**Keyframe Interpolation**

Introduction

In computer animation a technique called keyframing is used in which important frames of an animation are drawn or posed, the in-between frames are then drawn to create the illusion of motion. However, manually creating each individual frame by hand is very time consuming which is why a lot of modern animation automatically generates the frames based on the animators needs. These needs might include having an animated car move with a constant speed from point A to point B or if the car was already stationary, have it accelerate toward point B. This problem can be solved using a method called interpolation. We have two problems we need to solve: creating a curved path in 2D space from a given objects keyframe positions and controlling the speed of the object along that path whilst being tied to a specific frame rate.

**Moving between two points: Linear Interpolation**

The simplest form of interpolation is linear interpolation: given two points it calculates in-between positions on a straight line between those two points based on a variable *t* which can be between 0 and 1. Furthermore, if the spacing in *t* is equal as it goes from 0 to 1 then the points being generated are also equally spaced. Linear interpolation would allow us to interpolate an object between two points at a constant speed.

On the left is a visual representation of X(t). On the right shows a line made from both equations with regular intervals of *t*.

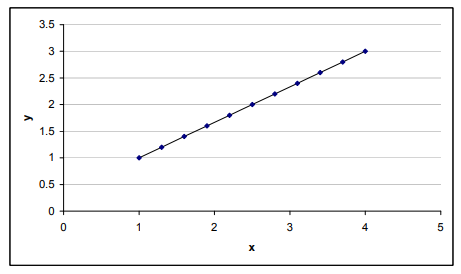


Fig 2. From page 5. <https://nccastaff.bmth.ac.uk/hncharif/MathsCGs/Interpolation.pdf>

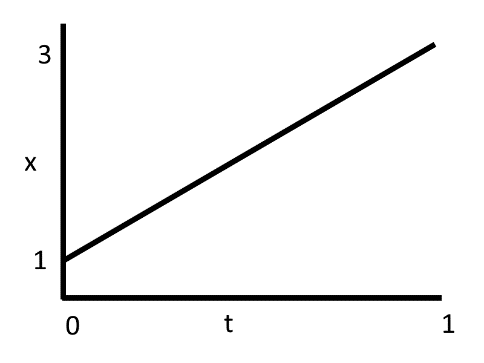


Fig 1. Blending function for x

**Non-linear Interpolation**

Non-linear interpolation is different to linear interpolation in that the ratio of spacing in *t* wont necessarily be the same as the output. This can create smooth motion like how physical object accelerate and decelerate in the real world. An example of this type of interpolation is trigonometric interpolation which uses a combination of sin and cosine functions that take a value of *t* that is between 0 and π/2 radians.

**Trigonometric interpolation**

On the left is an example of X(t) and how and add together to give interpolated values between 1 and 3 (The blue curve). On the right is the resulting interpolated values between the two end points.

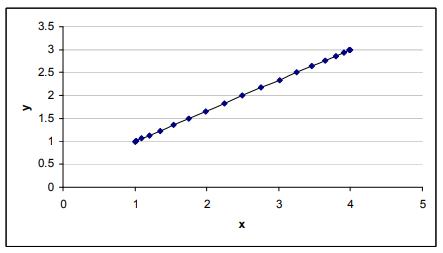
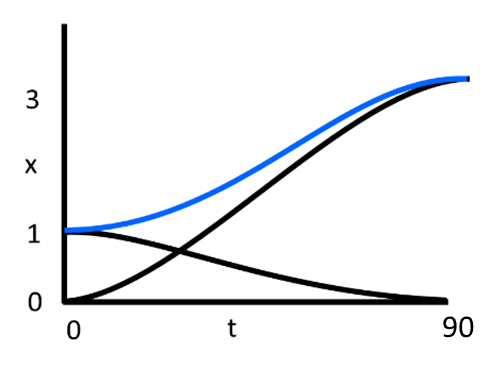


Fig 4. From page 10. <https://nccastaff.bmth.ac.uk/hncharif/MathsCGs/Interpolation.pdf>

Fig 3. Blending function for x. The 2 black curves are and .



Another example is cubic interpolation that produces similar results to the trigonometric version.

**Cubic interpolation**

One main difference is that the parameter *t* is between 0 and 1 rather than 0 and which is a lot more useable compared to radians.

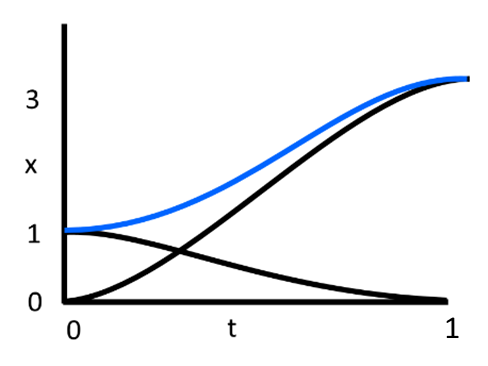


Fig 5. Blending function for x. The 2 black curves are and

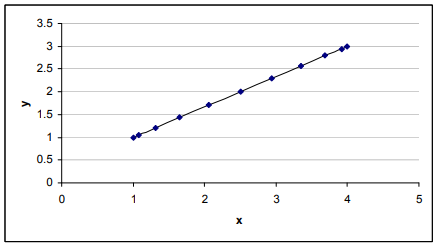


Fig 6. From page 14. <https://nccastaff.bmth.ac.uk/hncharif/MathsCGs/Interpolation.pdf>

These methods of interpolation are useful for certain circumstances and give us a good insight into how interpolation works. However, we still don’t have much control over the points being generated other than the variable *t* and that only describes how far the interpolation is between end points. What if we wanted to generate points that that result in a curved path rather than a straight path?

**Creating a curve between points: Bezier Curve**

Bezier curves go through the end control points and use any in-between control points as a suggestion for the curve. A Linear Bezier curve is no different to linear interpolation and consists of the two end points, a quadratic Bezier curve has three control points and a cubic Bezier curve has four control points. Bezier curves can keep increasing to the nth order but will increase in computational cost as *n* increases.

Starting simple with a quadratic Bezier curve, you can think of it as calculating in-between points between three sets of two end points – this is called DeCasteljau’s algorithm. One set of end points is directly taken from the in-between points of the other two sets of end points and the resulting curve is made from the linear interpolated points between the resulting end points.

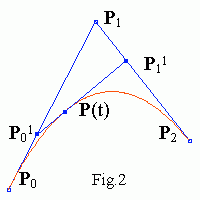


Fig 7. Example of quadratic Bezier curve with interpolation. From: <https://www.ibiblio.org/e-notes/Splines/bezier.html>

The above calculation is enough to code this curve, however, the whole thing can be put into one equation. Sub in and :

This equation can be constructed using another equation that can be applied to all orders of Bezier curve. It is constructed from Bernstein polynomials.

If we make P1 a movable point, we can control how stretched the curve is, this gives us more control over creating a path that an object might follow. If we want more control be can go up an order to a cubic Bezier curve which offers a second controllable point. Cubic Bezier curves allow you to create a curve like the quadratic method but also create curves like the image on the right.

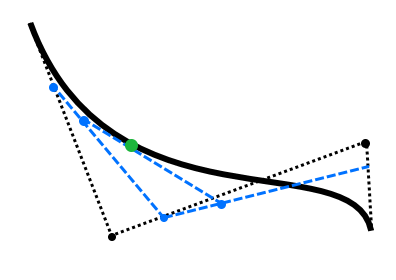
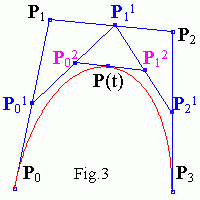


Fig 9. Example of cubic Bezier curve

Fig 8. Example of cubic Bezier curve from: <https://www.ibiblio.org/e-notes/Splines/bezier.html>

Stopping at cubic Bezier curves is a good idea, since it offers a lot of control over a curve, and any Bezier curves with an order that exceeds cubic starts to become more computationally expensive since were exponentially increasing the number of iterations where we linearly interpolate. However, there is a work around, Bezier and other curves methods can be pieced together to create longer curves without exponentially increasing the cost to compute.

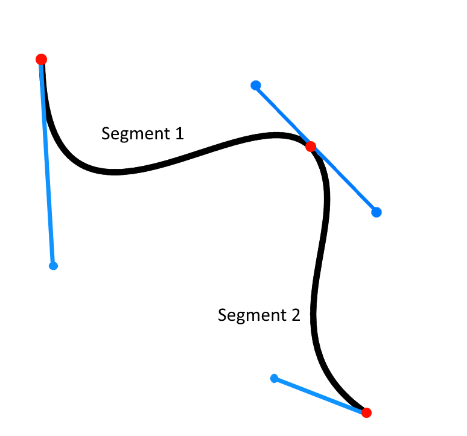


Fig 10. Piecewise Bezier curve

Above is an example of two connected cubic Bezier curves. For two Bezier curves to smoothly connect the two tangents formed either side of the middle end point must be equal in magnitude and opposite in direction. Bezier curves offer a lot control as they can be easily manipulated using control points to push and pull the curve in certain ways. However, what we just want a curve that goes through all the points we give it?

**Cardinal Splines and Catmull-Rom Splines**

Cardinal splines are a series of connected curves, so in the piecewise Bezier example above the curve would be considered a cardinal spline. Catmull-Rom splines are a type of cardinal spline made from multiple Hermite splines. Hermite spline interpolation creates a curve using two control points and two tangents. The tangents control the initial and end directions and the magnitude controls how much it curves.

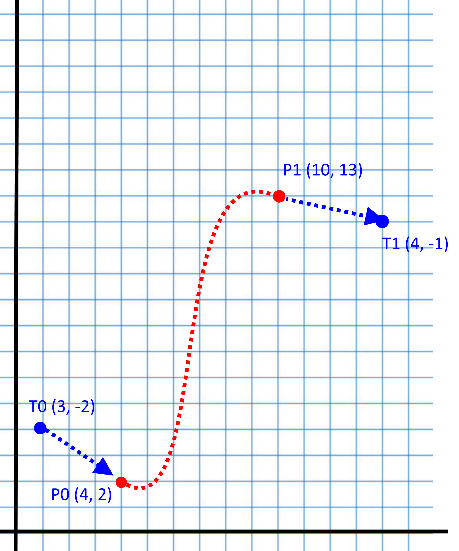


Fig 11. Hermite Curve

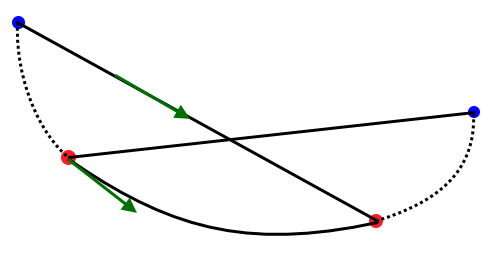
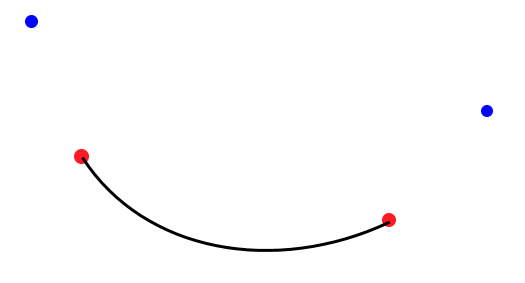
To go from individual Hermite splines to Catmull-Rom requires replacing the tangents. This is achieved by creating a tangent from adjacent control points e.g. the tangent for P1 would be (P2 – P0). We can also control the tension of the curve by adding a constraint to the tangent, α, which controls the magnitude.

Fig 12. Catmull-Rom spline



P-1

P0

P1

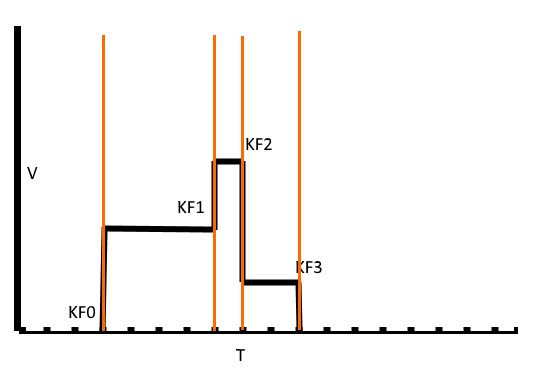
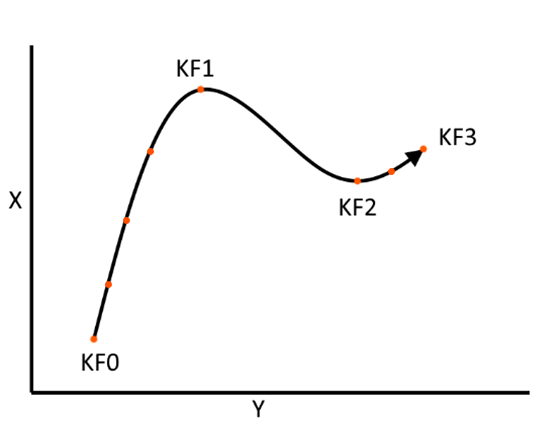
P2

P(t)

Fig 13. Interpolation equation for Catmull-Rom spline at time t. For derivation see appendices.

**Consolidation**

Using arc length reparameterization I can get the length of the curve between keyframes and place the dividing frames at correct intervals. Below: Given set time keyframes and a distance between them there can be a calculated resultant speed between keyframes. This wouldn’t work in reverse: if we want to increase the speed at which an object travels between KF0 and KF1 the distance between KF0 and KF1 would have to increase or the time between KF0 and KF1 would have to decrease. Speed = distance/time. If we control the time and positions of the keyframes we cant directly control the speed between them, we have to manipulate the time and distances to get a speed we desire. If we wanted to control the speed directly we would have to give up control of either the keyframes times or positions. Since we don’t want the path to physically change we are left with giving up control of the keyframe times.

****

Using this we can create a path using keyframe positions as control points. The keyframes also have time stamps so the speed can be control by changing the distance or time between points.

We now have two ways of creating curves with reasonable amounts of control, however, there is still issues that need solving. We have a way to interpolate between two points linearly and along multiple types of curve. Lets say we animate a car with a constant framerate, we want control over the speed and the path the car follows. Controlling the speed is the easy part since all we need to do is define speeds at keyframes and use linear interpolation to create acceleration. Below shows how you could plot speeds on a graph and get the resulting motion over time on the right.

The hard part is when we want this to happen along a defined path.

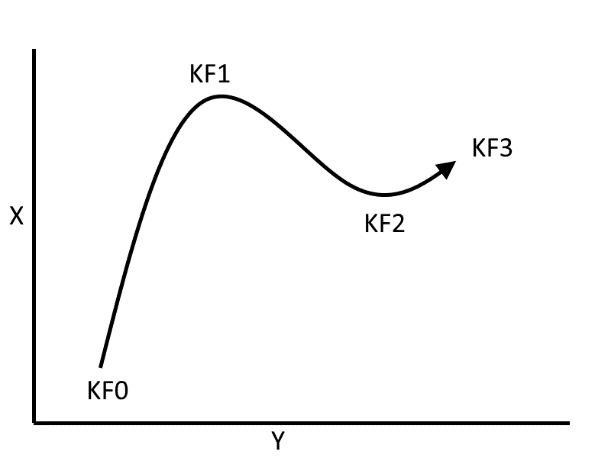
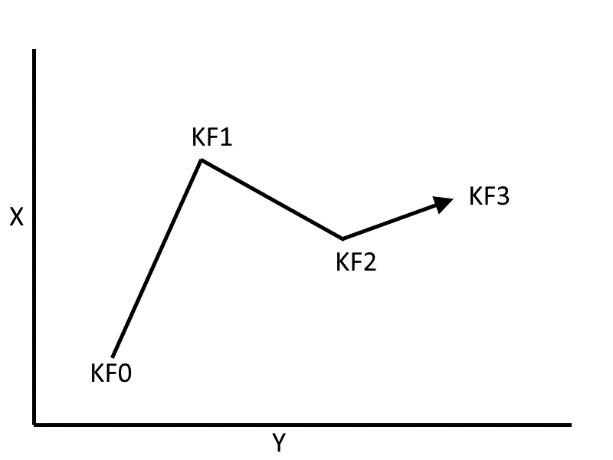
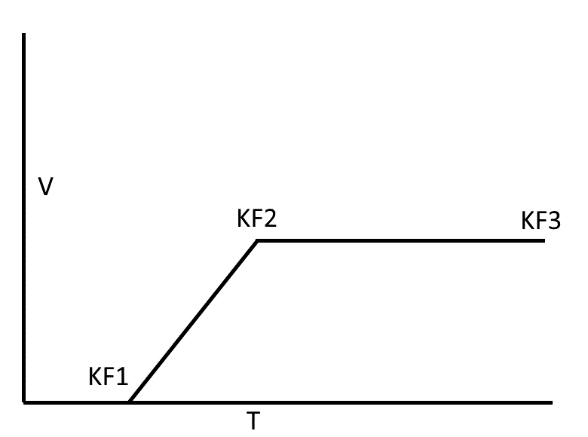
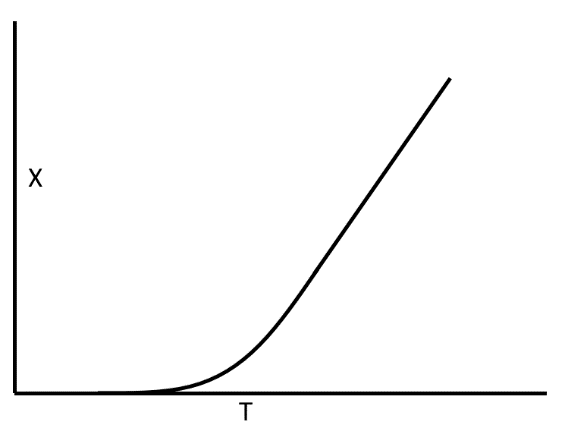


Fig 15. Example of straight and curved paths that an animated object might follow

Fig 14. Example of using linear interpolation for controlling speed and the resulting curve in motion



From the velocity-time graph we can work out the displacement for each keyframe

<https://codeplea.com/simple-interpolation>

<https://codeplea.com/introduction-to-splines>

<https://pages.cpsc.ucalgary.ca/~jungle/587/pdf/5-interpolation.pdf>