The Beauty of Bézier Curves

Source	YouTube Video
Duration	24 min
Generated	March 22, 2026
AI	Anthropic Claude
Topics	Bézier curve construction, De Casteljau's algorithm, Bernstein polynomials, Derivatives and calculus, Curvature analysis, Oscillating circles

Generated by VidSynth

Executive Summary

This is an educational mathematics and computer graphics video created by a game developer (Freya Holmér) explaining the theory and practical applications of Bézier curves and splines. The presentation targets developers, technical artists, and students interested in understanding how curves work in computer graphics, game development, and animation systems. The video was produced in Unity using custom tools and recorded over approximately one month, with support from Patreon backers.

The video provides comprehensive coverage of cubic Bézier curves, starting with intuitive construction through nested linear interpolations (De Casteljau's algorithm) and progressing to advanced topics including derivatives, curvature analysis, bounding box calculation, and arc length parameterization. Key technical achievements demonstrated include: how the derivative of a Bézier curve is itself a Bézier curve of lower degree; how to calculate curvature and oscillating circles; how to analytically compute tight bounding boxes using the quadratic formula on derivative roots; and why arc length requires numerical approximation since cubic Bézier curves produce elliptic integrals with no closed-form solution. Real-world applications are shown throughout, including camera animation paths, procedural track generation for the presenter's game Flowstorm, and road mesh creation using tangent/normal coordinates.

Viewers should understand that while Bézier curves have elegant mathematical properties for derivatives and geometric analysis, practical implementation requires approximation techniques for arc length and uniform-speed animation. The video encourages exploration of related topics like B-splines and other spline types in the broader "cinematic universe of splines." Those interested in implementing these techniques can reference the presenter's open-source Mathfs library and Shapes plugin for Unity, both used extensively in creating the video's visualizations.

Key Points

- Bézier curves are constructed through nested linear interpolations (lerps) using control points, with each level of nesting increasing the curve degree
- The De Casteljau algorithm and Bernstein polynomial form are two equivalent mathematical interpretations of the same Bézier curve
- The derivative of a Bézier curve is itself a Bézier curve of one degree lower, making calculus operations elegant and computationally efficient
- Curvature can be calculated using velocity and acceleration vectors via the formula: determinant(velocity, acceleration) / speed^3
- Tight bounding boxes for Bézier curves require finding local extrema by solving for the roots of the derivative using the quadratic formula
- Arc length of cubic Bézier curves cannot be expressed in closed form (it's an elliptic integral) and must be approximated numerically
- Arc length parameterization requires a lookup table to convert distance values to t-parameter values for uniform-speed animation

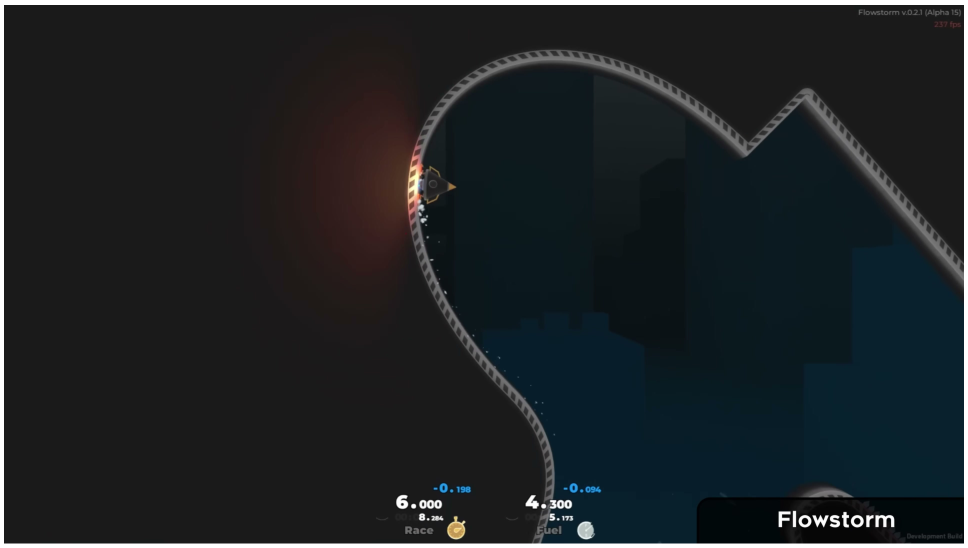
• Tangent and normal directions derived from the curve enable creation of offset points for procedural geometry like roads and tracks

• Bézier curves are ubiquitous in game development for camera paths, animation curves, level design tools, and procedural mesh generation

Detailed Breakdown

Introduction: The Power and Ubiquity of Bézier Curves

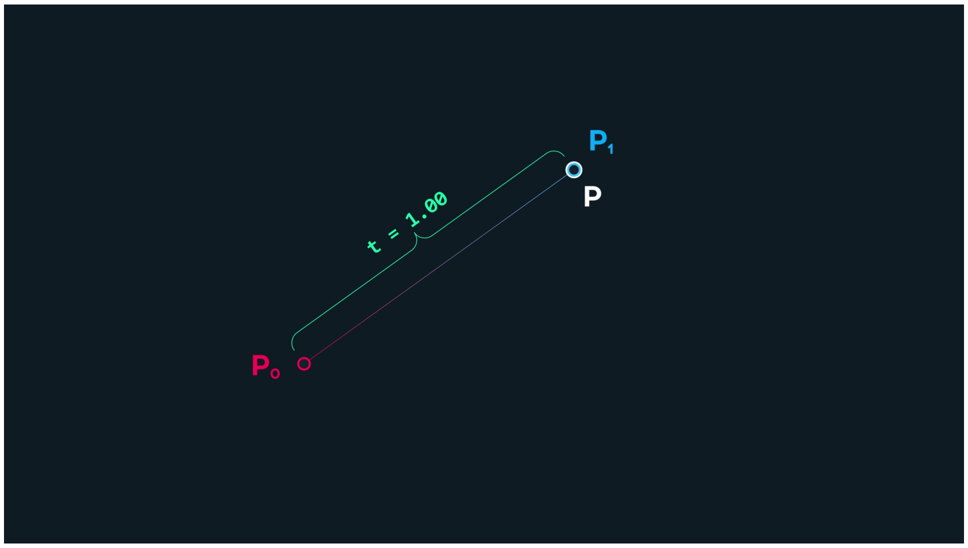
The video opens by contrasting basic mathematical curves (parabolas and circular arcs) with the more sophisticated Bézier curves and splines that can move in any direction, perform loops, and create complex shapes. The presenter emphasizes that once you understand Bézier curve construction, the concept of chaining them together into splines follows naturally. These curves are computationally elegant, allowing calculation of tangent and normal directions for orienting objects and constructing coordinate spaces around curve points—essential for procedural geometry generation. Analysis capabilities include calculating radius of curvature, determining turn direction and rate, and analytically evaluating bounding boxes for optimization in intersection testing. The introduction establishes that Bézier curves and splines are ubiquitous in games: camera animation paths, 3D model generation (like rollercoaster tracks that can be manipulated by bending and twisting), and level design tools. The presenter's own game, Flowstorm, uses Bézier curves for smooth curved tracks with an editor that exposes control handles similar to Photoshop's pen tool. This establishes both the theoretical beauty and practical necessity of understanding these mathematical constructs.



01:39 — Game or simulation interface showing a heart-shaped track with two players (Race and Elie1) at different positions with scores and statistics displayed at the bottom.

De Casteljau's Algorithm: Building Curves from Lerps

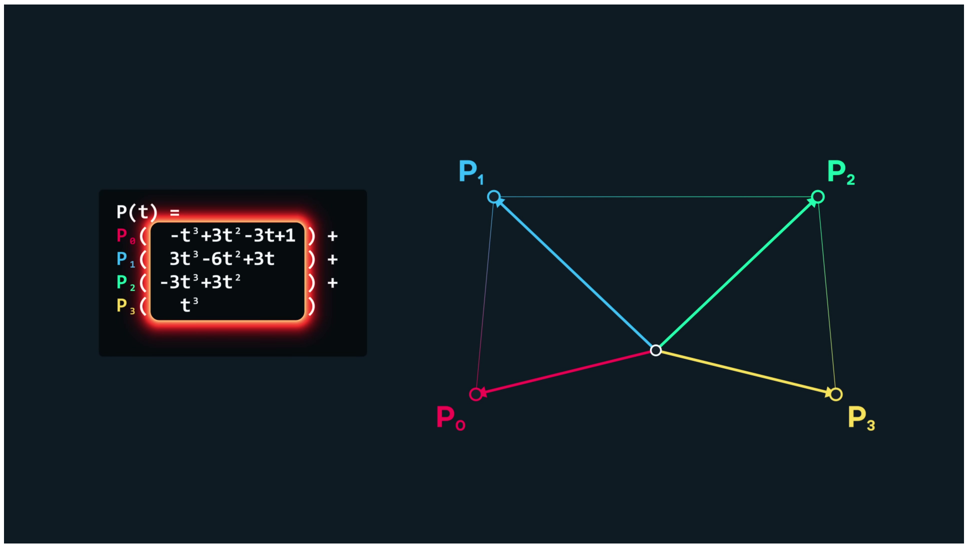
The fundamental construction of Bézier curves begins with two points P0 and P1 connected by a line segment. A third point P moves between them controlled by a t-value from 0 to 1, where t=0 places it at P0, t=1 at P1, and intermediate values create a blend. This function is linear interpolation (lerp), mathematically expressed as (1-t)×P0 + t×P1. Adding a third control point creates two line segments, each with their own interpolated point moving based on the same t-value. These two lerped points connect with another line segment, and a point lerping along this final segment traces a quadratic Bézier curve. The pattern extends: adding a fourth control point creates three initial segments, producing three lerped points, then two more lerped points, and finally one point that traces a cubic Bézier curve. This construction method—nested lerps where each point is calculated from lerps of previous points—is called De Casteljau's algorithm. The presenter emphasizes its beauty through numerical stability and memorability: "it's just lerps all the way down." The algorithm works universally regardless of control point positions, always producing smooth paths by following the same recursive rules. This visual and intuitive approach provides the foundation for understanding more abstract mathematical representations that follow.



02:16 — A technical diagram showing a quadratic Bézier curve with three control points (P■, P, P■) and parametric notation t = 1.00, illustrating curve construction geometry.

Bernstein Polynomial Form: An Alternate Mathematical Interpretation

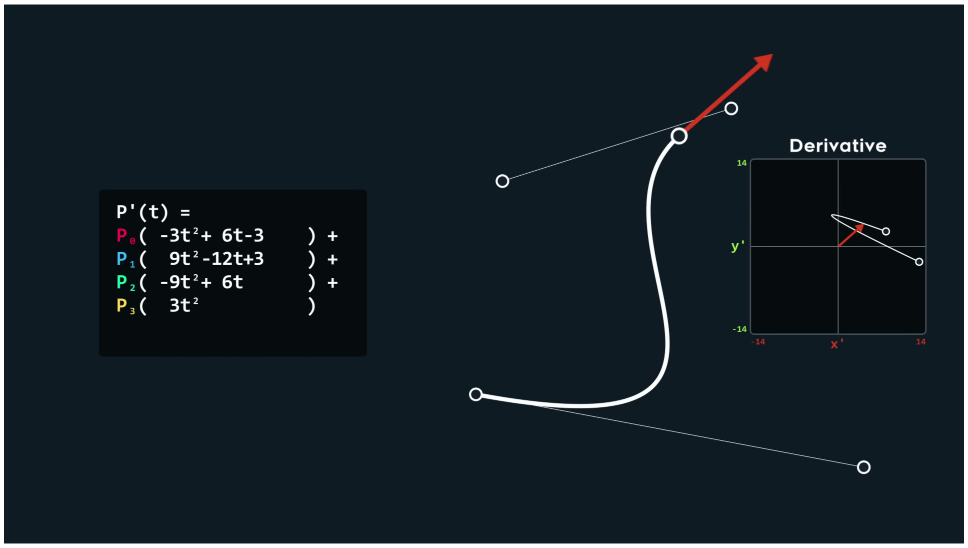
While De Casteljau's algorithm provides intuitive construction, the Bernstein polynomial form offers an algebraic perspective. By writing out all the lerp formulas for a cubic Bézier and expanding them completely, the expression can be rearranged in terms of each control point rather than nested t-values. Each control point can be visualized as a vector from the origin, and each is multiplied by one of four polynomials based on the t-value. When these four polynomial functions are graphed, they show fascinating behavior: the weights "trade off" with each other as t changes, always summing to 1 at any given t-value (a weighted sum property). Initially, the first weight equals one while others are zero; as t increases, values shift across the polynomials until the last weight reaches one. Applying these weights to the control point vectors and summing them produces exactly the same curve as De Casteljau's algorithm, but represents a different mathematical interpretation. This formulation isolates the polynomials from the control points, treating points as constants. This separation becomes crucial because it enables straightforward derivative calculation—you can differentiate the polynomials while leaving the control points unchanged, unlocking powerful analytical capabilities for curve analysis and manipulation.



05:02 — Educational diagram showing Bernstein polynomial basis functions with mathematical formula and geometric visualization of control points P0-P3 forming a Bézier curve structure.

First Derivative: Velocity, Tangents, and Coordinate Systems

The derivative of a Bézier curve represents the rate of change or velocity vector at any given t-value. A remarkable property emerges: the derivative of a cubic Bézier curve is itself a quadratic Bézier curve. This relationship holds universally—the derivative of any Bézier curve is another Bézier curve of one degree lower. The derivative changes dynamically as the curve shape changes, always remaining tangent to the curve. Normalizing this derivative vector yields the tangent direction, and rotating it 90 degrees provides the normal direction. These directional vectors enable construction of local coordinate systems offset from the curve, which is fundamental for generating procedural geometry like 3D roads. The visual demonstration shows how tangent and normal directions follow the curve's path, maintaining perpendicularity and providing a consistent reference frame at every point. This capability transforms a simple curve into a powerful tool for spatial organization and geometry generation. The derivative's interpretation as velocity also foreshadows discussions of arc length and uniform-speed animation, since varying derivative magnitudes indicate varying speeds along the curve—a property that becomes problematic when trying to animate objects at constant velocities.

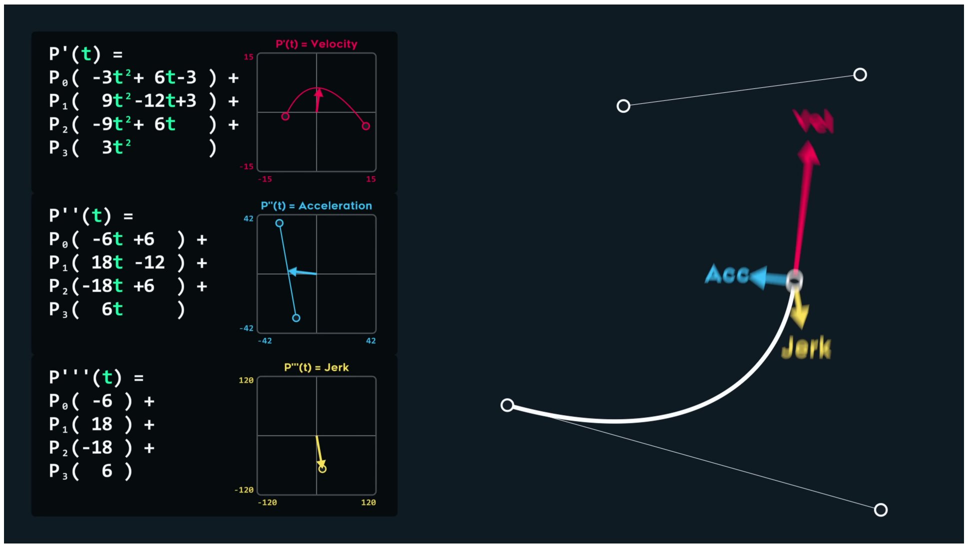


$$\begin{aligned} P'(t) = \; & P_0(-3t^2 + 6t - 3) + \\ & P_1(9t^2 - 12t + 3) + \\ & P_2(-9t^2 + 6t) + \\ & P_3(3t^2) \end{aligned}$$

07:09 — Mathematical visualization showing a derivative formula P'(t) with its component terms alongside a parametric curve with tangent vector illustration and derivative graph inset.

Higher Derivatives: Acceleration, Jerk, and Curvature Analysis

Taking the derivative again yields the second derivative, representing acceleration—the rate of change of velocity. For cubic Bézier curves, the velocity (first derivative) visualizes as a quadratic curve, so acceleration visualizes as a linear Bézier curve (simply a line segment or lerp). The third derivative, called jerk or jolt, represents the rate of change of acceleration and reduces to a constant (a point in 2D space) for cubic Béziers. These successive derivatives form a chain where each is one degree lower than the previous. The practical power emerges when combining velocity and acceleration to calculate curvature using a remarkably simple formula: determinant(velocity, acceleration) / speed^3. Curvature (denoted by kappa κ) quantifies how bent the curve is at any point—zero curvature indicates flatness, while larger absolute values indicate tighter bending. The curvature unit can be interpreted as radians per meter or, more intuitively, as reciprocal radius. This reciprocal relationship enables calculation of the oscillating circle—a circle whose curvature matches the curve at that point. The demonstration shows this circle dynamically adjusting size as it moves along the curve. Curvature can be positive or negative depending on turn direction, and when it passes through zero, that point is called an inflection point where the curve changes from bending one way to bending the other, making the oscillating circle undefined.



$$P'(t) = P_0(-3t^2+6t-3) + P_1(9t^2-12t+3) + P_2(-9t^2+6t) + P_3(3t^2)$$

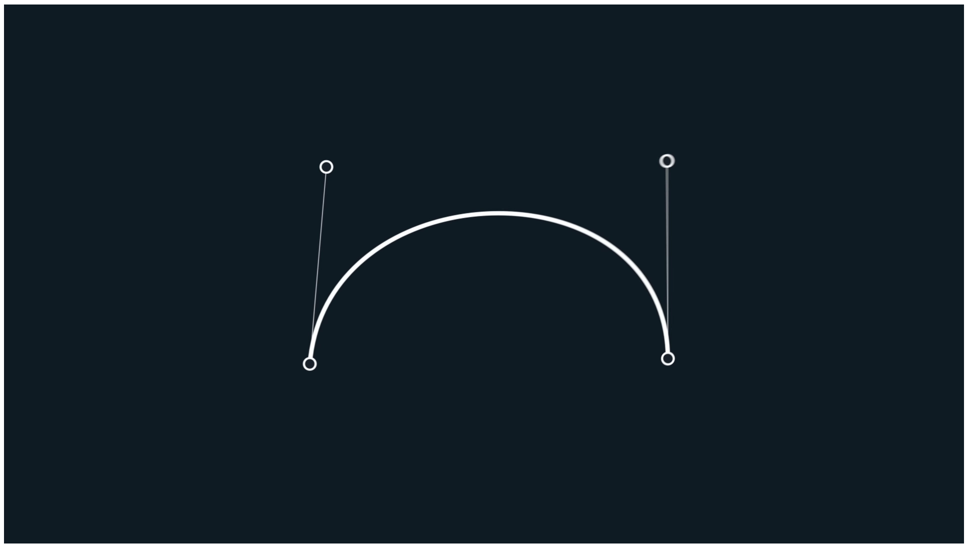
$$P''(t) = P_0(-6t+6) + P_1(18t-12) + P_2(-18t+6) + P_3(6t)$$

$$P'''(t) = P_0(-6) + P_1(18) + P_2(-18) + P_3(6)$$

09:05 — Educational diagram showing the mathematical relationship between position derivatives (velocity, acceleration, jerk) with corresponding graphs and a visual representation of motion along a curved path.

Bounding Box Calculation: Finding Extrema Analytically

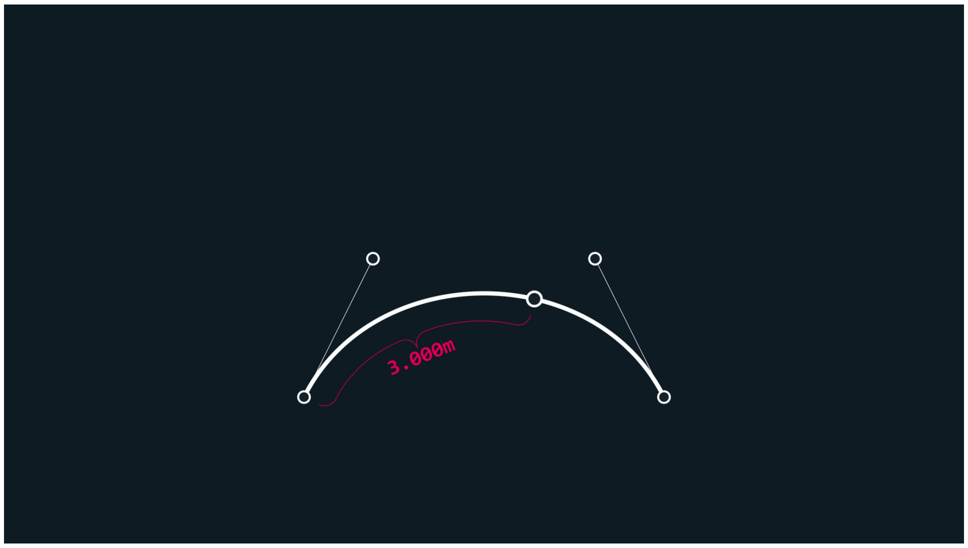
Bounding boxes are critical for optimization in game development, used for culling and preliminary intersection testing before expensive curve intersection calculations. The naive approach uses the control points to define bounds, but this can be dramatically oversized. The tight bounding box requires finding where the curve actually reaches its minimum and maximum x and y coordinates. Observing the curve reveals the bounding box touches specific points that correspond to local minima and maxima in the curve's component functions. When the curve is split into separate x(t) and y(t) functions and graphed independently, these extrema become visible. Crucially, local extrema occur where derivatives change sign—meaning where derivatives equal zero (their roots). For cubic Béziers, the derivative is quadratic, making root-finding straightforward using the quadratic formula. The derivative formula, currently expressed in terms of control points, can be rearranged into standard quadratic form ($at^2 + bt + c$) for both x and y components separately. Applying the quadratic formula yields up to four potential t-values (two from x, two from y). Some must be discarded: those outside the 0-1 range are beyond the curve segment, and those requiring square roots of negative numbers are imaginary. The remaining valid t-values identify points to include in bounds calculations, producing the tight bounding box analytically.



11:10 — A technical diagram showing a curved arc with four control points marked by circles connected by lines, illustrating a Bezier curve or similar mathematical curve construction.

Arc Length Problem: The Elliptic Integral Challenge

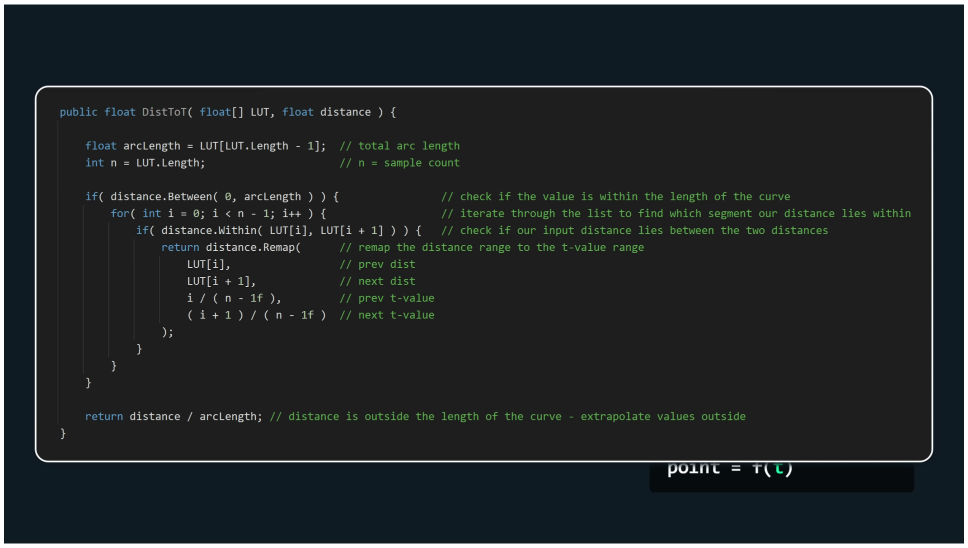
While t-values provide a simple 0-to-1 parameterization of the curve, they don't correspond to actual distances traveled along the curve. The question "how long is this curve?" leads to a fundamental limitation: there is no closed-form solution for the arc length of a cubic Bézier curve. The arc length integral is an elliptic integral, which the presenter colorfully describes as "sadness and despair." This necessitates approximation methods. One approach subdivides the curve into line segments and sums their lengths—more segments yield better accuracy but higher computational cost. The mathematical expression using summation notation is shown for completeness. Once approximate arc length is obtained, converting a distance value to a t-value seems simple: divide distance by total arc length. However, this produces incorrect results when control points change because equally-spaced t-values are not equally-spaced by distance along Bézier curves. The demonstration with dots shows them bunching and spreading as the curve shape changes. The root cause lies in the derivative's magnitude (the velocity): if velocity is constant, t-values correspond to equal distances, but Bézier curves generally have non-constant velocity. Visualizing the derivative as staying within a circle indicates near-constant speed for one particular curve, but other shapes show dramatic velocity variations, causing dots to cluster in high-speed regions and spread in low-speed regions.



15:20 — A technical diagram showing a curve with control points and a measurement of 3.000m, illustrating line segment approximation for arc length calculation.

Arc Length Parameterization: Lookup Table Solution

Since no closed-form solution exists, arc length parameterization requires another approximation: a lookup table. The curve is subdivided, and for each sampled t-value, the arc length (distance along the curve to that point) is calculated and stored in the table. More samples improve accuracy but increase computational cost. This table enables reverse lookup: given a desired distance value, the system searches the table for neighboring distance samples and interpolates to find the corresponding t-value. This effectively approximates finding the root of what would be a complicated polynomial. The presenter shares actual code used in practice: a list of distance values that returns a normalized position (0-1) within the list, which serves as the t-value. The transformation from equally-spaced t-values to equally-spaced distance values is demonstrated visually. Initially, dots placed at uniform t-intervals show uneven spacing that changes with curve shape. After implementing the lookup table to convert uniform distance intervals to their corresponding t-values, the dots achieve uniform spacing regardless of curve shape, and animated objects move at constant speed along the curve. While the approximation is "a little messy," it's sufficient for practical applications. The presenter acknowledges that cubic Béziers "just don't have a closed form solution" but emphasizes that approximation is "usually good enough in practice" for animation and procedural generation tasks.



```
public float DistToT( float[] LUT, float distance ) {

    float arcLength = LUT[LUT.Length - 1];  // total arc length
    int n = LUT.Length;                     // n = sample count

    if( distance.Between( 0, arcLength ) ) {                // check if the value is within the length of the curve
        for( int i = 0; i < n - 1; i++ ) {                  // iterate through the list to find which segment our distance lies within
            if( distance.Within( LUT[i], LUT[i + 1] ) ) {   // check if our input distance lies between the two distances
                return distance.Remap(      // remap the distance range to the t-value range
                    LUT[i],                 // prev dist
                    LUT[i + 1],             // next dist
                    i / ( n - 1f ),         // prev t-value
                    ( i + 1 ) / ( n - 1f )  // next t-value
                );
            }
        }
    }

    return distance / arcLength; // distance is outside the length of the curve - extrapolate values outside
}
```

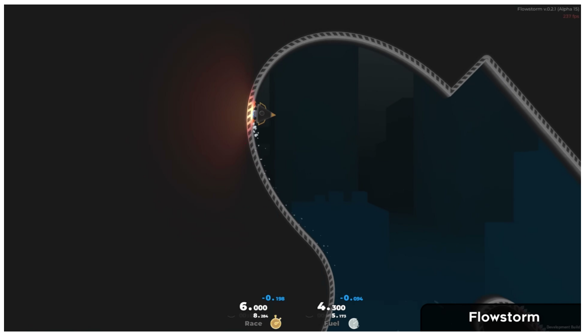
18:24 — Code implementation showing a DistToT function that converts distance along a curve to a t-value parameter using a lookup table (LUT) with linear interpolation between segments.

Conclusion and Broader Context

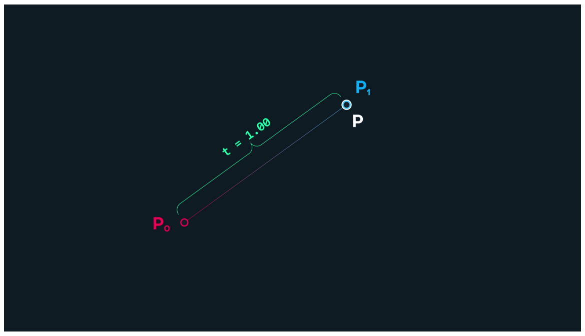
The presenter reflects on the duality of Bézier curves: they're "sometimes a little messy" due to challenges like arc length approximation, yet remain "beautiful despite their flaws" because of their elegant construction and flexible applications. Perfect precision isn't necessary—approximate arc length suffices for practical tasks like animating cameras along scenic paths. The video even demonstrates using 3D Bézier curves in RGB color space for blending point colors along the curve shown in the polynomial section. The presenter acknowledges Bézier curves aren't alone in the landscape, referencing "a whole cinematic universe of splines" including B-splines and others not covered in this video. The informal closing reveals production details: this was the longest video project undertaken, taking about a month of dedicated work, made possible by Patreon support. The technical stack included Unity for all animations, a custom timeline tool for arranging procedural animations, the presenter's Mathfs library for mathematical functions, a custom frame recorder for PNG sequence export, and Davinci Resolve for final editing. Vector graphics used the Shapes plugin for Unity. The presenter expresses openness to creating more mathematical content if the audience shows interest, while acknowledging uncertainty about audience preferences given the channel's diverse content history spanning game development, D&D campaigns, and project showcases.

Screenshots

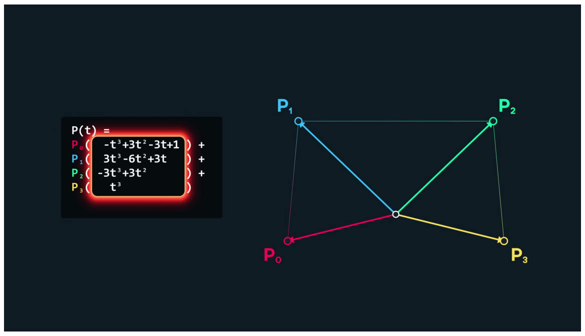
00:18 — A minimalist line diagram showing a curved line with circular elements on a dark background, illustrating basic curve concepts.



01:39 — Game or simulation interface showing a heart-shaped track with two players (Race and Elie1) at different positions with scores and statistics displayed at the bottom.

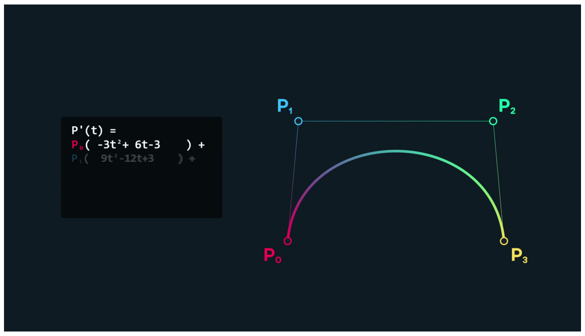


02:16 — A technical diagram showing a quadratic Bézier curve with three control points (P■, P, P■) and parametric notation t = 1.00, illustrating curve construction geometry.



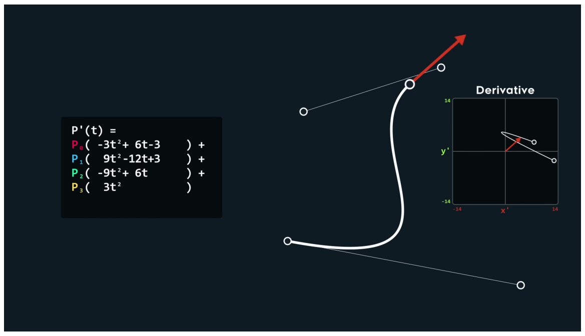
$$P(t) = P_0(-t^3+3t^2-3t+1) + P_1(3t^3-6t^2+3t) + P_2(-3t^3+3t^2) + P_3(t^3)$$

05:02 — Educational diagram showing Bernstein polynomial basis functions with mathematical formula and geometric visualization of control points P0-P3 forming a Bézier curve structure.



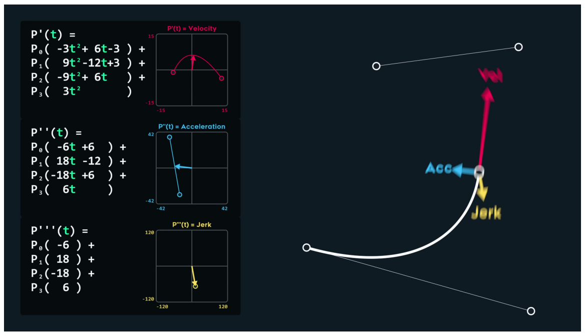
$$P'(t) = P_0(-3t^2+6t-3) + P_1(9t^2-12t+3) +$$

06:30 — Mathematical diagram showing a cubic Bézier curve with four control points (P0-P3) and the derivative formula P'(t) displayed, illustrating the relationship between the curve geometry and its mathematical representation.



$$P'(t) = P_0(-3t^2+6t-3) + P_1(9t^2-12t+3) + P_2(-9t^2+6t) + P_3(3t^2)$$

07:09 — Mathematical visualization showing a derivative formula P'(t) with its component terms alongside a parametric curve with tangent vector illustration and derivative graph inset.

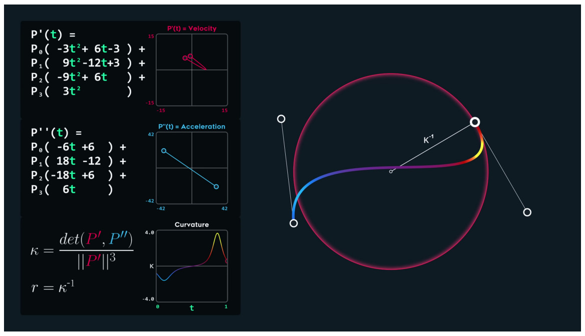


$$P'(t) = P_0(-3t^2+6t-3) + P_1(9t^2-12t+3) + P_2(-9t^2+6t) + P_3(3t^2)$$

$$P''(t) = P_0(-6t+6) + P_1(18t-12) + P_2(-18t+6) + P_3(6t)$$

$$P'''(t) = P_0(-6) + P_1(18) + P_2(-18) + P_3(6)$$

09:05 — Educational diagram showing the mathematical relationship between position derivatives (velocity, acceleration, jerk) with corresponding graphs and a visual representation of motion along a curved path.



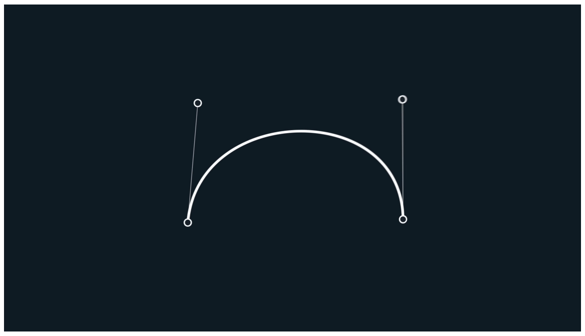
$$P'(t) = P_0(-3t^2+6t-3) + P_1(9t^2-12t+3) + P_2(-9t^2+6t) + P_3(3t^2)$$

$$P''(t) = P_0(-6t+6) + P_1(18t-12) + P_2(-18t+6) + P_3(6t)$$

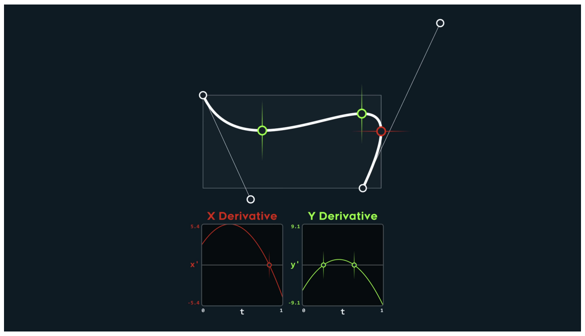
$$\kappa = \frac{det(P', P'')}{||P'||^3}$$

$$r = \kappa^{-1}$$

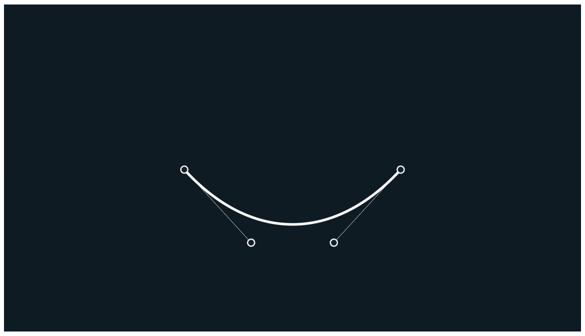
10:47 — Comprehensive visualization showing parametric curve equations with corresponding velocity, acceleration, and curvature graphs, plus geometric representation of the curve with marked points.



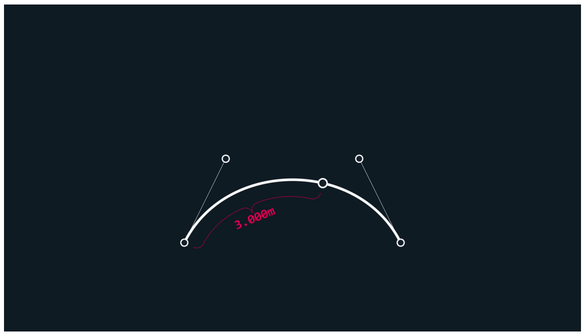
11:10 — A technical diagram showing a curved arc with four control points marked by circles connected by lines, illustrating a Bezier curve or similar mathematical curve construction.



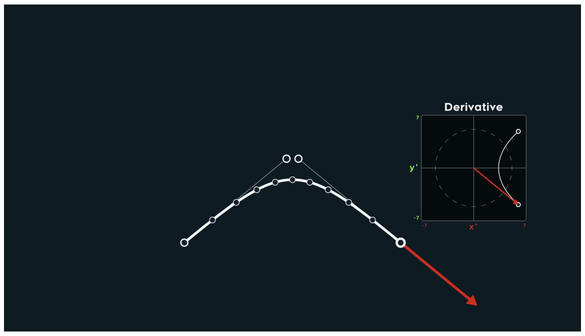
12:32 — A comprehensive Bézier curve diagram showing the relationship between curve control points and their X and Y derivatives, with clear visual representations of how derivatives relate to curve behavior.



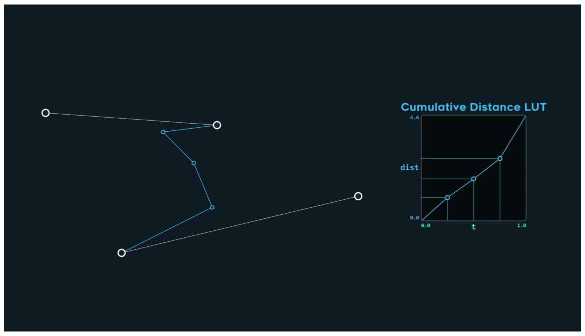
14:24 — A mathematical diagram showing a curved arc (ellipse segment) with four control points marked by circles, illustrating an arc length problem involving elliptic integrals.



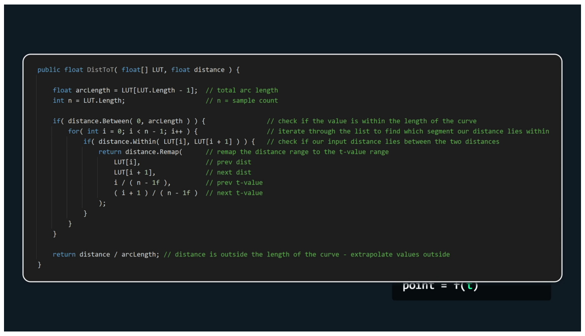
15:20 — A technical diagram showing a curve with control points and a measurement of 3.000m, illustrating line segment approximation for arc length calculation.



16:44 — Educational diagram showing the relationship between a function curve and its derivative representation with tangent line visualization in a coordinate system.



17:31 — A technical diagram showing a curved path with nodes and a corresponding 3D cumulative distance lookup table (LUT) visualization with labeled axes.



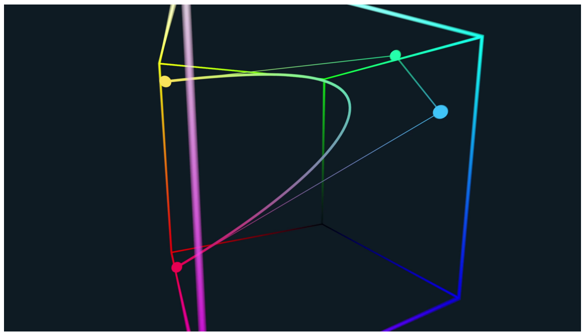
```
public float DistToT( float[] LUT, float distance ) {

    float arcLength = LUT[LUT.Length - 1];  // total arc length
    int n = LUT.Length;                     // n = sample count

    if( distance.Between( 0, arcLength ) ) {                // check if the value is within the length of the curve
        for( int i = 0; i < n - 1; i++ ) {                  // iterate through the list to find which segment our distance lies within
            if( distance.Within( LUT[i], LUT[i + 1] ) ) {   // check if our input distance lies between the two distances
                return distance.Remap(      // remap the distance range to the t-value range
                    LUT[i],                 // prev dist
                    LUT[i + 1],             // next dist
                    i / ( n - 1f ),         // prev t-value
                    ( i + 1 ) / ( n - 1f )  // next t-value
                );
            }
        }
    }

    return distance / arcLength; // distance is outside the length of the curve - extrapolate values outside
}
```

18:24 — Code implementation showing a DistToT function that converts distance along a curve to a t-value parameter using a lookup table (LUT) with linear interpolation between segments.



19:44 — A 3D geometric diagram showing a cube with colored edges and three spherical nodes connected by lines, illustrating spatial relationships and potentially demonstrating distance or parametric values.