}^i$.

Regardless of how the boundary tangent vectors are computed, if the λ parameters are specified to obtain continuous speed, the boundary tangent vectors cannot be computed directly using Equation (19) because $\boldsymbol{T}_0^i$ and $\boldsymbol{T}_{n-1}^o$ are not inputs to the spline construction. Instead, use

$$\widehat{\boldsymbol{T}}_0^o = \lambda_0\boldsymbol{T}_0^o, \quad \widehat{\boldsymbol{T}}_{n-1}^i = \lambda_{n-1}\boldsymbol{T}_{n-1}^i \tag{22}$$

which is equivalent to choosing $\boldsymbol{T}_0^i = \boldsymbol{T}_0^o$ and $\boldsymbol{T}_{n-1}^o = \boldsymbol{T}_{n-1}^i$ in Equation (19).

6 Implementation

A C++ implementation is provided in the GTE code, TCBSpline.h. The code listing is provided here so you can refer to it while reading the PDF.

```
// David Eberly, Geometric Tools, Redmond WA 98052
// Copyright (c) 1998-2022
// Distributed under the Boost Software License, Version 1.0.
// https://www.boost.org/LICENSE_1_0.txt
// https://www.geometrictools.com/License/Boost/LICENSE_1_0.txt
// Version: 6.0.2022.08.25

#pragma once

#include <Mathematics/Logger.h>
#include <Mathematics/ParametricCurve.h>

// Compute the tension-continuity-bias (TCB) spline for a set of key frames.
// The algorithm was invented by Kochanek and Bartels and is described in
// https://www.geometrictools.com/Documentation/KBSplines.pdf

namespace gte
{
    template <int32_t N, typename T>
    class TCBSplineCurve : public ParametricCurve<N, T>
    {
    public:
        // The inputs point[], time[], tension[], continuity[] and bias[] must
        // have the same number of elements n >= 2. If you want the speed to be
        // continuous for the entire spline, the input lambda[] must have n
        // elements that are all positive; otherwise lambda[] should have 0
        // elements. If you want to specify the outgoing tangent at time[0]
        // and the incoming tangent at time[n-1], pass nonnull pointers for
        // those parameters; otherwise, the boundary tangents are computed by
        // internally duplicating the boundary points, which effectively means
        // point[-1] = point[0] and point[n] = point[n-1].
        TCBSplineCurve(
            std::vector<Vector<N, T>> const& point,
            std::vector<T> const& time,
            std::vector<T> const& tension,
            std::vector<T> const& continuity,
            std::vector<T> const& bias,
            std::vector<T> const& lambda,
            Vector<N, T> const* firstOutTangent,
            Vector<N, T> const* lastInTangent)
            :
            ParametricCurve<N, T>(
                (point.size() >= 2 ? static_cast<int32_t>(point.size() - 1) : 0),
                time.data()),
            mPoint(point),
            mTension(tension),
            mContinuity(continuity),
            mBias(bias),
            mLambda(lambda),
            mInTangent(point.size()),
            mOutTangent(point.size()),
            mA(this->GetNumSegments()),
            mB(this->GetNumSegments()),
            mC(this->GetNumSegments()),
            mD(this->GetNumSegments())
        {
            LogAssert(
                point.size() >= 2 &&
                time.size() == point.size() &&
                tension.size() == point.size() &&
                continuity.size() == point.size() &&
                bias.size() == point.size() &&
                (lambda.size() == 0 || lambda.size() == point.size()),
                "Invalid size in TCBSpline constructor.");
```

```
        ComputeFirstTangents(firstOutTangent);
        ComputeInteriorTangents();
        ComputeLastTangents(lastInTangent);
        ComputeCoefficients();
    }

    virtual ~TCBSplineCurve() = default;


    // Member access.
    inline size_t GetNumKeyFrames() const
    {
        return mPoint.size();
    }

    inline std::vector<Vector<N, T>> const& GetPoints() const
    {
        return mPoint;
    }

    inline std::vector<T> const& GetTensions() const
    {
        return mTension;
    }

    inline std::vector<T> const& GetContinuities() const
    {
        return mContinuity;
    }

    inline std::vector<T> const& GetBiases() const
    {
        return mBias;
    }

    inline std::vector<T> const& GetLambdas() const
    {
        return mLambda;
    }

    inline std::vector<Vector<N, T>> const& GetInTangents() const
    {
        return mInTangent;
    }

    inline std::vector<Vector<N, T>> const& GetOutTangents() const
    {
        return mOutTangent;
    }

    // Evaluation of the curve. It is required that order <= 3, which
    // allows computing derivatives through order 3. If you want only the
    // position, pass in order of 0. If you want the position and first
    // derivative, pass in order of 1, and so on. The output array 'jet'
    // must have enough storage to support the specified order. The values
    // are ordered as: position, first derivative, second derivative, and
    // so on.
    virtual void Evaluate(T t, uint32_t order, Vector<N, T>* jet) const override
    {
        size_t key = 0;
        T u = static_cast<T>(0);
        GetKeyInfo(t, key, u);

        // Compute the position.
        jet[0] = mA[key] + u * (mB[key] + u * (mC[key] + u * mD[key]));
        if (order >= 1)
        {
            // Compute the first-order derivative.
            T delta = this->mTime[key + 1] - this->mTime[key];
            jet[1] = mB[key] + u * (static_cast<T>(2) * mC[key] + (static_cast<T>(3) * u) * mD[key]);
            jet[1] /= delta;
            if (order >= 2)
```

```
            {
                // Compute the second-order derivative.
                T deltaSqr = delta * delta;
                jet[2] = static_cast<T>(2) * mC[key] + (static_cast<T>(6) * u) * mD[key];
                jet[2] /= deltaSqr;
                if (order == 3)
                {
                    T deltaCub = deltaSqr * delta;
                    jet[3] = static_cast<T>(6) * mD[key];
                    jet[3] /= deltaCub;
                }
            }
        }
    }

protected:
    // Support for construction.
    void ComputeFirstTangents(Vector<N, T> const* firstOutTangent)
    {
        if (firstOutTangent != nullptr)
        {
            mOutTangent[0] = *firstOutTangent;
        }
        else
        {
            T omT = static_cast<T>(1) - mTension[0];
            T omC = static_cast<T>(1) - mContinuity[0];
            T omB = static_cast<T>(1) - mBias[0];
            T twoDelta = static_cast<T>(2) * (this->mTime[1] - this->mTime[0]);
            T coeff = omT * omC * omB / twoDelta;
            mOutTangent[0] = coeff * (mPoint[1] - mPoint[0]);
        }

        if (mLambda.size() > 0)
        {
            mOutTangent[0] *= mLambda[0];
        }

        mInTangent[0] = mOutTangent[0];
    }

    void ComputeLastTangents(Vector<N, T> const* lastInTangent)
    {
        size_t const nm1 = mPoint.size() - 1;
        if (lastInTangent != nullptr)
        {
            mInTangent[nm1] = *lastInTangent;
        }
        else
        {
            size_t const nm2 = nm1 - 1;
            T omT = static_cast<T>(1) - mTension[nm1];
            T omC = static_cast<T>(1) - mContinuity[nm1];
            T opB = static_cast<T>(1) + mBias[nm1];
            T twoDelta = static_cast<T>(2) * (this->mTime[nm1] - this->mTime[nm2]);
            T coeff = omT * omC * opB / twoDelta;
            mInTangent[nm1] = coeff * (mPoint[nm1] - mPoint[nm2]);
        }

        if (mLambda.size() > 0)
        {
            mInTangent[nm1] *= mLambda[nm1];
        }

        mOutTangent[nm1] = mInTangent[nm1];
    }

    void ComputeInteriorTangents()
    {
        size_t const n = mPoint.size();
        for (size_t km1 = 0, k = 1, kp1 = 2; kp1 < n; km1 = k, k = kp1++)
        {
```

```
            Vector<N, T> const& P0 = mPoint[km1];
            Vector<N, T> const& P1 = mPoint[k];
            Vector<N, T> const& P2 = mPoint[kp1];
            Vector<N, T> P1mP0 = P1 - P0;
            Vector<N, T> P2mP1 = P2 - P1;
            T omT = static_cast<T>(1) - mTension[k];
            T omC = static_cast<T>(1) - mContinuity[k];
            T opC = static_cast<T>(1) + mContinuity[k];
            T omB = static_cast<T>(1) - mBias[k];
            T opB = static_cast<T>(1) + mBias[k];
            T twoDelta0 = static_cast<T>(2) * (this->mTime[k] - this->mTime[km1]);
            T twoDelta1 = static_cast<T>(2) * (this->mTime[kp1] - this->mTime[k]);
            T inCoeff0 = omT * omC * opB / twoDelta0;
            T inCoeff1 = omT * opC * omB / twoDelta1;
            T outCoeff0 = omT * opC * opB / twoDelta0;
            T outCoeff1 = omT * omC * omB / twoDelta1;
            mInTangent[k] = inCoeff0 * P1mP0 + inCoeff1 * P2mP1;
            mOutTangent[k] = outCoeff0 * P1mP0 + outCoeff1 * P2mP1;
        }

        if (mLambda.size() > 0)
        {
            for (size_t k = 1, kp1 = 2; kp1 < n; k = kp1++)
            {
                T inLength = Length(mInTangent[k]);
                T outLength = Length(mOutTangent[k]);
                T common = static_cast<T>(2) * mLambda[k] / (inLength + outLength);
                T inCoeff = outLength * common;
                T outCoeff = inLength * common;
                mInTangent[k] *= inCoeff;
                mOutTangent[k] *= outCoeff;
            }
        }
    }

    void ComputeCoefficients()
    {
        for (size_t k = 0, kp1 = 1; kp1 < mPoint.size(); k = kp1++)
        {
            auto const& P0 = mPoint[k];
            auto const& P1 = mPoint[kp1];
            auto const& TOut0 = mOutTangent[k];
            auto const& TIn1 = mInTangent[kp1];
            Vector<N, T> P1mP0 = P1 - P0;
            T delta = this->mTime[kp1] - this->mTime[k];
            mA[k] = P0;
            mB[k] = delta * TOut0;
            mC[k] = static_cast<T>(3) * P1mP0 - delta * (static_cast<T>(2) * TOut0 + TIn1);
            mD[k] = static_cast<T>(-2) * P1mP0 + delta * (TOut0 + TIn1);
        }
    }

    // Determine the index i for which time[i] <= t < time[i+1]. The
    // returned value is u is in [0,1].
    void GetKeyInfo(T const& t, size_t& key, T& u) const
    {
        auto const* time = this->mTime.data();
        if (t <= time[0])
        {
            key = 0;
            u = static_cast<T>(0);
            return;
        }

        size_t const numSegments = mA.size();
        if (t < time[numSegments])
        {
            for (size_t i = 0; i < numSegments; ++i)
            {
                if (t < time[i + 1])
                {
                    key = i;
```

```
                    u = (t - time[i]) / (time[i + 1] - time[i]);
                    return;
                }
            }
        }

        key = numSegments - 1;
        u = static_cast<T>(1);
    }

    // The constructor inputs.
    std::vector<Vector<N, T>> mPoint;
    std::vector<T> mTension, mContinuity, mBias, mLambda;

    // Tangent vectors derived from the constructor inputs.
    std::vector<Vector<N, T>> mInTangent;
    std::vector<Vector<N, T>> mOutTangent;

    // Polynomial coefficients. The mA[] are the degree 0 coefficients,
    // the mB[] are the degree 1 coefficients, the mC[] are the degree 2
    // coefficients and the mD[] are the degree 3 coefficients.
    std::vector<Vector<N, T>> mA, mB, mC, mD;
};
}
```

References

[1] Doris H.U. Kochanek and Richard H. Bartels. Interpolating splines with local tension, continuity, and bias control. *ACM SIGGRAPH*, 18(3):33–41, 1984.

[2] Wolfram Research, Inc. *Mathematica 13.1.0*. Wolfram Research, Inc., Champaign, Illinois, 2022.