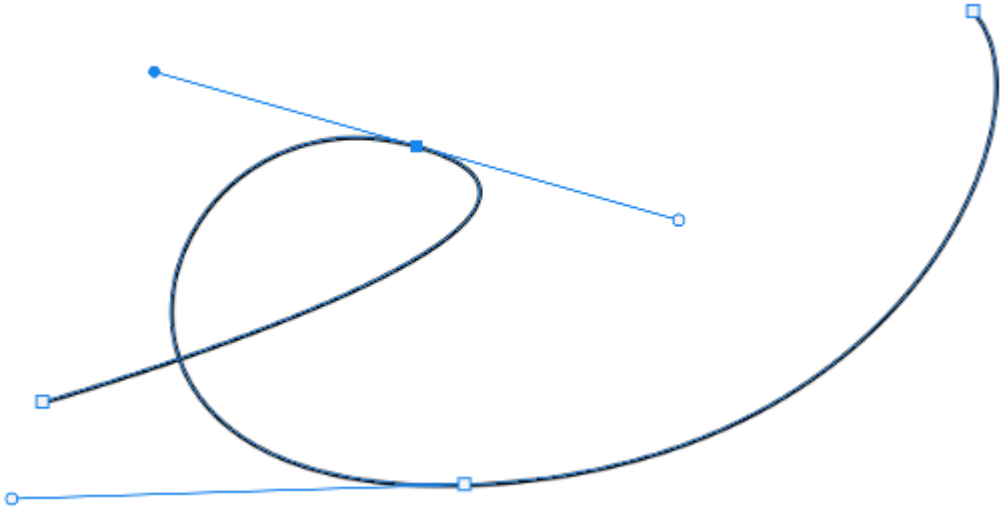
Bezier curves anatomy

Finding the smallest possible box a curve entirely fits into.

Pages: 20



Introduction

Aim - The aim of this exploration is to better understand the tools used by digital artists and designers such as myself on the daily, this will help me make better use of them in my future work. I will aim to develop a way to rigorously calculate the rectangle of the smallest area possible such as all the points in a curve fit into it, and then extend the formula into the third dimension in order to aid my understanding of 3D modelling tools – such an algorithm could then be implemented in the field of video game design, which also lies closely to my interests.

Rationale - I have chosen this topic for my exploration because I have always been interested in the digital world and have been creating art and designs using the tools I'm about to describe for years now. Although I always have had an intuitive knowledge of their inner workings, I am excited to learn about how they implement the Bezier curves rigorously. I read about the technology used to create the movie *Coco (2017)* and I would like to study how the bounding box property of a curve helped the studio deliver such stunning scenes.

Plan - I am going to analyze the mathematics behind the curves and splines created by the computer graphics programs. I will attempt to draw the rectangle and look for patterns between the curves which will help me to find a formula for the final formula. I will then attempt to model a 3D curve in 3D design software Blender and rigorously calculate its bounding box.

Breaking down the Bezier

Bezier curves, invented by the French mathematician Pierre Bezier are defined by a mathematical technique called *linear interpolation*. Linear interpolation is a method of plotting a function or a relation through some data points, here it is quite straight forward to figure out the shortest distance between two points, you just draw a straight line through them. A curve by this definition is a set of control points P_0 through P_n between which a moving point P is traced. By convention, the time taken to draw the curve is $t = 1$. Drawing a line between two points P_0 and P_1 is in principle tracing the moving point P as it moves from P_0 to P_1. The curve is then a function of time, where the coefficients of the points tell the computer about how close to the static point the moving point is at a current time. Therefore, a linear Bezier curve from P_0 to P_1 is defined as such:

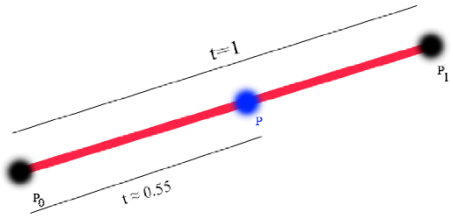


Figure 1. An annotated linear Bezier curve

$$B(t) = (1 - t)P_0 + tP_1, \qquad 0 \leq t \leq 1$$

This makes intuitive sense, as the distance of the moving point to the first static point is decreasing, and the distance to the final static point is increasing. The static points are also called the *control* points. Imagine that both control points are tugging on the point P, but the strength with which they do so varies over time and is described by the equation above. It is important to notice that the total strength of all points added up is always 1. I will get back to why that is the case when we introduce more complicated curves. This method of fitting a curve using a linear polynomial is linear interpolation, or *lerp* for short. The position of P is then:

$$P = lerp(P_0, P_1, t)$$

Figure 3. A logo mockup

A higher order type of Bezier curves in computer graphics is the Quadratic Bezier Curve, a direct expansion upon the linear model Quadratic Bezier curves are used all the time, for example when drawing the leaf of the apple in the logo mockup on the left. A quadratic Bezier curve actually curves now, and is made up of two linear Bezier curves connected at one end. Let P_0, P_1, P_2 be the three points describing the curve.

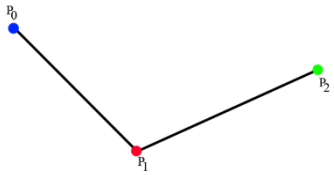


Figure 2. A quadratic Bezier curve foundation

Each of the line segments has its own moving point P_1' and P_2', the position of which is a function of the same variable t in a way analogous to the linear case.

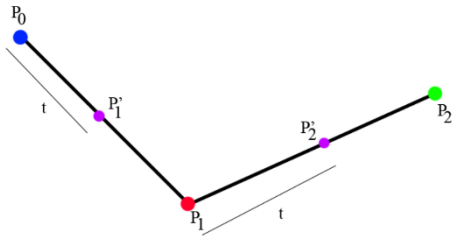


Figure 4. A quadratic Bezier curve foundation with intermediate points

Now, upon connecting the two points P_1' and P_2' using another linear Bezier curve we get the third line segment. The line segment is not static, it has its position dependent on the time t, as the points P_1' and P_2' it is connecting change their position as well.

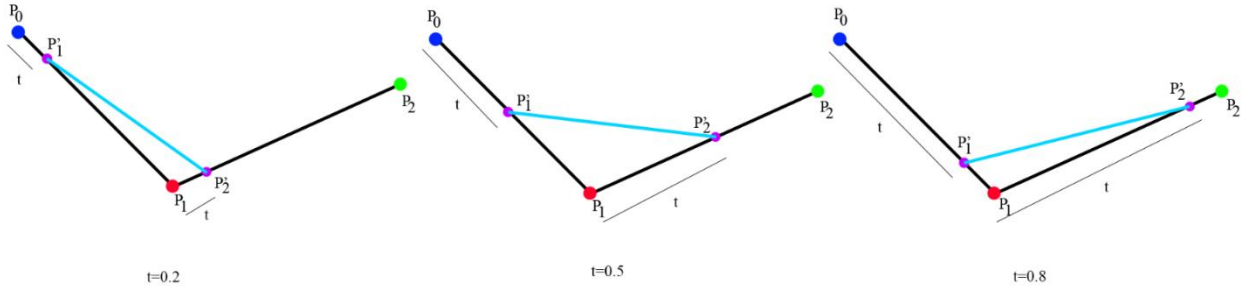


Figure 5. The line through the intermediate points at different times

We have everything we need to draw the proper Bezier curve now. It is drawn using the same principle of interpolation as points P_1' and P_2'. We define a moving point P dependent on t on the moving line segment connecting the other two moving points, which are also dependent on t. The computer then traces its position over time.

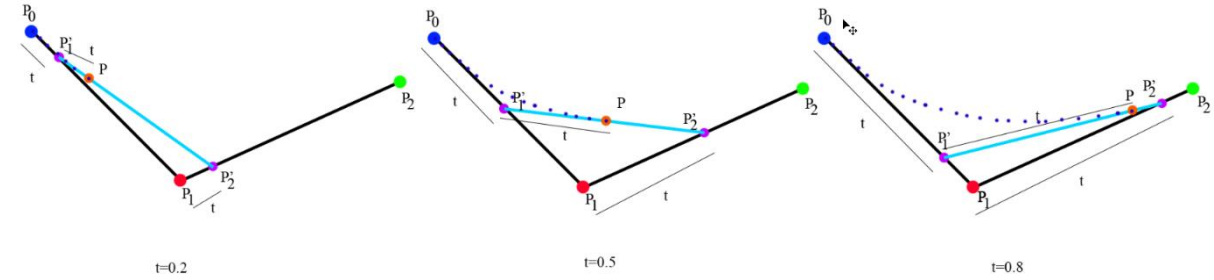


Figure 6. Construction of a quadratic Bezier curve

The result is a smooth and continuous curve described solely by the position of the first three points.

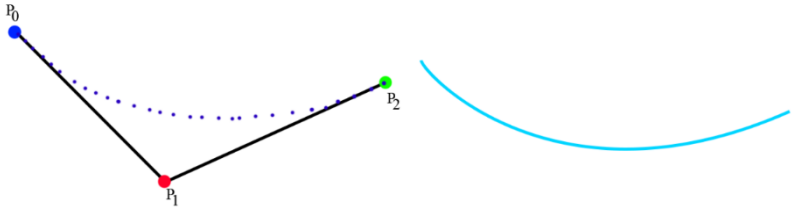


Figure 7. A finished quadratic Bezier curve

We do not need to stop here – The most used type of the Bezier curve is the cubic Bezier cube, which works the same way as the two types I've described already. It is made up of three linear bezier curves connected end-to-end and points interpolated until you get the final point describing the curve.

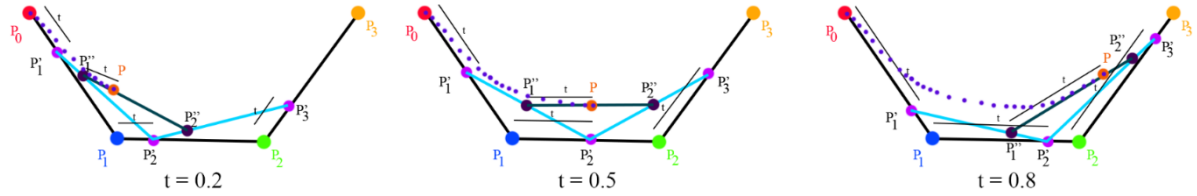


Figure 8. Construction of a cubic Bezier curve

Each point along the way to calculate the final curve is calculated from linear interpolations from the points that came before it.

$$
\begin{aligned}
P_1' &= lerp(P_0, P_1, t) \\
P_2' &= lerp(P_1, P_2, t) \\
P_3' &= lerp(P_2, P_3, t) \\
P_1'' &= lerp(P_1', P_2', t) \\
P_2'' &= lerp(P_2', P_3', t) \\
P &= lerp(P_1'', P_2'', t)
\end{aligned}
$$

We can use the linear interpolation formula we used in the simplest case to define the position of every moving point in terms of strength of the other points tugging on it. Here is what it looks like:

$$
\begin{aligned}
P_1' &= (1-t)\boldsymbol{P_0} + t\boldsymbol{P_1} \\
P_2' &= (1-t)\boldsymbol{P_1} + t\boldsymbol{P_2} \\
P_3' &= (1-t)\boldsymbol{P_2} + t\boldsymbol{P_3} \\
P_1'' &= (1-t)P_1' + tP_2' \\
P_2'' &= (1-t)P_2' + tP_3' \\
P &= (1-t)P_1'' + tP_2''
\end{aligned}
$$

After substituting each equation into the appropriate formula that came before it, we obtain a rigorous formula for the position of P over time as a function of the 4 initial control points.

$$
\begin{aligned}
P(t) = &-\boldsymbol{P_0}t^3 + 3\boldsymbol{P_0}t^2 - 3\boldsymbol{P_0}t + \boldsymbol{P_0} + \\
&3\boldsymbol{P_1}t^3 - 6\boldsymbol{P_1}t^2 + 3\boldsymbol{P_1}t + \\
&-3\boldsymbol{P_2}t^3 + 3\boldsymbol{P_2}t^2 + \\
&\boldsymbol{P_3}t^3
\end{aligned}
$$

$$
\begin{aligned}
P(t) = &\boldsymbol{P_0}(-t^3 + 3t^2 - 3t + 1) + \\
&\boldsymbol{P_1}(3t^3 - 6t^2 + 3t) + \\
&\boldsymbol{P_2}(-3t^3 + 3t^2) + \\
&\boldsymbol{P_3}(t^3)
\end{aligned}
$$

I will now use Desmos to graph the polynomials inside the brackets to visualize the impact each point has on the position of the point P in a cubic Bezier curve

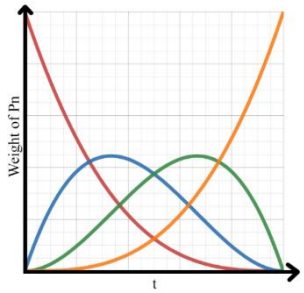


Figure 9. Weight distribution of the control points over time

We notice that the weights always add up to 1 for any given t. What happens when the weights don't add up to 1? Let's say that the green point P_2 has its weight doubled, and that the total weight is not constant. Since the green point now tugs at double the strength of the other points, the curve acts as if there were two distinct points in the same position of the green point, and the curve acts like a quartic Bezier curve. Non-integer scaling of the individual points can be approximated by higher and higher orders of the Bezier curves pretending to be a lower degree curve. For example, if one point was to have its weight at 1.5 the standard weight, the curve would act as if there were two points in place of every point and an extra point at the heaviest point, adding up to a 9th degree Bezier curve. This throws off the formulas I'm going to derive in later sections, which is why it is important to keep in mind the constant total weight/strength requirement.

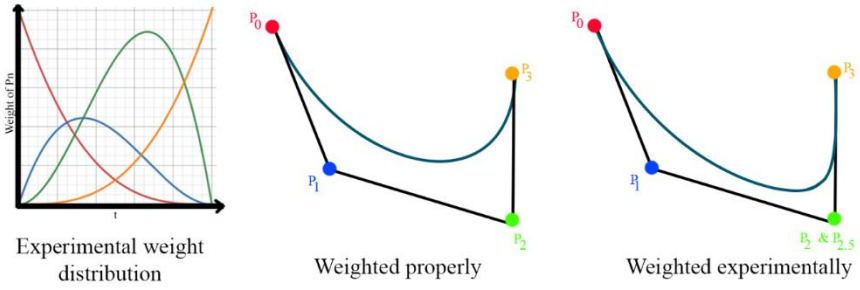


Figure 10. A proper cubic Bezier curve compared to a cubic Bezier curve with the point P2 double the weight of the other points.

Since the points P_n are constant, and the other parts of the formula are just polynomials, we can easily get the derivative of a Bezier curve.

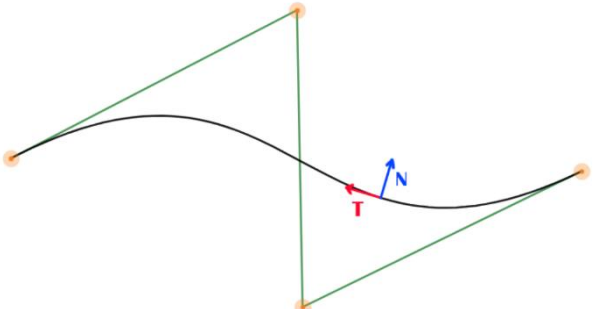


Figure 11. A cubic Bezier curve with a normal and tangent

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

Knowing a derivative is useful, as it gives us the velocity vector which is tangent to the curve at a given point, therefore providing us the tangent line to the curve at a given point. Once we have the tangent line, the normal vector can easily be obtained by rotating the velocity vector by $\frac{\pi}{2}$ in either direction, as in Figure 13. When we get the exact vectors for the tangent and normal, we can offset the curve in the two normal directions by a certain value and fill in the space created by the two offset curves, giving the whole curve a width, which is precisely what computer art programs are calculating behind the scenes when I change the width of a shape. Deriving the cubic curve again, we obtain the acceleration vector of the curve.

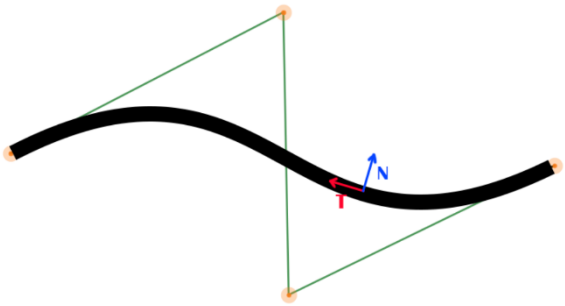


Figure 12. A cubic Bezier curve with width

The derivative of the Bezier curve can be used to precisely compute the smallest possible rectangle that contains all points on the curve – a bounding box.

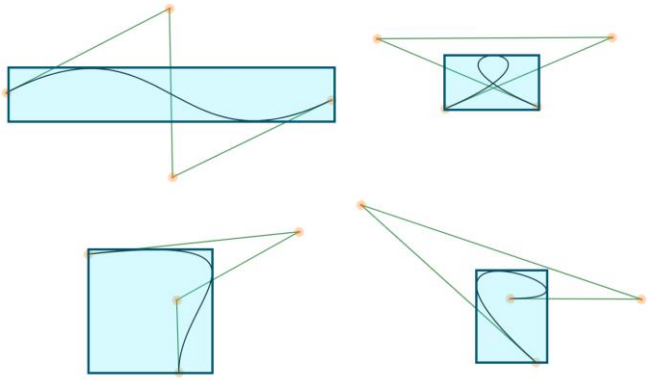


Figure 13. Examples of bounding boxes on cubic Bezier curves

Reflection

Although my initial guess was just to plot the box around the control points, as I more or less did in Figure 15. I observe that my initial train of thought is not precise enough, as in most cases that rectangle is needlessly large – the bottom left one would be unreasonably large. I derived a way to get the bounding box more precisely using the derivative. I notice that the hand-drawn box touches the cubic curve at a maximum of 2 spots other than the starting and ending control points – the local extrema. The derivative gives these extrema when solved for roots, but since my object is a curve, not a function, every axis has its own function of position over time. I will split the curve into two separate x-position and y-position curves. First, I will rearrange $P'(t)$ into a quadratic form with the points as coefficients of t.

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

Let the 4 control points of an example cubic curve be as follows:

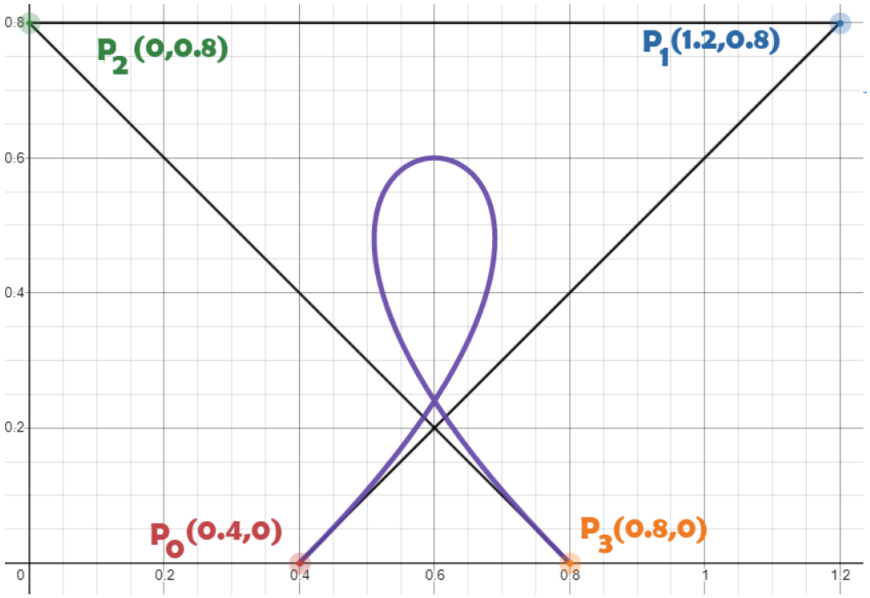


Figure 14. The example cubic Bezier curve

Splitting up the formula into two separate X and Y instances, we obtain these equations of the derivatives of the x and y positions:

$$P'_x(t) = t^2\big(-3(\mathbf{0.4}) + 9(\mathbf{1.2}) - 9(\mathbf{0}) + 3(\mathbf{0.8})\big) + t\big(6(\mathbf{0.4}) - 12(\mathbf{1.2}) + 6(\mathbf{0})\big) + \big(-3(\mathbf{0.4}) + 3(\mathbf{1.2})\big) = 12t^2 - 12t + 4.8$$

$$P'_y(t) = t^2\big(-3(\mathbf{0}) + 9(\mathbf{0.8}) - 9(\mathbf{0.8}) + 3(\mathbf{0})\big) + t\big(6(\mathbf{0}) - 12(\mathbf{0.8}) + 6(\mathbf{0.8})\big) + (-3(\mathbf{0}) + 3(\mathbf{0.8})) = -4.8t + 2.4$$

Now, to find the local extrema I find the real roots of the derivatives. If this method works, one of the roots should be exactly $t = 0.5$, as the curve I picked to test the method on is such that it is symmetrical exactly halfway through the length of the curve, and that's where the highest point of it is. For the x position function the quadratic discriminant is negative:

$$\Delta = b^2 - 4ac = 144 - 4(12)(4.8) = -86.4$$

Therefore, there are no real roots to be found here. For the y position function there is only one root, as it is linear.

$$-4.8t + 2.4 = 0 \iff t = 0.5$$

Sure enough, there is a local extremum at $t = 0.5$. I've decided to calculate the second derivative of $P_x(t)$, and found out that it amounts to $24t - 12$, which also has a root at $t = 0.5$. This is an interesting result, I think that for more higher order curves, the higher order derivatives will be important when looking for the bounding box.

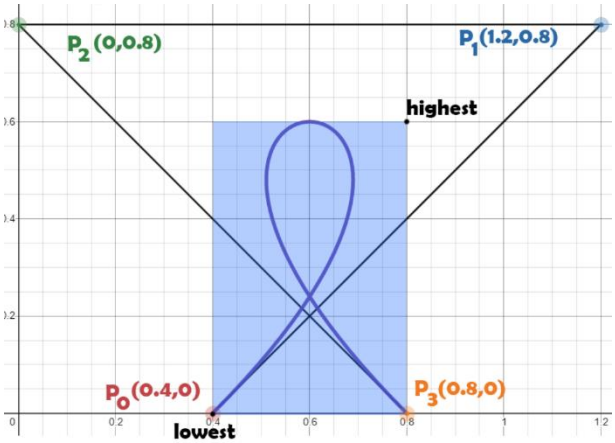


Figure 15. The example cubic Bezier curve bound by a box

Now that I have the local extrema and the two start and end static points, I compare all x values and y values of the two points and $P(t)$, where t is a local extremum and draw the rectangle around the highest and lowest found values. For my curve with the loop, the points are

$$P_0 = (0.4,0)$$
$$P_3 = (0.8,0)$$
$$P(0.5) = (0.6,0.6)$$

The lowest x value is 0.4, the highest x value is 0.8, the lowest y value is 0 and the highest y value is 0.6. I draw a box through (x_{lowest}, y_{lowest}) and $(x_{highest}, y_{highest})$. In my case its (0.4,0) through (0.8,0.6)

Reflection

This is a better result, there is a problem with this method though – This is what happens, when I rotate the curve by an angle, for example $\frac{3\pi}{4}$ around the central point $(0.6,0.4)$. I obtain the rotated curve by converting each point P_n to a complex number, subtracting $0.6 + 0.4i$ from it to center it at the origin, multiplying it by the complex number $\cos\left(-\frac{3\pi}{4}\right) + isin\left(-\frac{3\pi}{4}\right)$ and then adding the $0.6 + 0.4i$ back to it to move it where it was initially. Let P_n^* be the new point after rotation, given by the formula

$$P_n^* = \left(\left(P_{n_x} - 0.6\right) + \left(P_{n_y} - 0.4\right)i\right)\left(\cos\left(-\frac{3\pi}{4}\right) + isin\left(-\frac{3\pi}{4}\right)\right) + 0.6 + 0.4i$$

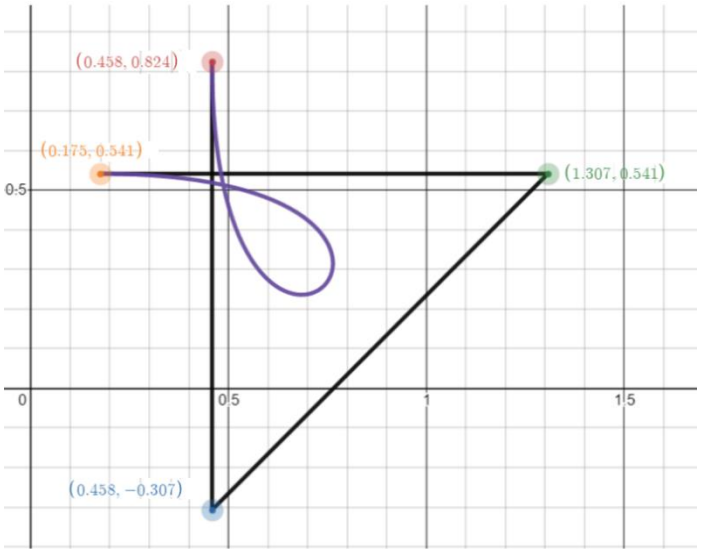


Figure 16. The rotated example cubic Bezier curve

The initial approach to finding the bounding box proves not to be the optimal way! On the left is the box that would have been drawn if I used the derivative approach on the rotated Bezier, and on the right is the true smallest bounding box.

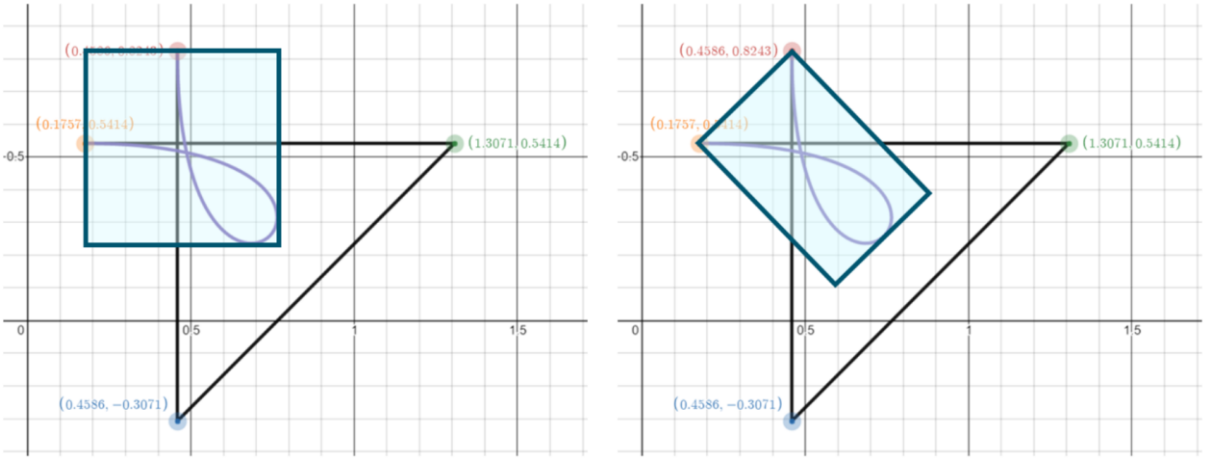


Figure 17. The rotated example cubic Bezier curve bound by a wide box, and by a tight box

Reflection

Something interesting happens here – Although these two curves are exactly the same, but rotated, their bounding box doesn't fit tightly in both cases. There appears to be a notion of having to standardize a curve first, before finding it's bounding box. The curve in *Fig. 16* seems to be standardized already, but that isn't the case for the rotated one in *Fig. 18*. The rotation is the thing that is throwing off the bounding box calculations, so I need to rotate each curve someway before trying to bound it in a box. A sensible method of standardizing the curve would be to make sure that each curve starts at the same position. The origin is a good choice for this, as although the

choice doesn't impact the rotation, choosing the origin allows to rotate the curve around the origin with complex numbers without the need to find a central point, which is less convoluted. What makes the example curve in *Fig. 18* already rotated in the "standardized" way? Since the points that our method is looking for are the start point & the end point and since our method of drawing rectangles only results in sides parallel to one of the axes, to get the tightest possible box we should rotate the curve around its start point until the end point is also situated on a line through the start point parallel to one of the axes. I will choose the x-axis for consistency, although choosing the y-axis will also yield proper results. How do we find the correct values to translate and rotate by to standardize a curve? The translation of the curve is easy, we just need to subtract (P_{0_x}, P_{0_y}) from every point value.

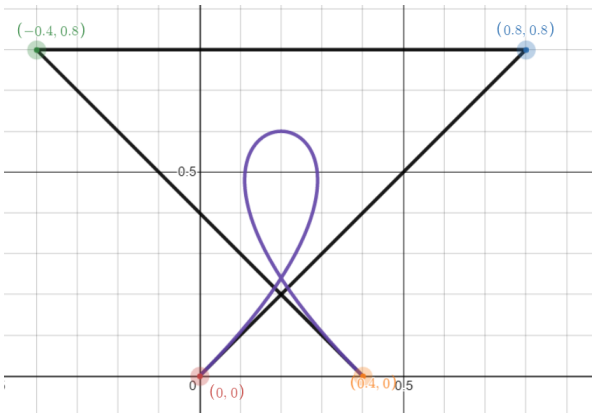


Figure 18. The example cubic Bezier curve standardized

We should now derive the method of finding the required angle of rotation - let's look at another cubic Bezier curve which is not standardized already.

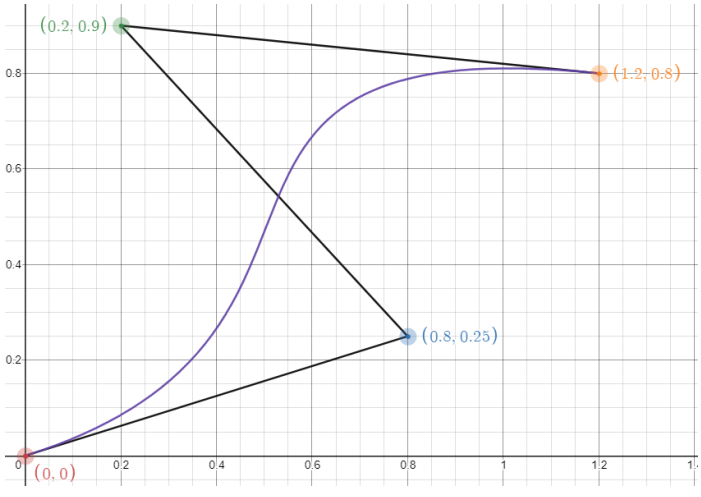


Figure 19. A translated but not rotated cubic Bezier curve

After translating, the line segment joining P_0 and P_1 may be thought of as a vector from the origin represented by $\begin{pmatrix} P_{1_x} \\ P_{1_y} \end{pmatrix}$. In our case it's $\begin{pmatrix} 0.8 \\ 0.25 \end{pmatrix}$. We need to find the angle the vector makes with the x-axis, and then rotate the whole curve by the same angle in opposite direction. We know everything we need to know about the vector to find the angle θ:

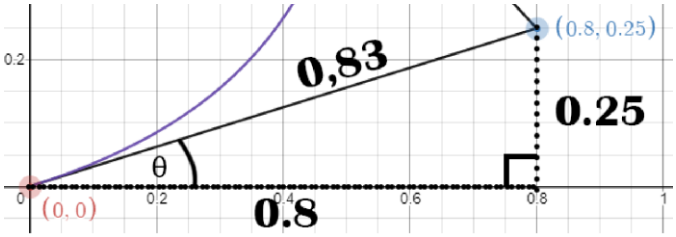


Figure 20. Close up of information given.

We use the relationship $\tan\theta = \frac{length\ of\ the\ opposite\ side}{length\ of\ the\ adjacent\ side}$ to find that the tangent of θ is

$$tan\theta = \frac{0.25}{0.8} = 0.31$$

We use arctangent to find the exact angle

$$\theta = \arctan(tan\theta) = \arctan(0.31) \approx 0.3006\, rad \approx 17.2^{\circ}$$

We need to rotate each control point on the graph by $-\theta$ employing a simplified version of the formula used before. Let P_n^* again be the rotated point. We can use the simplified version because we translated the curve to start at the origin.

$$P_n^* = \left(P_{n_x} + P_{n_y} i\right)\left(\cos(-\theta) + isin(-\theta)\right)$$

Let's standardize the curve!

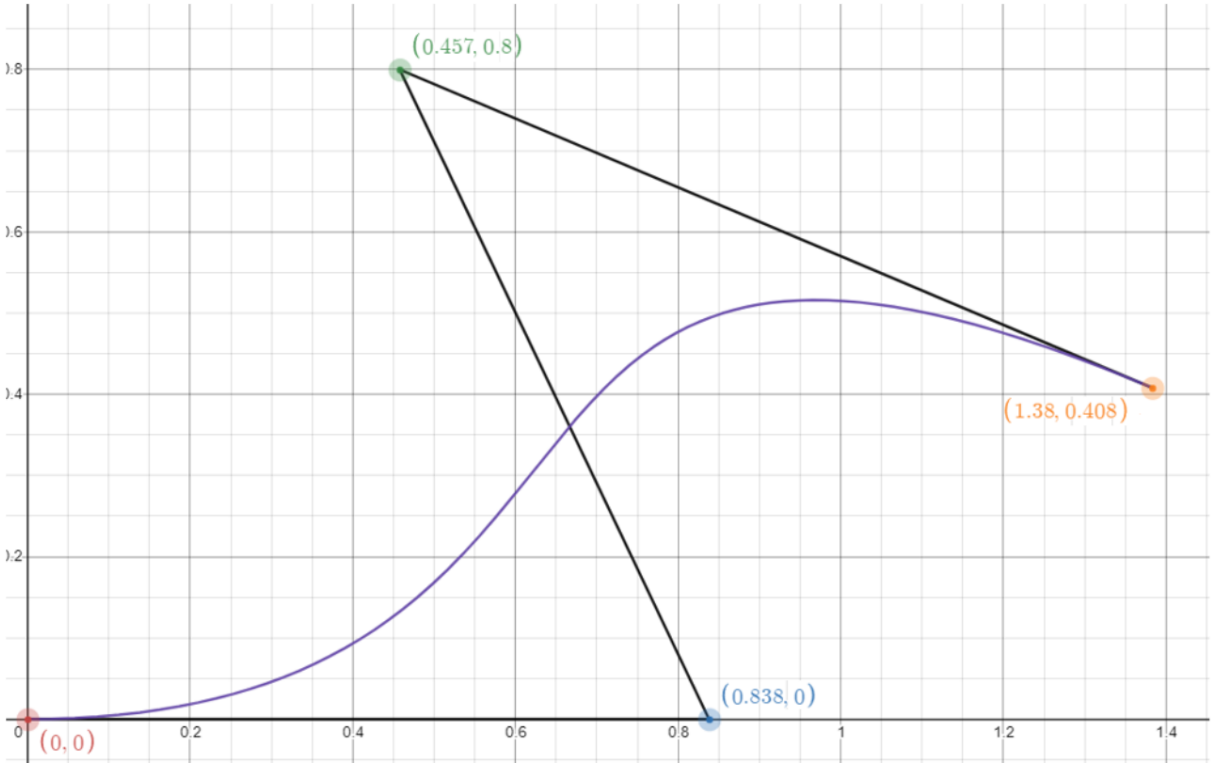


Figure 21. Standardized second example cubic Bezier

Now, we shall employ the derivative method of finding the bounding box on the non-standardized curve and compare it to the optimized strategy.

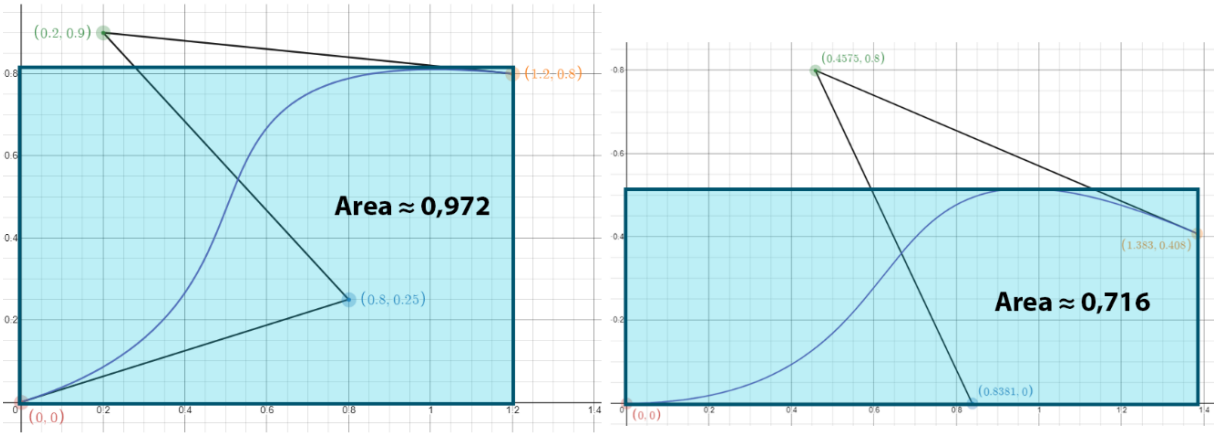


Figure 22. Comparison of the Area of the curve before and after standardization.

After setting the box on the standardized version of the curve, we can rotate it and the box again by θ to reverse the rotation and translate it back into its original position by a vector $-\begin{pmatrix} P_{0_x} \\ P_{0_x} \end{pmatrix}$

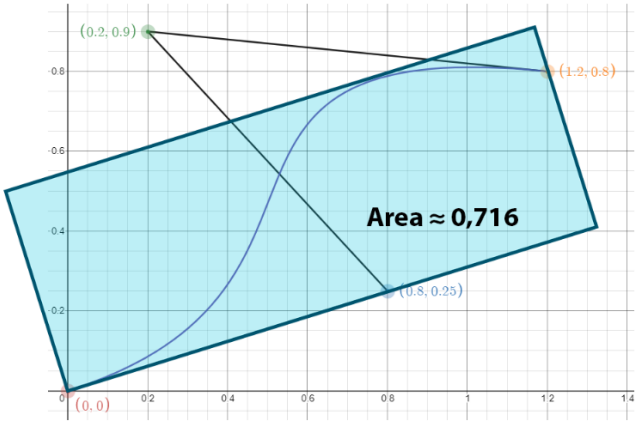


Figure 23. A succesful analytically found tight bounding box

We now combine all the knowledge to obtain a formula for the standardization of any 2D cubic Bezier cube.

$$P_n^* = \left(P_{n_x} - P_{0_x} + \left(P_{n_y} - P_{0_y}\right)i\right)\left(\cos\left(\arctan\left(-\frac{P_{1_y}}{P_{1_x}}\right)\right) + i\sin\left(\arctan\left(-\frac{P_{1_y}}{P_{1_x}}\right)\right)\right)$$

After standardizing, differentiate the curve and split it into two parts, one for each axis.

$$P^{*\prime}{}_x(t) = t^2\left(-3\boldsymbol{P}^*_{0_x} + 9\boldsymbol{P}^*_{1_x} - 9\boldsymbol{P}^*_{2_x} + 3\boldsymbol{P}^*_{3_x}\right) + t\left(6\boldsymbol{P}^*_{0_x} - 12\boldsymbol{P}^*_{1_x} + 6\boldsymbol{P}^*_{2_x}\right) + \left(-3\boldsymbol{P}^*_{0_x} + \boldsymbol{P}^*_{1_x}\right)$$

$$P^{*\prime}{}_y(t) = t^2\left(-3\boldsymbol{P}^*_{0_y} + 9\boldsymbol{P}^*_{1_y} - 9\boldsymbol{P}^*_{2_y} + 3\boldsymbol{P}^*_{3_y}\right) + t\left(6\boldsymbol{P}^*_{0_y} - 12\boldsymbol{P}^*_{1_y} + 6\boldsymbol{P}^*_{2_y}\right) + \left(-3\boldsymbol{P}^*_{0_y} + \boldsymbol{P}^*_{1_y}\right)$$

Find all the real roots between $0 \leq t \leq 1$ using the quadratic formula and evaluate $P^*(t)$ at these moments. Then, draw the box described by a vector from the origin to the point $(x_{highest}, y_{highest})$. Reverse the transformations by reversing the standardization formula, apply it to the bounding box as well.

$$P_n = \left(P^*_n{}_x + iP^*_n{}_y\right)\left(\cos\left(\arctan\left(\frac{P_{1_y}}{P_{1_x}}\right)\right) + i\sin\left(\arctan\left(\frac{P_{1_y}}{P_{1_x}}\right)\right)\right) + P_{0_x} + P_{0_y}i$$

Bezier further

This mathematics can be extended into the three-dimensional space, and is used by the biggest companies in the animation industry to model things like light and hair. In order to know if a light ray is supposed to shine upon an object, the object must first be modelled using Bezier curves. Then, the computer calculates if the light ray crosses the bounding box of the curve, in order to decide if it should perform the more expensive calculation of figuring out how exactly the light affects the shading.

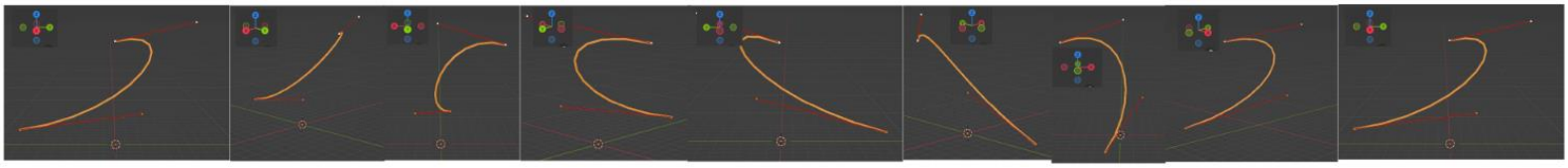


Figure 24. A 3D cubic Bezier curve viewed at from angles varying by 45 degrees

Finding an efficient algorithm for plotting the bounding boxes of 3D curves is crucial to save computational time and effort. The process of extending the math to the 3rd dimension is pretty straight forward:

Firstly, let's define our axes. The x and y axis are going to be the left-right and up-down axes respectively, just as in 2D, whereas the newly added z axis is going to be into-and-out-of the page.

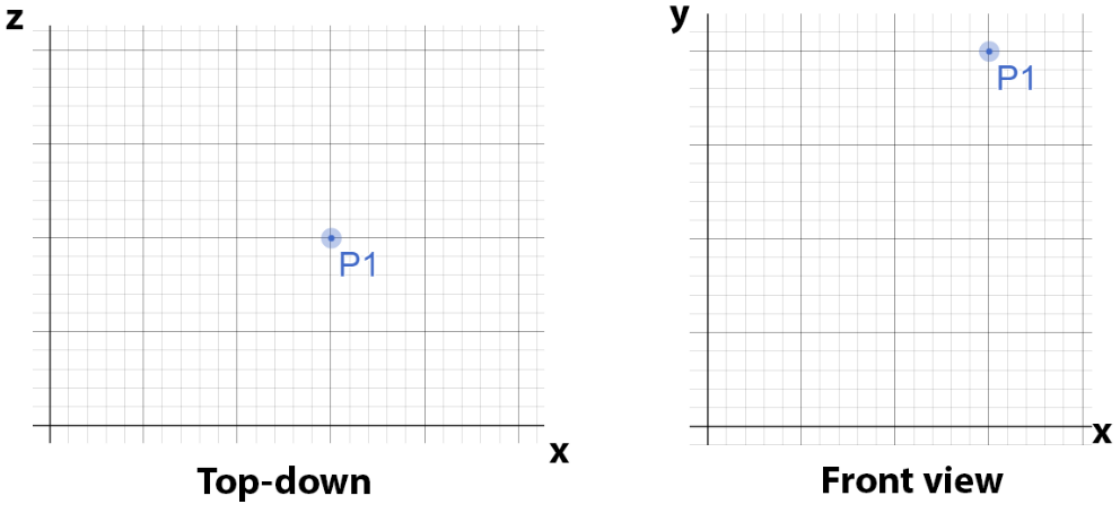


Figure 25. A point described in three dimensions

Let $P_0, P_1, P_2, \dots, P_n$ be points in the three-dimensional space, and again interpolate through them until you get to a level with only a single line segment with the final moving point P. Because there is no 3-dimensional number system analogous to the 2-dimensional complex numbers, I will use slightly different notation. In order to find the tightest-possible cuboid-shaped box for the curve to fit into, I again need to standardize the Bezier. Translating the curve to start at the origin doesn't require any more work than in 2 dimensions, we just have to translate all the control points by the vector $\begin{pmatrix} -P_{0_x} \\ -P_{0_y} \\ -P_{0_z} \end{pmatrix}$.

The process of rotating the curve in such a way that the point P_1 lies on one of the axes requires one more rotation than the 2-dimensional case, to account for the extra axis. Let's look at an arbitrary point P_1 from a top-down perspective and a front-view perspective.

Let's draw the necessary angles and label the lengths.

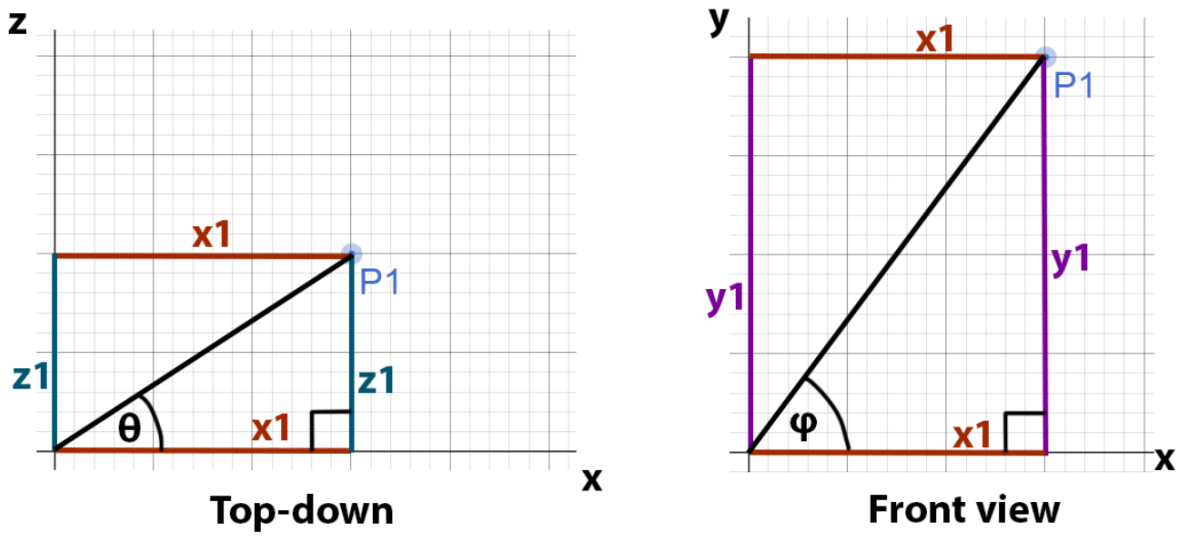


Figure 26. 3D point P1 annotated

The strategy is to first rotate the whole curve around the z-axis, by the angle $-\varphi$, causing P_1 to lay on the $x-z$ plane, and then to rotate the whole curve again around the y-axis, by the angle $-\theta$, causing P_1 to be on the x-axis. The order of operations doesn't play a role here and the angle between the z and y axes could have also been used, but it's redundant here as you only need two rotations. It is important to use the same rotations in reversed order when undoing the standardization after plotting the bounding box. Using the $tan = \frac{opposite}{adjacent}$ relationship we obtain:

$$tan\theta = \frac{P_{1_z}}{P_{1_x}}$$
$$\theta = \arctan\left(\frac{P_{1_z}}{P_{1_x}}\right)$$

$$tan\varphi = \frac{P_{1_y}}{P_{1_x}}$$
$$\varphi = \arctan\left(\frac{P_{1_y}}{P_{1_x}}\right)$$

We will use rotation matrices to formulate the $3D$ rotations. Let P_n^* be the point after rotations. Just as in two dimensions we can rotate using the complex numbers, three-dimensional rotations are represented by rotation matrices. Upon carrying out the matrix multiplications, you get the points rotated by a certain axis. I used the z-axis rotation matrix and then the y-axis rotation matrix.

$$P_n^* = \left(\left(\begin{pmatrix} P_{n_x} \\ P_{n_y} \\ P_{n_z} \end{pmatrix} - \begin{pmatrix} P_{0_x} \\ P_{0_y} \\ P_{0_z} \end{pmatrix}\right)\begin{bmatrix} cos -\arctan\left(\frac{P_{1_z}}{P_{1_x}}\right) & -sin -\arctan\left(\frac{P_{1_z}}{P_{1_x}}\right) & 0 \\ sin -\arctan\left(\frac{P_{1_z}}{P_{1_x}}\right) & cos -\arctan\left(\frac{P_{1_z}}{P_{1_x}}\right) & 0 \\ 0 & 0 & 1 \end{bmatrix}\right)\begin{bmatrix} cos -\arctan\left(\frac{P_{1_y}}{P_{1_x}}\right) & 0 & sin -\arctan\left(\frac{P_{1_y}}{P_{1_x}}\right) \\ 0 & 1 & 0 \\ -sin -\arctan\left(\frac{P_{1_y}}{P_{1_x}}\right) & 0 & cos -\arctan\left(\frac{P_{1_y}}{P_{1_x}}\right) \end{bmatrix}$$

The curve I am going to use as example of the algorithm is going to be defined by the following control points:

$$P_0 = (-1,0,0)$$
$$P_1 = (0,1,-1)$$
$$P_2 = (2,2,1)$$
$$P_3 = (1,3,1)$$

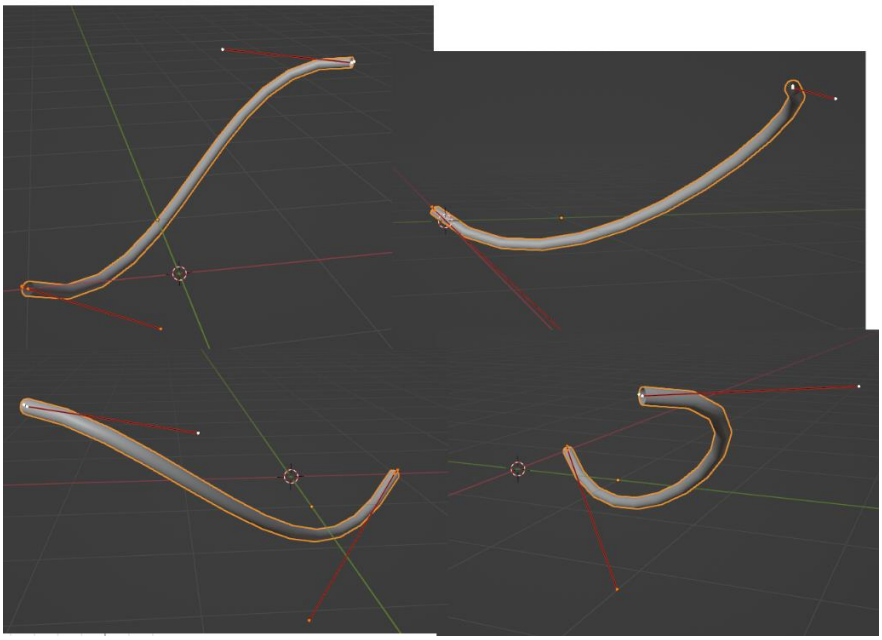


Figure 27. The example 3D cubic Bezier cube viewed from different angles

I am going to translate the curve to start at the origin (the thin circle shape where the colored axes meet on the image) by moving all points by the vector $1\boldsymbol{i}$ Now I need to find the angles by which I will rotate the curve by substituting in the translated values into the formulas we derived before -

$$\theta = \arctan\left(\frac{P_{1_z}}{P_{1_x}}\right) = \arctan\left(\frac{-1}{1}\right) = \arctan(-1) = -\frac{\pi}{4}$$
$$\varphi = \arctan\left(\frac{P_{1_y}}{P_{1_x}}\right) = \arctan\left(\frac{1}{1}\right) = \frac{\pi}{4}$$

Let's calculate all the standardized points by using the formula

$$P_n^* = \left(\left(\begin{pmatrix} P_{n_x} \\ P_{n_y} \\ P_{n_z} \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}\right)\begin{bmatrix} \cos\left(\frac{\pi}{4}\right) & -\sin\left(\frac{\pi}{4}\right) & 0 \\ \sin\left(\frac{\pi}{4}\right) & \cos\left(\frac{\pi}{4}\right) & 0 \\ 0 & 0 & 1 \end{bmatrix}\right)\begin{bmatrix} \cos\left(-\frac{\pi}{4}\right) & 0 & sin\left(-\frac{\pi}{4}\right) \\ 0 & 1 & 0 \\ -sin\left(-\frac{\pi}{4}\right) & 0 & cos\left(-\frac{\pi}{4}\right) \end{bmatrix}$$

$$P_0^* = (0,0,0)$$
$$P_1^* = (\sqrt{3},0,0)$$
$$P_2^* = (2.304, 2.121, 2.046)$$
$$P_3^* = (2.306, 0.8, 2.85)$$

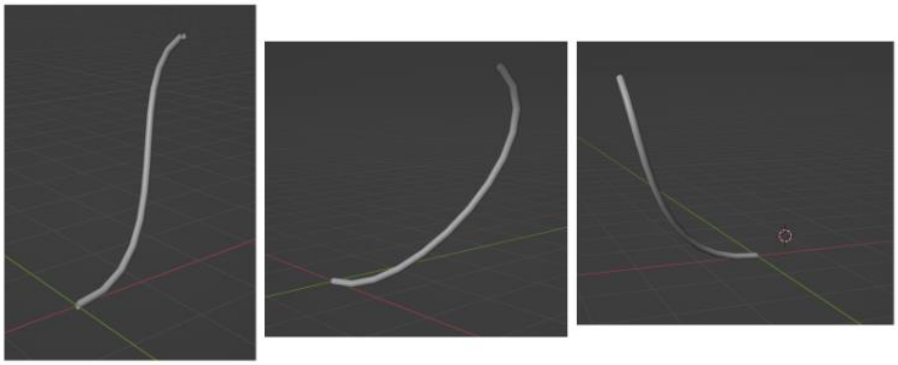


Figure 28. Standardized 3D Bezier Curve

Now that our curve is standardized, we need to split it into three separate curves, as each axis has its own function. I am going to assume I am always working with a cubic Bezier, although the calculations don't get much harder for higher order curves. These are the general formulae:

$$P^{*\prime}{}_x(t) = t^2\left(-3P^*_{0_x} + 9P^*_{1_x} - 9P^*_{2_x} + 3P^*_{3_x}\right) + t\left(6P^*_{0_x} - 12P^*_{1_x} + 6P^*_{2_x}\right) + \left(-3P^*_{0_x} + P^*_{1_x}\right)$$

$$P^{*\prime}{}_y(t) = t^2\left(-3P^*_{0_y} + 9P^*_{1_y} - 9P^*_{2_y} + 3P^*_{3_y}\right) + t\left(6P^*_{0_y} - 12P^*_{1_y} + 6P^*_{2_y}\right) + \left(-3P^*_{0_y} + P^*_{1_y}\right)$$

$$P^{*\prime}{}_z(t) = t^2\left(-3P^*_{0_z} + 9P^*_{1_z} - 9P^*_{2_z} + 3P^*_{3_z}\right) + t\left(6P^*_{0_z} - 12P^*_{1_z} + 6P^*_{2_z}\right) + \left(-3P^*_{0_z} + P^*_{1_z}\right)$$

And these are the formulae with values from the example substituted in:

$$P^{*\prime}{}_x(t) = t^2\left(9\sqrt{3} - 9(2.304) + 3(2.306)\right) + t\left(12\sqrt{3} + 6(2.304)\right) + \sqrt{3}$$

$$P^{*\prime}{}_y(t) = t^2(-9(2.121) + 3(0.8)) + t(6(2.121)) +$$

$$P^{*\prime}{}_z(t) = t^2(-9(2.046) + 3(2.85)) + t(6(2.046)) +$$

I use the quadratic formula to find all real roots between [0,1] and evaluate $P^*(t)$ at these values. Then, I plot the box from the origin to the end of the vector $\begin{pmatrix} x_{highest} \\ y_{highest} \\ z_{highest} \end{pmatrix}$ out of all the points I evaluated.I reverse the standardization operations while applying them also to the vertices making up the box using this formula.

$$P_n = \left(\begin{pmatrix} P^*_{n_x} \\ P^*_{n_y} \\ P^*_{n_z} \end{pmatrix}\begin{bmatrix} cos\arctan\left(\frac{P_{1_y}}{P_{1_x}}\right) & 0 & sin\arctan\left(\frac{P_{1_y}}{P_{1_x}}\right) \\ 0 & 1 & 0 \\ -sin\arctan\left(\frac{P_{1_y}}{P_{1_x}}\right) & 0 & cos\arctan\left(\frac{P_{1_y}}{P_{1_x}}\right) \end{bmatrix}\right)\begin{bmatrix} cos\arctan\left(\frac{P_{1_z}}{P_{1_x}}\right) & -sin\arctan\left(\frac{P_{1_z}}{P_{1_x}}\right) & 0 \\ sin\arctan\left(\frac{P_{1_z}}{P_{1_x}}\right) & cos\arctan\left(\frac{P_{1_z}}{P_{1_x}}\right) & 0 \\ 0 & 0 & 1 \end{bmatrix} + \begin{pmatrix} P_{0_x} \\ P_{0_y} \\ P_{0_z} \end{pmatrix}$$

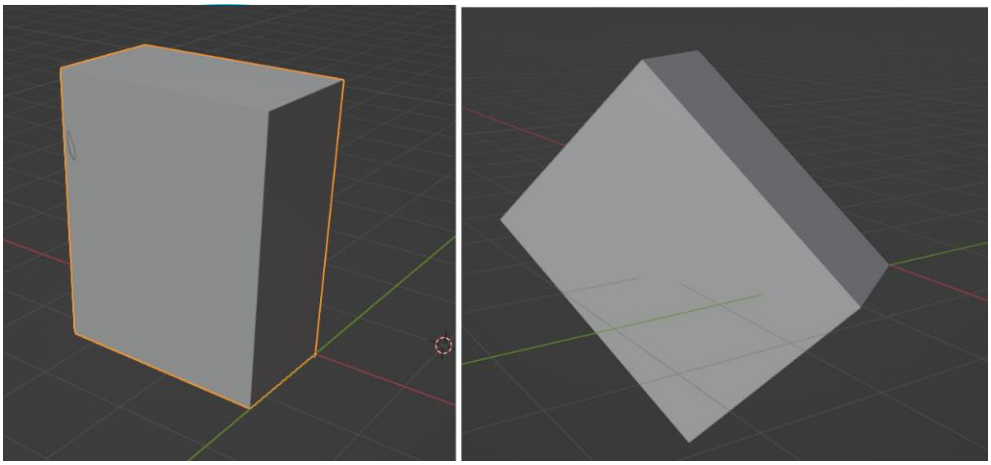


Figure 29. The Bounding Cuboid before and after reversing the standarization

Conclusion

I effectively devised an algorithm for building a three-dimensional bounding box building upon my research about the behavior of Bezier curves I've attempted to learn about. The goal of the exploration was fulfilled, as I now can predict how a curve will behave analytically as opposed to just examining it empirically after the computer has done the work for me. There are other interesting aspects of the Bezier curves such as calculating the curvature at a particular point to find if a curve is sufficiently curved or straight which provides a lot of utility but couldn't be touched upon in this essay because of length constraints. Calculus proves to be an invaluable tool in designing a tight bounding box and complex number based rotation is helpful to standardize the process and catch any better solutions.

Bibliography:

References

Chang, G. and Sederberg, T., 1997. *Over and over again*. Washington, DC: Mathematical Association of America.

Havil, J., 2019. *Curves for the Mathematically Curious : An Anthology of the Unpredictable, Historical, Beautiful, and Romantic*. Princeton University Press.

Long, C. and Norton, V., 1980. Bezier Polynomials in Computer-Aided Geometric Design. *The Two-Year College Mathematics Journal*, 11(5), p.320.

SAHINER, A., YILMAZ, N. and KAPUSUZ, G., 2017. A descent global optimization method based on smoothing techniques via Bezier curves. *Carpathian Journal of Mathematics*, 33(3), pp.373-380.